

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Evaluation of Distributed Statistical Learning

André Ismael Ferraz Ávila



Mestrado em Engenharia Informática e Computação

Supervisor: Prof. Carlos Miguel Ferraz Baquero-Moreno

Second Supervisor: Prof. Daniel Jorge Barros Tinoco

July 29, 2025

Evaluation of Distributed Statistical Learning

André Ismael Ferraz Ávila

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

President: Prof. José Magalhães Cruz

Examiner: Prof. João Marco Silva

July 29, 2025

Abstract

Natural Language Processing models have gained significant attention due to the development of large-scale models such as OpenAI's GPT. These models rely on extensive and diverse datasets, which presents data-sharing challenges such as privacy and ownership.

Federated Learning addresses those challenges by allowing multiple actors to collaboratively train a shared model without exchanging the raw data. Not sharing the raw data enables the use of data that, in normal conditions, could not be shared due to privacy, ethical or legal concerns. This decentralised approach minimises the central storage requirements while also lessening data privacy risks.

Statistical learning algorithms are commonly used in Federated Learning. However, they must be adapted into distributed statistical learning algorithms in order to handle decentralised data. These distributed algorithms are being developed and, therefore, must obtain empirical results to assess their theoretical foundations. Due to the distributed nature of the algorithms, performing an empirical evaluation is a complex task, as the environments these algorithms operate in, and consequently, the adversities they encounter are difficult to replicate physically and consistently.

This dissertation aims to support the development, improvement and analysis of distributed statistical learning algorithms by introducing an evaluation framework implemented as a discrete-event simulator. The existent discrete-event simulators are compared and analysed with the evaluation of the target algorithms in mind. Then, a simulator is designed and purpose-built to be extensible, configurable and observable.

The developed simulator is validated by comparing its functioning to that of an already-established simulator, and its metrics visualisation capabilities are demonstrated. Furthermore, the simulator is used to evaluate a distributed statistical learning algorithm. Based on the evaluation results, a solution is proposed to address the algorithm's identified functional shortcomings. The proposed solution is also evaluated using the designed simulator, and its results are compared to those of the original implementation.

Keywords: Distributed Statistical Learning, Discrete Event Simulation, Algorithm Simulation, Distributed Systems Simulation, Evaluation of Distributed Systems

Resumo

O desenvolvimento de modelos de larga escala, como o GPT da OpenAI, trouxe visibilidade significativa aos modelos de Processamento de Linguagem Natural. Estes modelos dependem de conjuntos de dados extensos e diversos, o que traz desafios quanto à partilha de dados, como a privacidade e a propriedade.

A Aprendizagem Federada lida com esses desafios ao permitir que múltiplos participantes colaborem no treino de um modelo partilhado sem que os dados originais sejam partilhados. A não partilha dos dados originais permite o uso de dados que, em condições normais, não poderiam ser partilhados devido a questões de privacidade, questões éticas ou legais. Esta abordagem descentralizada minimiza a necessidade de armazenamento centralizado, ao mesmo tempo que reduz os riscos relacionados à privacidade dos dados.

Os algoritmos de aprendizagem estatística são normalmente utilizados em Aprendizagem Federada. No entanto, devem ser adaptados para algoritmos de aprendizagem estatística distribuída, de modo a poderem lidar com dados descentralizados. Estes algoritmos distribuídos estão a ser desenvolvidos e, por isso, precisam de resultados empíricos para validar as suas bases teóricas. Devido à natureza distribuída dos algoritmos, a realização de uma avaliação empírica é uma tarefa complexa, pois os ambientes em que estes algoritmos operam, e consequentemente, as adversidades que encontram, são difíceis de replicar fisicamente e consistentemente.

Esta dissertação tem como objetivo apoiar o desenvolvimento, aprimoramento e análise de algoritmos de aprendizagem estatística distribuída, por meio da introdução de uma estrutura de avaliação implementada através do uso de um simulador de eventos discretos. Os simuladores de eventos discretos existentes são comparados e analisados tendo em conta a avaliação dos algoritmos-alvo. De seguida, um simulador é planeado e desenvolvido com o objetivo de ser extensível, configurável e observável.

O simulador desenvolvido é validado ao comparar o seu funcionamento com o de um simulador já estabelecido, e as suas capacidades de visualização de métricas são demonstradas. Além disso, o simulador é usado para avaliar um algoritmo de aprendizagem estatística distribuída. Com base nos resultados da avaliação, é proposta uma solução que resolva as lacunas funcionais do algoritmo. A solução proposta é também avaliada utilizando o simulador desenvolvido e os resultados são comparados com os resultados da implementação original.

Keywords: Aprendizagem Estatística Distribuída, Simulação de Eventos Discretos, Simulação de Algoritmos, Diminuição de Sistemas Distribuídos, Avaliação de Sistemas Distribuídos

UN Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) provide a global framework to achieve a better and more sustainable future for all. It includes 17 goals to address the world's most pressing challenges, including poverty, inequality, climate change, environmental degradation, peace, and justice.

This thesis contributes to this mission by providing a framework for evaluating distributed statistical learning algorithms. This framework is a discrete-event simulator which allows researchers to obtain empirical results that validate the simulated algorithm's theoretical foundations. It is a less expensive and faster method of obtaining empirical results than the alternative, which is to replicate the environment in which the target algorithm operates physically.

Federated Learning is an active research topic in the Artificial Intelligence (AI) field, which adapts statistical learning algorithms to privacy-preserving, decentralized settings. Therefore, it does not require centralized datasets and, consequently, centralized data infrastructure.

The specific Sustainable Development Goals mentioned have the following names:

SDG 9 Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation.

SDG 16 Promote peaceful and inclusive societies for sustainable development, provide access to justice for all and build effective, accountable and inclusive institutions at all levels.

SDG 17 Strengthen the means of implementation and revitalize the Global Partnership for Sustainable Development.

SGD	Target	Contribution	Performance Indicators and Metrics
9	9.5	Allows for empirical results to be obtained during the research phase, being faster and less expensive than replicating a distributed system physically.	Number of researchers using the simulation framework; Number of statistical learning algorithms evaluated; Reduction in time and cost for distributed statistical learning algorithms simulation.
	9.b	Enables developing countries to participate in AI research without needing massive centralized datasets	Number of institutions participating in AI research through decentralized datasets; Amount of computational resources saved compared to centralized approaches; Percentage reduction in data infrastructure costs.
16	16.10	Supports fundamental privacy rights by enabling AI model development without exposing raw data, ensuring compliance with data protection regulations and international privacy agreements.	Number of privacy-compliant AI models developed; Reduction in data exposure incidents; Compliance rate with international data protection standards (GDPR, etc.).
17	17.6	Facilitates international cooperation in AI research by enabling cross-border knowledge sharing while respecting data privacy regulations and national sovereignty while encouraging both North-South and South-South collaboration.	Number of international collaborative projects using the framework; Number of cross-border federated learning studies conducted; Percentage of projects maintaining data sovereignty compliance; Number of developing countries participating in international AI research partnerships.

Agradecimentos

Primeiramente, gostaria de agradecer o apoio e orientação dos meus orientadores, Carlos Baquero-Moreno e Daniel Tinoco, cuja disponibilidade e feedback foram fundamentais ao longo da elaboração desta dissertação. Sinto-me grato por vos ter tido como orientadores e professores. O vosso conhecimento e empenho na partilha do mesmo influenciaram significativamente a minha jornada académica e o meu interesse por informática.

De seguida, quero agradecer a quem não influenciou diretamente a elaboração da tese mas que, mesmo assim, foram essenciais para que ela fosse possível. Obrigado aos meus pais. É graças ao vosso esforço e sacrifício que foi sequer possível concluir esta etapa. Agradeço o vosso apoio e a vossa confiança em mim, e por garantirem que nunca me faltasse nada. Mesmo longe, estiveram sempre perto. E por falar em perto, obrigado, Igor, pelo teu apoio e por, mesmo tendo uma agenda ocupada, me ajudares sempre que precisei. Obrigado também ao Simba, o nosso patudo, pelos vídeos engraçados. Como o pai diz que tudo o que temos é graças a elas, fica também um agradecimento às nossas vacas.

À minha namorada, Matilde, que mais do que ninguém, aturou os meus devaneios sobre a dissertação. Obrigado pelo teu apoio, constante presença e por tantas vezes me teres inspirado quando faltava a motivação.

Obrigado aos amigos que fiz durante a jornada académica. Tenho muita sorte por serem tantos e tão bons. Um agradecimento especial à malta do nosso arraial, à Ana Sofia, ao André, ao Dani, ao Dinis, à Lara, ao Martim, ao Miguel, ao Pedro e ao Fonseca pelos bons momentos que passámos juntos. E não haveria grupinho sem a Sofia à qual fico muito grato pelo companheirismo, bom humor e amizade. Um obrigado também ao Tony por me ter acompanhado todos os dias na biblioteca e por ser um amigo espetacular.

Por fim, e não menos importante, aos meus familiares na Terceira, um obrigado com saudade.

André Ávila

“Be curious, not judgemental.”

Ted Lasso

Contents

1	Introduction	1
1.1	Context	1
1.2	Motivation	2
1.3	Problem	2
1.4	Objectives	2
1.5	Document Structure	3
2	State of the Art	4
2.1	Simulation	4
2.2	Discrete-Event Simulation	4
2.3	Discrete-Event System Specification (DEVS)	5
2.4	Parallel Discrete-Event Simulation	6
2.4.1	Optimistic Synchronisation	7
2.4.2	Conservative Synchronisation	7
2.4.3	Relaxed Synchronisation	8
2.4.4	Combined Synchronisation	9
2.5	Distributed Discrete-Event Simulation	9
2.6	Simulation Performance	9
2.7	GPU-Based Discrete-Event Simulation	10
2.8	Existing Simulators	10
2.9	Conclusions	10
3	Simulators	12
3.1	Simulator Characterisation	12
3.1.1	Environment Simulation	13
3.1.2	Metric Visualisation	14
3.1.3	Scenario Configuration	14
3.1.4	Fault Injection	14
3.1.5	Reproducibility	15
3.2	Sinalgo	15
3.2.1	Environment Simulation	15
3.2.2	Metric Visualisation	17
3.2.3	Scenario Configuration	21
3.2.4	Fault Injection	21
3.2.5	Reproducibility	23
3.3	QUANTAS	24
3.3.1	Environment Simulation	24
3.3.2	Metric Visualisation	25

3.3.3	Scenario Configuration	26
3.3.4	Fault Injection	27
3.3.5	Reproducibility	27
3.4	SimGrid	28
3.4.1	Environment Simulation	28
3.4.2	Metric Visualisation	29
3.4.3	Scenario Configuration	30
3.4.4	Fault Injection	31
3.4.5	Reproducibility	32
3.5	Comparison	32
3.5.1	Environment Simulation	32
3.5.2	Metric Visualisation	33
3.5.3	Scenario Configuration	34
3.5.4	Fault Injection	34
3.5.5	Reproducibility	34
3.5.6	Conclusions	35
4	Designing a Discrete Event Simulator	36
4.1	Design Philosophy and Requirements	36
4.2	Core Architecture Overview	40
4.3	Event System Design	43
4.3.1	Events	43
4.3.2	Event Queue	43
4.3.3	Simulation State	44
4.4	Network Topology and Communication	45
4.4.1	Peers	46
4.4.2	Connections	47
4.4.3	Messages	48
4.4.4	Communication	48
4.5	Timers and Asynchronous Logic	49
4.6	Observability and Monitoring	50
4.6.1	Logging Architecture	51
4.6.2	Behaviour Logging	52
4.6.3	Metrics Logging	53
4.7	Fault Injection	55
4.7.1	Message-Based Faults	55
4.7.2	Peer-Based Faults	57
4.8	Reproducibility	58
4.9	Registry Systems	59
4.9.1	Scenario Registry	60
4.9.2	Topology Registry	61
4.9.3	Arrival Time Registry	62
4.10	Configuration Management	64
4.10.1	Command Line Argumentes	64
4.10.2	Configuration File	66
4.11	Implementation Considerations	73

5	Experimental Evaluation	74
5.1	Simulator Validation	74
5.1.1	Message Passing Scenario	75
5.1.2	Flow Updating Scenario	87
5.2	Statistical Learning Algorithm Evaluation	98
5.2.1	Algorithm	98
5.2.2	Implementation	99
5.2.3	Experiments	103
5.2.4	Evaluation Overview	113
5.2.5	Proposed Solution	114
6	Conclusion	124
6.1	Main Contributions	124
6.2	Research Answers	125
6.2.1	RQ1. How can the state-of-the-art discrete-event simulation methodologies be used to evaluate distributed algorithms efficiently?	125
6.2.2	RQ2. How can distributed statistical learning be specified and modelled for discrete-event simulation?	125
6.2.3	RQ3. How is a discrete-event simulator for the purpose of evaluating distributed statistical learning algorithms designed?	125
6.2.4	RQ4. How can we characterise and evaluate a distributed algorithm?	126
6.3	Future Work	126
	References	128
A	Designing a Discrete Event Simulator	131
A.1	Simulator Structure	132
A.2	Code Definitions	133
B	Experimental Evaluation	141
B.1	Simulator Validation	141
B.1.1	Message Passing Scenario	141
B.1.2	Flow Updating Scenario	144
B.2	Statistical Learning Algorithm Evaluation	158
B.2.1	Implementation	158
B.2.2	Experiments	174
B.2.3	Proposed Solution	176

List of Figures

3.1	Pop-up interface used to create nodes	18
4.1	The simulator's logic upon starting	42
4.2	Selected items' reference graph	45
4.3	Histogram of jitter values up to the 95th percentile	57
4.4	Weibull fit to jitter data	57
5.1	Two peers connected by a bidirectional link	76
5.2	Five peers connected by bidirectional links in a star topology	80
5.3	Four peers connected by bidirectional links in a star topology	84
5.4	Fully connected experiment topology with values	91
5.5	Average value over time by peer in the designed simulator	96
5.6	Average value over time by peer in SimGrid	96
5.7	Global difference between the calculated and the actual average value over time in the designed simulator	97
5.8	Global difference between the calculated and the actual average value over time in SimGrid	97
5.9	Average value over time by peer with no faults	97
5.10	Average value over time by peer with dropped messages	97
5.11	Global difference between the calculated and the actual average value over time with no faults	98
5.12	Global difference between the calculated and the actual average value over time with dropped messages	98
5.13	Fully connected experiment topology	107
5.14	Custom ring topology	109
5.15	Peer requesting R matrix from its neighbour	115
5.16	Peer requesting R matrix from its neighbour through a timeout	116
5.17	Fully connected topology with known identifiers and hash	118
5.18	Non fully connected topology with known identifiers and hash	119
5.19	An example of how the timeout mechanism propagates messages	121
A.1	The created simulator's structure	132

List of Tables

5.1 Comparison of Distributed Generalised Linear Model Algorithm Experiments Results 122

List of Acronyms

AI	Artificial Intelligence
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DEVS	Discrete-Event System Specification
DoD	US Department of Defense
GDPR	General Data Protection Regulation
GLM	Generalised Linear Model
GPGPU	General Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
GUI	Graphical User Interface
GVT	Global Virtual Time
JSON	JavaScript Object Notation
LBTS	Lowest Bound Timestamp
MPI	Message Passing Interface
OSI	Open Systems Interconnection
QUANTAS	Quantitative User-friendly Adaptive Networked Things Abstract Simulator
Sinalgo	Simulator for Network Algorithms
UI	User Interface
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

Chapter 1

Introduction

This chapter contextualises the problem (Section 1.1), presents our motivation (Section 1.2), exposes the problem (Section 1.3), explains our objective (Section 1.4), and describes the document structure (Section 1.5).

1.1 Context

In recent years, Natural Language Processing models have been in the spotlight due to, and not only due to, OpenAI's GPT model. The GPT4 model requires a vast dataset mainly consisting of publicly available data. As data is very valuable, data sharing becomes essential, not only to increase the amount of data but also to diversify it. However, sharing data brings concerns, such as privacy and ownership. In 2017, in a blog post [21], Google announced that for their keyboard, GBoard, they would be using a new notion called Federated Learning. The objective was to achieve low latency by training the machine learning model behind the keyboard locally with data gathered from the phone's input while also maintaining a single centralised model trained by a large number of users. Google managed to fulfil its objective by having the user train a model on its machine and then sharing the resulting model with the network. By having multiple users perform this task and then combining their results, Google created a single centralised model that could then be used. The concept of having multiple actors perform their calculations and then combining their results into a single model without the need for sharing the training data is the mentioned notion of Federated Learning. A related use for Federated Learning can be that of decentralising Natural Language Processing models [2], allowing for data to be captured in single independent devices fulfilling the data model's data needs. Minimising the amount of data that needs to be stored in a central server cuts costs and can lessen privacy risks.

1.2 Motivation

Federated learning is an active topic of research. It can utilise data that would otherwise be unreachable due to privacy, ethical, or legal concerns without the need to be aware of its value. The ability to use non-disclosed data is crucial for use cases where data is partitioned and private but should be combined. For example, if a group of hospitals would like to create a machine learning model for detecting breast cancer, as per their data, they would build a model that is not as reliable as it could be because it is limited by the data the hospital has access to, seeing as medical records usually cannot be shared between hospitals for this purpose, they are not able to request other hospitals for more training data. In this example, a single model could be created by sharing the machine learning model's parameters after being trained independently by each hospital. By sharing and combining parameters instead of the actual training data, multiple hospitals can create a single model that is shared between them without disclosing sensitive information. The motivation for this research is to aid in the development of distributed statistical learning algorithms by creating a discrete event simulation system that supports the evaluation of new under-research algorithms, assessing their performance in distributed environments while considering the adversities those environments bring, such as delays and faults.

1.3 Problem

Machine learning models are composed of many mathematical functions that need to be replicated along the network to be able to achieve the same results as when run locally on a machine, being aware that data should not be available to other machines. Algorithms that replicate these mathematical functions are being developed and, as such, need to be evaluated to weigh the advantages and disadvantages of their use, as well as to optimise them.

1.4 Objectives

Throughout this work, we seek to understand how statistical distributed algorithms work, how to design a discrete event simulation suitable for the purpose, research if there are already simulators that can be adapted or if a new one needs to be programmed, and lastly, evaluate the statistical distributed algorithms. Our goal is to answer the following questions:

1. How can the state-of-the-art discrete-event simulation methodologies be used to evaluate distributed algorithms efficiently?
2. How can distributed statistical learning be specified and modelled for discrete-event simulation?
3. How is a discrete-event simulator for the purpose of evaluating distributed statistical learning algorithms designed?
4. How can we characterise and evaluate a distributed algorithm?

1.5 Document Structure

In the present chapter, Chapter 1, the problem is contextualised, the motivation presented, the problem exposed and the objectives explained.

Chapter 2 presents the State of the Art regarding discrete-event simulation.

Chapter 3 explains which characteristics will be looked for in a discrete-event simulator to evaluate distributed learning algorithms, explores existing simulators and documents how and if they fulfil those characteristics, ultimately comparing them.

Chapter 4 describes how a discrete-event simulator purpose-built for evaluating distributed statistical learning algorithms can be designed and describes its implementation.

Chapter 5 validates the developed solution, presenting how it can be used to evaluate and improve the target algorithms.

The final chapter, Chapter 6, summarises the work developed, highlighting the main contributions, answering the research questions and identifying possible future work.

Chapter 2

State of the Art

In this chapter, the concepts and advancements made in the field of discrete-event simulation will be presented. Simulation will be the methodology used to simulate and evaluate distributed statistical learning algorithms. Section 2.1 will target simulation in general, while Section 2.2 will target discrete-event simulation in specific. Section 2.3 will explain how physical systems can be clarified for discrete-event simulation. Section 2.4 will introduce parallel discrete-event simulation and existing synchronisation techniques, and Section 2.5 will cover distributed discrete-event simulation. Section 2.6 will target the performance of discrete-event simulators, leading to the introduction of GPU-based simulators in Section 2.7 and then Section 2.8 will introduce some of the existing simulators. Finally, Section 2.9 will conclude the topic.

2.1 Simulation

Simulation is the process of replicating a real-world entity, situation, or process over time, representing the main characteristics and behaviours of physical or abstract systems. It is widely used in and outside academia due to its ability to obtain results at a much smaller associated cost. A physical system can be imitated without needing to be physically recreated, aiding in decision-making and what-if analysis. Due to this advantage, it is used by many industries and for many use cases. For example, it is used in video games, logistics, manufacturing, and many others. Some companies specialise in simulation. It can also be used for optimisation, as a system can be tweaked to obtain performance results.

2.2 Discrete-Event Simulation

Event-oriented simulation attempts to describe a system as a series of moments to simulate it, modelling a system as a sequence of events. Discrete-event simulation is an event-oriented simulation that models a system as a discrete sequence of events, processing the system's functioning as

consecutive timestamps [27]. Discrete-event simulation can be characterised by three main parts: the state, the events, and the clock. The state is a collection of variables that can describe the system at any time. Events are occurrences that change the system's state and are associated with a timestamp. Between two consecutive events, the system's state cannot be changed [27]. The clock keeps track of the simulation's time. Put into simple terms, the simulation works by advancing the clock to the timestamp of the earliest unprocessed event and then processing it. The event then changes the system's state, and the simulation proceeds to repeat the same process until it is either stopped or no more unprocessed events exist. Another concept associated with event-oriented simulation is that of activities, they differ from events because they transform the system's state over time, however, for discrete-event simulation, they can be abstracted as two events, the start and the end of the activity [34].

2.3 Discrete-Event System Specification (DEVS)

The specification of a system can be done through the Discrete Event System Specification (DEVS) formalism [34], used for modelling and analysis of discrete event systems. Suppose a system, over time, accepts events as inputs and produces events as outputs. In that case, it can have its behaviour and structure described by a DEVS [34]. The results of that specification are the system's states, inputs, outputs and state transitions.

Classic DEVS is described in the following structure (according to [35]) :

$$DEVS = (X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta)$$

where:

- X is the set of input events;
- Y is the set of output events;
- S is the set of sequential states;
- $\delta_{ext}: Q \times X \rightarrow S$ is the external state transition function;
- $\delta_{int}: S \rightarrow S$ is the internal state transition function;
- $\lambda: S \rightarrow Y$ is the output function;
- $ta: S \rightarrow \mathbb{R}_0^+ \cup \infty$ is the time advance function;
- $Q = (s, e) | s \in S, 0 \leq e \leq ta(s)$, the set of total states

State transitions are divided into internal and external transitions. Internal transitions are triggered by internal events (events from the outputs set Y), while external transitions are triggered by external input (events from the inputs set X).

For example, consider a conversation between two persons where one is either talking and the other listening or the other way around, the two tasks being mutually exclusive. Describing the system as a person, its inputs would be $\{listen\}$, and its outputs would be $\{talk\}$, it would have the two states $\{Listening, Talking\}$ and start with the state *Talking*. Once the event *listen* occurs, as it is an external input (i.e. not an event created by this system), an external transition is triggered,

and the output event *talking* is added to the set S , changing its state from *Talking* to *Listening*. In the physical system, this would be when the person stops responding and starts listening to what the other person says. Once the event *talk* occurs, as it is an internal input (i.e. created by the very same system as an output event), an internal transition is triggered, and the event *listen* is added to the set S , changing the state back from *Listening* to *Talking*. In the physical system, this would be when the person responds to what he just listened to. The specification is applied in multiple academic papers such as Mayrhofer et al. [20] and development environments such as DEVSimPy [5].

Zeigler introduced Classic DEVS in 1976 [33], with its specification being limited to the idea of processing a single event at a time, meaning that there could be only one external input, later, Parallel DEVS [8] was introduced to overcome this limitation, adapting the formalism so that external transitions take into consideration more than a single external input.

Parallel DEVS is described in the following structure (according to [35]) :

$$DEVS = (X, Y, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta)$$

where:

X is the set of input events;.

X is the set of input events;

Y is the set of output events;

S is the set of sequential states;

$\delta_{ext}: Q \times X^b \rightarrow S$ is the external state transition function, where X^b is a set of bags over elements in X ,

$\delta_{int}: S \rightarrow S$ is the internal state transition function;

$\delta_{con}: S \times X^b \rightarrow S$ is the confluent transition function, subject to $\delta_{con}(s, \phi) = \delta_{int}(s)$

$\lambda: S \rightarrow Y^b$ is the output function;

$ta: S \rightarrow \mathbb{R}_0^+ \cup \infty$ is the time advance function, where $Q = \{(s, e) | s \in S, 0 < e < ta(s)\}$, and e is the elapsed time since the last state transition.

2.4 Parallel Discrete-Event Simulation

A simulator tasked with simulating a complex or resource-intensive physical system is limited by its processing power. Therefore, breaking the system into multiple logical processes can speed up the simulation by processing a higher amount of events in the same time interval. A simulation that divides itself into multiple logical processes is distributed or parallel [34]. While having the system divided into logical processes allows for more scalability, causality between logical processes needs to be enforced. Causality is important because events need to be processed in chronological order. Therefore, a logical process's clock cannot be too far ahead as other logical processes might send an event to a process whose timestamp is before the current logical process's clock [27]. To avoid this problem, time-synchronisation between logical processes is introduced.

The algorithms that enforce time-synchronisation between logical processes can be divided into four categories: optimistic, conservative, relaxed, and combined [24].

2.4.1 Optimistic Synchronisation

Optimistic time-synchronisation algorithms allow logical processes to process events without being certain that they will persist in order. If a logical process is notified that it processed an out-of-order event, it must undo or roll back the local state to the state before becoming out of order. The Time Warp [14] mechanism is the most commonly used one [27]. It can roll back to previous states as it stores every state in memory. To use as little memory as possible, optimistic time-synchronisation algorithms use a global clock called Global Virtual Time (GVT) to track the lowest clock value of every logical process. If a state is prior to the global clock, then that state will not be rolled back and, therefore, need not be stored.

2.4.2 Conservative Synchronisation

In conservative time-synchronisation algorithms, logical processes only process an event if guaranteed that its timestamp is prior to any event it may receive. These algorithms are susceptible to situations where every logical process is waiting for an update from another logical process, resulting in none of them being able to process the next event. These situations are called *deadlocks* [27], and they are the reason for the guarantee that a logical process will only process safe events. A safe event is an unprocessed event whose logical process cannot receive any event before it. If a logical process can receive an event prior to the next unprocessed event, then the unprocessed event is not a safe event. The guarantee can be obtained by having logical processes collaborate. If every logical process communicates with all the others, then it is a *synchronous* algorithm, else if each logical process communicates only with a smaller subset, then it is an *asynchronous* algorithm. [27]

2.4.2.1 Asynchronous Conservative Synchronisation

An example of a popular asynchronous conservative time-synchronisation algorithm is the Chandy-Misra-Bryant algorithm, developed independently by Chandy and Misra (1979) [7] and Bryant (1979) [4], it is an asynchronous conservative time-synchronisation algorithm that synchronises time between logical processes and prevents *deadlocks* through the sending of *null messages*, these messages contain the timestamp of the sender's earliest unprocessed event and the *lookahead* value. The *lookahead* value is the smallest possible time interval between the occurrence of an event at the sender's logical process and the event created by that occurrence in the receiver's logical process. Therefore, this value depends on the system being modelled. For example, if the system being modelled was a conversation, the *lookahead* could be the time that sound waves take to travel from one person to another. In order to reduce the number of messages between logical processes in the Chandy-Misra-Bryant algorithm, when a logical process cannot process

any safe event, it requests *null messages* to the other logical processes [22]. The algorithm, however, does not prevent *deadlocks* if the system contains a cycle among logical processes with zero *lookahead* [11].

2.4.2.2 Synchronous Conservative Synchronisation

Synchronous conservative time-synchronisation algorithms determine if an event is safe, similar to how GVT is calculated in optimistic time-synchronisation algorithms (2.4.1). The logical processes find the lowest bound timestamp (LBTS) from their unprocessed events, being able to take into consideration the *lookahead* and the existence of transient messages [27], transient messages need to be considered because after the sender processes an event and before the receiver receives it, the event is not accounted for, that resulting event could, when finally delivered, have a prior timestamp to the LBTS calculated. The LBTS can be calculated in different ways depending on the algorithm. In fact, Mattern (1993) [19] considers the determination of a tight lower bound on the GVT value to be important, defining the GVT as the minimum of all local locks and of the timestamps of transient event messages.

2.4.2.3 Lookahead

Conservative time-synchronisation algorithms rely on the lookahead value to operate. Swenson (2015) [27] says that a parallel simulation environment that operates with minimal to no *lookahead* is comparable to a non-parallel simulation with the additional overhead of logical process communication. The less frequently a logical process requires synchronisation, the more performant the simulator can be because it spends more time processing internal events [27]. Therefore, maximising the *lookahead* is an important objective of parallel discrete-event simulation. Techniques have been developed to maximise *lookahead* by calculating it dynamically, and they can achieve higher *lookahead* values than those stipulated statically. Even if their difference is insignificant, their impact on the simulation is [24]. For examples of dynamic *lookahead* extraction algorithms, see Liu and Nicol (2002) [16] and Zhou et al. (2004) [36].

2.4.3 Relaxed Synchronisation

Relaxed time-synchronisation algorithms permit for logical processes to occasionally process events without being constrained by their timestamp order, they can do it out of order if the unprocessed events' timestamps have a small difference between each other. Not being constrained by the timestamp order can simplify synchronisation by not relying as much on the *lookahead* value. However, guaranteeing that processing events out of order does not impact the validity of those events remains important and is a significant challenge [24]. According to Perumalla (2006) [24], repeatability is another issue that might come with relaxed synchronisation. Repeatability is the ability for simulations to execute multiple times with the same inputs, guaranteeing the same results every time. As events can be processed out of order, special care should be taken to preserve this property.

2.4.4 Combined Synchronisation

Combined time-synchronisation algorithms combine the characteristics of the three mentioned above, for example, using optimistic synchronisation when the *lookahead* value is difficult to obtain, otherwise behaving as a conservative time-synchronisation algorithm. This time-synchronisation approach can be seen in a practical application, namely, the US Department of Defense (DoD) High Level Architecture [9].

2.5 Distributed Discrete-Event Simulation

Parallel discrete-event simulation can have its logical processes divided into multiple machines, resulting in distributed discrete-event simulation. This type of simulation is helpful in simulating complex systems because a single machine's physical capabilities do not constrain it. As the number of machines involved in the simulation need not be fixed, more processing power can be added by introducing new machines to the simulation environment. Therefore, distributed discrete-event simulation is scalable and potentially more efficient [32].

2.6 Simulation Performance

Although having more machines can translate to more processing power, in distributed discrete-event simulation, due to synchronisation, communication delay must be taken into consideration, along with how messages are transmitted. Logical processes in different machines need to communicate with each other through sockets, which is less effective than process or memory communication [32].

For communication between processes on a single machine, the Message Passing Interface (MPI) is generally used [27]. As processes use virtual address spaces and pointers do not translate to other processes unless they have shared memory, techniques to speed up communication between logical processes through the MPI have been introduced. The zero-copy message-passing technique [27], which, instead of serialising and deserialising data into shared memory, uses smart pointers that are sent to receiver logical processes, as such, this technique eliminates the processing cost of copying large messages into memory by having the receiver access the memory directly. Consequently, zero-copy message passing also leads to smaller amounts of data being sent [27]. Swenson (2015) [27] proposes a new approach to zero-copy message passing that improves performance even more.

In pursuit of more simulation performance, various uses of hardware were proposed to mitigate some of the challenges posed by discrete-event simulation. To increase the Time Warp [14] mechanism's performance in optimistic time-synchronisation algorithms, the *Rollback Chip* was proposed by Fujimoto, Tsai, and Gopalakrishnan (1992) [12], reducing state saving and rollback overhead and improving the simulator's performance. Another chip was proposed by Lynch and

Riley (2009) [17] for conservative time-synchronisation algorithms in order to calculate the logical processes' lowest bound timestamp, speeding up global synchronisation. This chip resulted in half the runtime in a simulation carried out using the GTNetS [26] simulator.

2.7 GPU-Based Discrete-Event Simulation

The problem with specialised hardware for simulations is that it is not widely available. However, a hardware component that can be more easily found in the machine of whoever is conducting a simulation is the General Purpose Graphics Processing Unit (GPGPU). GPGPUs have been used in simulation, for example, in Cunetsim, a CPU-GPU co-simulation framework developed using the Compute-Unified Device Architecture (CUDA), according to Bilel, Navid, and Bouksiaa (2012) [3], "(...) is able to achieve up to 260 times faster execution time than existing simulators, when targeting large scale mobile networks.", adding that current simulators could benefit from GPU parallelism, an idea that was also concluded by Swenson (2015) [27]. The downside of GPU-based simulation is that GPUs require parallelisation to be efficient, therefore, how the system being simulated is modelled will have a heavy influence on whether or not, GPU-based simulation or CPU-based is the most efficient. Bilel, Navid, and Bouksiaa (2012) [3] report that Cunetsim can be outperformed by CPU-based simulators on small and medium-scale networks due to the need to transfer data between the computer's memory and the GPU's memory and due to the GPU being underused, as only a few cores are active. As the performance of a simulator depends on the simulation model created by a user, the "follow-the-leader" approach was created to dynamically select between the CPU and the GPU at runtime, being able to reduce the runtime value when compared to having only one of them selected [18].

2.8 Existing Simulators

There are multiple network simulators available, and some of them have already been mentioned. The choice of simulator must be considered due to the advantages and disadvantages mentioned before. On the subject of network simulation, multiple of the existing options simulate the multiple layers of the Open Systems Interconnection (OSI) model. However, the purpose of this paper is to evaluate distributed statistical learning, and therefore, the layers below the application layer can be abstracted and need not be simulated. This requirement rules out simulators such as ns-2, ns-3, GTNetS [26], and GloMoSim. Suitable simulators for this purpose would be simulators that implement the DEVS formalism, such as DEVSimPy [5], or network simulators focused on the study of distributed algorithms, such as SimGrid [6] and Sinalgo [13].

2.9 Conclusions

Simulation is used extensively in research to assess results due to the possible high cost associated with testing theory in a practical environment. It provides a cost-effective solution to decision-

making and what-if analysis. Physical systems can be described using the DEVS formalism, allowing for a transparent and standardised specification that is then adapted to a simulation environment. With the need to simulate more complex systems, parallel and distributed discrete-event simulations were introduced, increasing performance, scalability, and adaptability to resource-intensive scenarios. These were complemented by introducing multiple techniques that attempted to mitigate the performance lost with logical process synchronisation. Moreover, GPU-based simulation was also introduced, scoring better results than CPU-based simulation the more complex the modelling scenario is. However, to optimise resource utilisation, the choice of simulation tools and techniques depends on the system's scale, requirements, and attributes, emphasising the importance of choosing a simulator design that aligns with the specific needs of the simulated physical system.

Chapter 3

Simulators

This chapter aims to understand which simulator fulfils the objectives of this thesis, concluding which one to use or whether a new one should be created.

Section 3.1 describes which characteristics are relevant to evaluating statistical learning algorithms and, as such, are to be considered. After defining the characteristics, three simulators will have their functionalities described based on the traits pursued. These three simulators are Sinalgo [13] (Section 3.2), QUANTAS [23] (Section 3.3), and SimGrid [6] (Section 3.4). They were chosen because, as already mentioned, they focus on the study of distributed algorithms, abstracting the layers of the OSI model below the application layer. Finally, Section 3.5 compares the simulators based on the characteristics and concludes the chapter.

3.1 Simulator Characterisation

In order to characterise the simulators in question, the Flow Updating [1] algorithms were implemented in each one of them, however, due to how these simulators process events, not every variation of the algorithms was implemented¹²³. In order to compare them, and assess the necessity of a simulator that fulfills all the requirements, the requirements were listed and the characteristics of the simulators were collected. The requirements are the following:

- Environment Simulation - The simulator must be able to replicate a distributed network.
- Metric Visualisation - Must provide logging or other forms of debugging and/or metric collection.
- Scenario Configuration - Must allow the configuration of custom scenarios (such as different network topologies).

¹<https://github.com/AvilaAndre/simgrid-flow-updating-implementation>

²<https://github.com/AvilaAndre/sinalgo-flow-updating-implementation>

³<https://github.com/AvilaAndre/quantas-flow-updating-implementation>

- Fault Injection - Must either have or allow the introduction of errors, in order to simulate system and network failures.
- Reproducibility - The simulation must be deterministic, being able to replicate the results.

3.1.1 Environment Simulation

The objective of the simulator is to allow the observation of the behaviour of an algorithm in a distributed environment. To observe the behaviour as close to reality as possible, it is crucial to simulate the environment that interacts with and influences that behaviour.

Simulating the network is essential because it is the primary component of a distributed environment. Routing and latency are two important aspects of that component that influence how actors in the network communicate with each other.

Some simulators attempt to reproduce the environment as close to reality as possible at the expense of performance. They achieve high fidelity by recreating low-level network behaviour, such as routing individual packets throughout the network infrastructure.

However, recreating a network is computationally expensive. Some aspects of the network, such as routing and latency, can be abstracted using numerical values and probabilistic distributions. For instance, the simulator can use a predefined value corresponding to the latency instead of figuring out the route between two actors to determine the delay between them.

The simulator and the objectives of the simulation determine the values used for abstracting. The values abstracting characteristics such as latency can be manually configured by the researcher, calculated from spatial coordinates, set to a constant value, or even randomly generated from a probabilistic distribution.

These abstractions allow the simulator to handle more complex scenarios and operate more efficiently while still capturing the essential characteristics of the distributed environment.

Regarding the processing of discrete events, the simulators may support synchronous simulation, asynchronous simulation, or both. The difference comes down to how events are processed. Synchronous and asynchronous simulation is not the same concept as synchronous and asynchronous conservative synchronisation.

When a simulation is synchronous, it advances through rounds, with the peers performing their calculations every round. However, it is still a discrete event simulation because events are processed within their respective round.

When a simulation is asynchronous, it advances through a sequence of discrete events, dealing with the earliest event before moving to the next one.

Some distributed algorithms function through rounds. For example, peers may only broadcast a new message after receiving one from every other peer. This behaviour can be modelled for an asynchronous simulation or abstracted as a synchronous simulation. However, even though synchronous simulation can be modelled as an asynchronous simulation, the opposite is not always possible. Therefore, the simulators should be able to support at least asynchronous simulation.

3.1.2 Metric Visualisation

The behaviour of an algorithm in the simulator must be observable to allow the researcher to make conclusions.

To make the behaviour of the algorithms observable, the simulator could provide logs. If, once they are processed, the events are logged, a researcher can easily understand the order in which the events took effect and assess if the algorithm is working as expected.

Logging the time events are processed is useful but may not be sufficient. A researcher might also need to inspect the value of specific variables. This issue can be addressed by allowing the researcher to introduce custom log entries into the simulator.

For example, imagine a scenario where a researcher wants to study a distributed algorithm that counts the number of peers in a network. The peers have a set of identifiers of the peers they know. If the researcher only wanted to find out the moment when a peer receives an identifier and the number of peers that he is aware of at the time, he would benefit from being able to introduce a custom log message. In this case, the custom log would only show the simulation time in which a peer received a new identifier and the size of its set of known peers, allowing the researcher to visualise how fast the peers become aware of each other and at what rhythm.

Another visualisation method is the use of a graphical user interface (GUI). The GUI introduces a more intuitive way to observe and follow the timeline of events and their impact on the system. However, a UI may not be the most adequate tool to extract metrics compared to logs.

3.1.3 Scenario Configuration

To assess the behaviour of an algorithm under varying conditions, the simulator should allow scenario configuration. This enables researchers to run new simulations quickly without modifying the simulator's implementation.

An important aspect that should be configurable is the topology of the distributed system. By simulating an algorithm across multiple topologies, the researchers can evaluate and compare the behaviour of the algorithm in different network structures.

The settings which can be configured are dependent on the simulator. However, some settings are important, such as the number of peers simulated and their connections, which must be present.

While the available configuration options depend on the simulator, specific parameters, such as the number of simulated peers and how they are connected, are fundamental and should be supported.

3.1.4 Fault Injection

Much like fault injection in software testing, which intends to assess a program or system's robustness, resilience and fault tolerance, a simulator should allow the researcher to introduce faults into its simulated environment. This capacity facilitates the study of the algorithm's behaviour upon encountering adverse conditions.

Faults can be introduced either automatically, through stochastic methods, or manually, by allowing the researcher to choose when, how and for how long those faults occur. The manual introduction of faults tests the algorithms in controlled scenarios, where the researcher is aware of the fault's presence, resulting in a more precise evaluation of the reactive behaviour. In contrast, automatic fault injection tests the algorithms in unpredictable conditions, which may reveal unexpected behaviours and edge cases.

Simulating faults enables the assessment of an algorithm's fault tolerance and exposes flaws while the algorithm is still under study. Errors identified through simulation can lead to better error handling, while discovered vulnerabilities and weaknesses can enhance resilience. Exposing the algorithm to previously untested scenarios also contributes to broader test coverage and a deeper understanding of the algorithm's behaviour under stress.

3.1.5 Reproducibility

Reproducibility reinforces all previously discussed requirements by allowing the researcher and others to verify the results obtained through simulation. For a simulation to be reproducible, its configuration and execution conditions must be replicated precisely. However, this alone is insufficient, as reproducibility is only possible if the simulator behaves deterministically.

Determinism can be compromised by various factors, such as exposure to external events and improperly handled multithreading. External events refer to anything outside the simulator's control, such as user input during execution or accessing the system's clock instead of the simulated time source.

While multithreading typically introduces non-determinism because of unpredictable thread scheduling, a multithreaded simulator can be deterministic if it can control the thread execution, avoid race conditions and ensure all sources of randomness are repeatable.

Repeatable randomness is essential for the simulator to be deterministic and can be achieved using a fixed random seed. The researcher could provide this seed in the simulation's configuration, ensuring that the same sequence of pseudo-random values is generated on each run, resulting in observable consistent behaviour.

3.2 Sinalgo

Sinalgo (Simulator for Network Algorithms) [13] is a framework for simulating and evaluating distributed algorithms. It is programmed in *Java* and is, at the time of writing, in version 1.0.0-beta6, which was last updated on the 13th of June 2019. Sinalgo is a very extensible simulator that allows the customisation of many of its components.

3.2.1 Environment Simulation

Sinalgo replicates a distributed environment by representing it as a graph, where the nodes are peers, and the edges are the connections. Nodes have a position associated with them, and, by

default, there is no notion of routing. If a peer can send a message to another peer, then, in the graph abstraction, the node corresponding to the first peer should have an edge to the node representing the second peer.

The simulator decides whether the peers should be connected through the connectivity model. This model can be implemented by extending the class `ConnectivityModelHelper` and overriding the function `isConnected(Node from, Node to)`, which returns a boolean and decides whether an edge between those two nodes should exist. Listing 3.1 shows an example. In synchronous mode, the connectivity model updates every round.

```

1 public class MaxDistanceConnectivity extends ConnectivityModelHelper {
2
3     static double maxDistance = 800.0;
4
5     @Override
6     protected boolean isConnected(Node from, Node to) {
7         double dist = from.getPosition().distanceTo(to.getPosition());
8         return dist < maxDistance;
9     }
10
11 }

```

Listing 3.1: Example of a connectivity model implementation

Updating the connectivity model can be important if a mobility model is used. The mobility model changes the positions of the nodes over time. Therefore, if a connectivity model such as the example in Listing 3.1 decides if two nodes are connected based on their position, then two nodes could become or stop being connected after a position change.

The mobility model can also be customised, by extending the `MobilityModel` class and overriding the function `getNextPos(Node n)` which returns the position which the node will be set.

The researcher can implement the amount of latency between nodes through a message transmission model, one of the many extensible components offered by the simulator. This model is obtained by extending the abstract class `MessageTransmissionModel` and overriding the method `timeToReach`, which returns a double with the delay value. By default, Sinalgo includes two transmission model classes, `ConstantTime`, which returns 1.0 and `RandomTime`, which returns a random value from a distribution (the configuration file must indicate the distribution type). A custom implementation of a message transmission model called `DistanceTime` is shown in Listing 3.2, setting the delay between nodes as their distance in the graph.

```

1 public class DistanceTime extends MessageTransmissionModel {
2
3     @Override
4     public double timeToReach(Node startNode, Node endNode, Message _msg) {
5         return startNode.getPosition().distanceTo(endNode.getPosition());
6     }
7 }

```

```

6     }
7
8 }
```

Listing 3.2: Example of a message transmission model implementation

This simulator carries out the simulation sequentially (in a single thread) and can process events synchronously or asynchronously.

3.2.2 Metric Visualisation

This requirement is one of Sinalgo's strong points. The simulator offers logging and a GUI.

3.2.2.1 Logging

Starting with the logging capabilities, Sinalgo includes a class called `Logging`, which implements the methods necessary to be able to create logs. This class allows for creating log files by providing a file name (as seen in line 2 of Listing 3.3). Consequently, the researcher can create multiple log files for different metrics. The configuration file for a simulation can have an attribute called `logFileName`. The default log file uses this attribute's value as its file name. Therefore, it is not mandatory to provide a file name when calling `Logging.getLogger` (as seen in line 1 of Listing 3.3) because, by default, it will retrieve the file name from the configuration, throwing an error if the name is not present. The default log file can also be accessed through `sinalgo.runtime.Global.log`.

```

1 Logging defaultLog = Logging.getLogger();
2 Logging newLog = Logging.getLogger("unexisting_example_log.txt");
3
4 defaultLog.log("This is a new log!");
5 defaultLog.println(); // Appends a line break to the log file.
6 newLog.println("This log appends a line break to the given string");
7
8 boolean shouldLog = true;
9 newLog.println(shouldLog, "This log is added only if shouldLog is true");
```

Listing 3.3: Example of a log file creation

The method `Logging.getLogger()` implements the singleton design pattern, and therefore, subsequent calls with the same argument will return the same instance. As a consequence, that instance can be obtained anywhere in the simulator.

To add a log into a log file during simulation, the researcher has to call one of the many functions available in the `Logging` class (as seen from line 4 to line 9 of Listing 3.3).

Additionally, the configuration file includes the attributes explained below:

- `outputToConsole`: takes a boolean as value and, when set to true, does not create the default log file but instead writes its logs to the console.

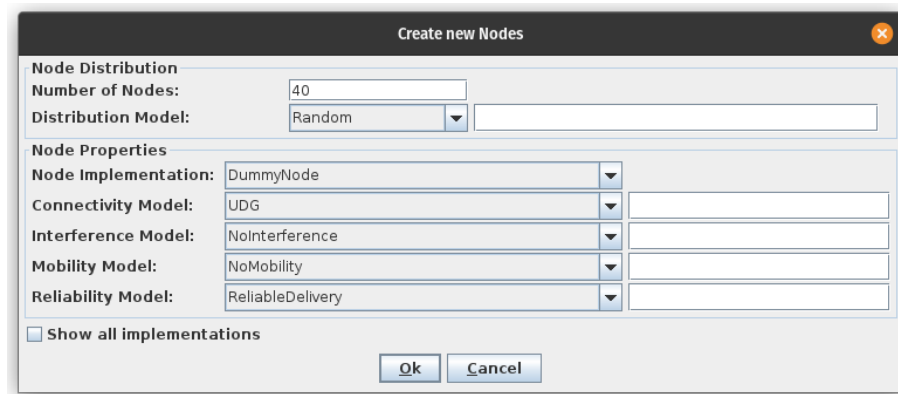


Figure 3.1: Pop-up interface used to create nodes

- `logToTimeDirectory`: takes a boolean as value and, when set to true, creates a new directory named after the string representation of the date when the simulation started.
- `logConfiguration`: takes a boolean as value and, when set to true, writes the current configuration to the default log file after the simulation starts.
- `eagerFlush`: takes a boolean as value and, when set to true, flushes the log files upon adding a new log, trading performance for the guarantee that the logs saved to the file.

On Linux systems, log files would be located at:

```
$HOME/.sinalgo/logs/
```

3.2.2.2 Graphical User Interface

Moving on to the graphical user interface. Sinalgo provides a visualisation method that uses its graph-like peer structure to draw the nodes on the screen. The node's position on the screen corresponds to its position in the simulation.

Using this interface, nodes can be manually created by selecting the distribution and their properties in the pop-up presented in Figure 3.1

It is possible to create a specific amount of nodes and position them all using a distribution model. By default, Sinalgo provides five distribution models, those being:

- **Circle**: Arranges the nodes in a circular formation.
- **Grid2D**: Arranges the nodes in a grid-like formation.
- **Line2D**: Arranges the nodes in a line formation.
- **Position File**: Positions the nodes according to the positions defined in a file, requiring the file name.
- **Random**: Positions the nodes randomly (its implementation can be seen in Listing 3.4).

New distribution models can be created by extending the `DistributionModel` class as can be seen in Listing 3.4.

```

1  /**
2   * A distribution model that randomly distributes the nodes in the entire
3   * simulation area.
4   */
5  public class Random extends DistributionModel {
6
7      // The random-number generator
8      @Getter(AccessLevel.PRIVATE)
9      @Setter(AccessLevel.PRIVATE)
10     private java.util.Random rand = Distribution.getRandom();
11
12     @Override
13     public Position getNextPosition() {
14         return Tools.getRandomPosition(this.getRand());
15     }
16
17 }

```

Listing 3.4: The provided implementation of the Random distribution model

The properties of the nodes that can be selected, as seen in Figure 3.1, are:

- **Node Implementation:** Implements the node's functionality. This is where the algorithm being tested should be implemented.
- **Connectivity Model:** Defines when two nodes are in communication range. This model is explained in Section 3.2.1, and an example can be seen in Listing 3.1.
- **Mobility Model:** Defines how the position of the nodes changes over time. This model is explained in Section 3.2.1.
- **Interference Model:** Defines whether messages can interfere with each other. This model is explained in Section 3.2.4.
- **Reliability Model:** Defines whether a message reaches its destination. This model is explained in Section 3.2.4.

After creating the nodes, their connections can be generated using the button `Reevaluate Connections`, which will use the chosen connectivity model.

Sinalgo's graphical interface also allows the researcher to move around nodes by pressing their graphical representation with the mouse's right click and dragging them to the desired destination. New nodes can be created by double-tapping a desired position, and edges can be created by pressing a node and dragging a line to the next node. While creating an edge, a line representing the edge starting from the selected node appears, disappearing if the edge creation is unsuccessful.

The nodes' graphical representation can be customised while running the simulation, which means that, for example, if a simulation had multiple nodes of the same type and each node had a number, the number could be drawn in the graphical representation of the node, additionally, if the simulation had the purpose of maintaining the value above 0, the researcher could customise its appearance to be coloured red when the value falls below 0. Hovering over a node also shows information that can be customised (in the previous example, it could show the minimum value the node has ever had during the simulation).

Regarding the control of the simulation, there is a start and stop button. The start button has the option to run forever (until the stop button is pressed or, if the simulation is asynchronous, there are no events to be processed) and an option to, if the simulation is asynchronous, run only a desired amount of events or if the simulation is synchronous, a desired amount of rounds. If the option to run only a desired amount of rounds or events is pressed, a field becomes editable, which takes the desired amount. Every time the run button is pressed, the simulator will run the given amount (stopping earlier if the simulation is asynchronous and there are no more events to be processed). Additionally, the simulator provides a field with the refresh rate value. What this value does is that it sets the number of rounds or events that the simulator processes before updating the graphical representation of the simulation. As updating the graphical representation can be computationally costly, this option benefits simulations with many graphical elements (nodes and edges).

Sinalgo's interface also includes project control, which can show custom buttons that can be pressed while the simulation is paused. The custom buttons allow for the introduction of behaviours into the simulation and the provision of insights. For example, in a simulation where every node has a randomly generated number assigned to it, the researcher can create a button which informs him of the average value obtained by summing the values of the nodes. Listing 3.5 shows an example of the described button functionality.

```

1 public class CustomGlobal extends AbstractCustomGlobal {
2
3     @AbstractCustomGlobal.CustomButton(buttonText = "AVERAGE", toolTipText = "Check the
        average value from the nodes.")
4     public void sampleButton() {
5
6         double valuesTotal = 0.0;
7         int nValues = 0;
8
9         for (Node node : SinalgoRuntime.getNodes()) {
10             if (node instanceof ExampleNode) {
11                 ExampleNode exampleNode = (ExampleNode) node;
12
13                 valuesTotal += exampleNode.getValue();
14                 nValues++;
15             }
16         }
17
18         double valuesAvg = nValues > 0
19             ? valuesTotal / nValues
20             : 0.0;

```

```
21
22     String formattedAvg = new DecimalFormat("###.##").format(valuesAvg);
23     String output = "The average value is " + formattedAvg + ".";
24
25     JOptionPane.showMessageDialog(null, output);
26 }
27
28 ...
29 }
```

Listing 3.5: Example of a GUI button

3.2.3 Scenario Configuration

As already seen, Sinalgo is extensible and, as such, contains a vast number of options and configurations. Each Sinalgo project has a configuration file, which has to be created in the simulator's source files.

The configuration files are in the XML format. However, the simulator allows for the customisation of the framework-specific configurations through the GUI, which has a button that saves the modified configuration file. Project-specific configurations do not have specific user interface elements for each attribute and, therefore, must be typed out.

Although the configuration of a project is initially in the simulator's source files, when the configuration is modified using the GUI, the configuration is stored in another configuration file. On Linux systems, for a project named `example`, the modified configuration file would be located at:

```
$HOME/.sinalgo/projects/example/Config.xml
```

The project-specific configurations can be accessed anywhere in the simulator. Therefore, it allows a researcher to add specific values and configurations to the already customised components.

The configuration file must include two important settings: `simulation area` and `random number generators`. The simulation area informs the simulator if positions have two or three dimensions and what the limits of those dimensions are. The setting about `random number generators` includes options regarding the seed used.

Sinalgo contains many more settings that can be tweaked. The information about each setting in the configuration file can be found in the example configuration file that the simulator includes.

3.2.4 Fault Injection

Natively, Sinalgo supports fault injection, and the already-mentioned interference and reliability models create message-related problems. However, Sinalgo does not implement any mechanism related to the peer's normal functioning being affected (such as crashes).

Regarding message-level faults, the available models allow the researcher to customise the faults that affect the system's message communication.

The interference model defines whether messages can interfere with each other. It can have a custom implementation by extending the class `InterferenceModel` and overriding the method `isDisturbed(Packet p)`, which returns a boolean. Suppose the return value is `false`. In that case, the simulator will change the packet's (a packet is the data object which contains information about the message, including the message) `positiveDelivery` value to `false`, which means that the message will not reach the node it was supposed to.

The interference model is used in both synchronous and asynchronous simulations. In synchronous simulation, interference is calculated once per message per round. In asynchronous simulation, the amount of times that interference is calculated depends on the configuration flag `interferenceIsAdditive`.

When the flag is `false`, interference is recalculated every time a message starts or stops being sent (packet addition and removal). Non-additive interference is slower but precise.

When the flag is `true`, interference only recalculates on packet removal and only if new packets were added before the removal. Additive interference reduces the number of interference checks, making them faster but less precise.

An example of an interference model implementation can be seen in Listing 3.6

```
1 public class ExampleInterference extends InterferenceModel {  
2  
3     private Random rand = Distribution.getRandom();  
4  
5     @Override  
6     public boolean isDisturbed(Packet p) {  
7         double r = this.getRand().nextDouble();  
8         return (r > 0.5);  
9     }  
10  
11 }
```

Listing 3.6: Example of an interference model implementation

The reliability model defines whether a message reaches its destination. It can have a custom implementation by extending the class `ReliabilityModel` and overriding the method `reachesDestination(Packet p)`, which returns a boolean. If the return value is `false`, then the simulator will change the packet's `positiveDelivery` value to `false`. The reliability model is used in both synchronous and asynchronous simulation. When a packet is created, reliability is calculated, changing the value of the packet's `positiveDelivery` to the return value of the reliability `reachesDestination(Packet p)` function. An example of a reliability model implementation can be seen in Listing 3.7

```
1 public class ExampleLossyDelivery extends ReliabilityModel {
```

```

2
3     private Random rand = Distribution.getRandom();
4     private double dropRate = 0.1;
5
6     @Override
7     public boolean reachesDestination(Packet p) {
8         double r = this.getRand().nextDouble();
9         return (r > this.getDropRate());
10    }
11
12 }

```

Listing 3.7: Example of a reliability model implementation

Regarding peer-level faults, Sinalgo has no purpose-built features, meaning that the researcher must attempt to implement them in the project from scratch. One of Sinalgo's features, which has not yet been mentioned but might be helpful for this use case, is the existence of timers. A timer is a customisable object that performs a programmed action after a given delay. When starting a timer, it can be started relative to the simulation clock with `startRelative(double relativeTime, Node n)` or indicating when, in the simulation time, it should be fired with `startAbsolute(double absoluteTime, Node n)`. The second parameter the two functions receive is the node bound to the timer. However, the value can be `null` if the timer should execute globally. A timer can be implemented by extending the class `Timer` and overriding the function `fire()`. An example of a timer implementation can be seen in Listing 3.8

```

1 public class ExampleTimer extends Timer {
2
3     public ExampleTimer(double delay, ExampleNode node) {
4
5         // This line is usually called outside the timer
6         this.startRelative(delay, node);
7     }
8
9     @Override
10    public void fire() {
11        System.out.println("This is called after the delay!");
12        // targetNode is the node which started the timer
13        this.getTargetNode().doSomething();
14    }
15 }

```

Listing 3.8: Example of a timer implementation

3.2.5 Reproducibility

Sinalgo supports repeatable randomness, provided that the simulator supplies the only source of randomness used by the researcher. Using the provided source of randomness ensures that con-

sequent simulations obtain the same results when using the same configuration. This guarantee ensures that the simulation produces consistent and reproducible results, particularly valuable for debugging and evaluation.

The simulator's source of randomness is available in the implementations extending the abstract class `Distribution`. By default, Sinalgo includes the following five distribution implementations:

- `ConstantDistribution`
- `ExponentialDistribution`
- `GaussianDistribution`
- `PoissonDistribution`
- `UniformDistribution`

As already mentioned in Section 3.2.3, a project's configuration includes a `random number generators` setting. In this setting, a fixed seed can be specified, which will be used by the `Distribution` class to ensure deterministic behaviour across simulation runs.

3.3 QUANTAS

QUANTAS (Quantitative User-friendly Adaptive Networked Things Abstract Simulator) [23] is a simulator aimed at providing quantitative performance analysis of distributed algorithms. It is implemented using the C++ programming language and is, at the time of writing, not versioned but available at GitHub [25], having been last updated on the 5th of March 2025. While QUANTAS supports some customisation, it is not as extensible as Sinalgo, as it allows the customisation of fewer components.

3.3.1 Environment Simulation

QUANTAS replicates a distributed environment by presenting it as a graph, where the graph's nodes are the simulation's peers and the graph's edges are the simulation's channels.

This simulator carries out the simulation sequentially (in a single thread), only supporting synchronous simulation. Synchronous simulation does not carry the risk of having a peer process a message out of order since every message sent will only be processed in consequent rounds. Therefore, this simulator can speed up the simulation by processing logic in parallel.

Connectivity is decided when the simulation is configured. The simulator starts by creating channels between every peer and assigning a random delay to each (the randomness depends on the distribution provided in the simulation configuration). It is worth noting, however, that there will be two channels in each direction between two peers, and both channels will have the same delay. After setting up the channels, the simulator will inform each peer which of the other

peers are his neighbours, depending on the topology type provided in the configuration. Although channels exist in both directions, peers can only communicate with their neighbours. Therefore, unidirectional connections are possible. Channels also limit the number of messages received per round. This setting can be customised in the configuration. By default, this limit is the maximum integer number possible.

QUANTAS does not provide any method that dynamically changes the connectivity of the peers. However, that behaviour can be implemented in the peer's custom logic using the function `addNeighbor(interfaceId neighborIdAdd)`, which a peer can call.

This simulator organises the rounds in three parallelisable parts. The first is where messages that arrived are moved from the `_inBoundChannels` (which stores messages from other channels into the target peer) into the peers' `_inStream` deque (which represents the messages that arrived). The second is where every peer performs its computation (it is called for every peer using the `performComputation()` function that the researcher can override). This computation should read the incoming messages from the `inStream` deque. The third and last part is the one where messages sent are moved into the target `_inBoundChannels` if the correct conditions are met.

3.3.2 Metric Visualisation

There is no graphical user interface for metric visualisation. However, this simulator offers a different approach to logging. Rather than writing plain text to a log file, QUANTAS formats its log output as a JSON file.

The first key in the log file is `RunTime`, which indicates the total time it took to run the simulation. The second key is called `tests` and is a list that includes the information that the researcher wants to output in the format desired. The configuration includes a setting "tests", which indicates the number of times the simulator will repeat the simulation for the given amount in the mentioned setting. Each repetition appends its log output to the `tests` list. The second key is only present if it contains content. An example of how to log information to the simulator and its respective output format can be seen in Listing 3.9.

```

1 int number_of_messages_sent = 50;
2 LogWriter::getTestLog()["message_sent"] = number_of_messages_sent;
3 LogWriter::getTestLog()["outter_key"]["inner_key"] = "example_text";
4
5 // Outputs the following
6 {
7     "RunTime": ...,
8     "tests": [
9         {
10             "message_sent": 50,
11             "outter_key": {
12                 "inner_key": "example_text"
13             }
14         }
15     ]

```

16 }

Listing 3.9: Example of how to add output to a QUANTAS log

Plain text can still be used for logging by outputting text directly to the console. However, this approach is only reliable if the simulation does not take advantage of the simulator's parallelism. Logs can become out of order or even interleaved if the simulator is processing logic that writes to the console in parallel.

3.3.3 Scenario Configuration

QUANTAS supports configuration files, with some of its settings already mentioned in the previous sections. Like Sinalgo, each project has a configuration file. However, they differ in format, with this simulator's configuration files being in the JSON format.

QUANTAS configuration files have a beneficial characteristic. Each configuration file can include different configurations. The JSON file starts with a key called `experiments`, which contains a list. This list should contain experiments, which are individual configurations for the specific project the configuration file is for. Experiments are a way of running multiple scenarios at once.

An experiment configuration includes multiple required settings as well as some optional ones. They are shown below:

- `logFile` (required): should be the file path to the desired log file. If the value set is `cout`, then instead of writing to a file, the simulator writes to the console.
- `threadCount` (optional): can be used to specify the number of threads the simulator will use. By default, the simulator will use as many threads as the machine running the simulation can handle. Its value must be higher than zero and lower or equal to the number of peers.
- `distribution` (optional): can define the delay characteristics for the random number generator. It can contain up to 3 optional sub-settings, `minDelay`, `avgDelay` and `maxDelay`.
- `topology` (required): contains several sub-settings.
 - `topology.maxMsgsRec` (optional): indicates the limit number of messages a channel can receive per round.
 - `topology.totalPeers` (required): indicates the number of peers the simulation will have.
 - `topology.type` (required): should be one of the supported topologies. At the time of writing, those are: `complete`, `star`, `grid`, `torus`, `chain`, `ring`, `unidirectionalRing`, `userList` and `dynamic`. Some of the mentioned topologies

require additional information, such as, for example, `userList`, which requires a setting `topology.list` with the identifiers of the peers and their corresponding list of neighbours (also represented by their identifiers). The simulator then introduces these topologies by assigning the peers the correct list of neighbours.

- `rounds` (required): indicates how many rounds will be simulated.
- `tests` (required): indicates how many times the experiment in question will be simulated.
- `parameters` (optional): its purpose is to allow the researcher to be able to send custom information for the peers as this setting's value will be then sent to the peer through overriding the function `initParameters`, which must be implemented in researcher's custom peer.

A single file allows for multiple scenarios without creating multiple configuration files, saving time and organising the scenarios by project.

3.3.4 Fault Injection

The QUANTAS simulator does not include any fault injection capabilities natively. However, the researcher can implement fault injection artificially.

In order to implement faults, the researcher can ignore messages when reading from a peer's incoming messages deque. Simulating message-level faults or assigning a flag to the peer and simulating peer-level faults.

3.3.5 Reproducibility

One of the settings the QUANTAS simulator can receive in the configuration file is the `distribution` setting. This setting allows the researcher to choose the simulation's random number generator. The random number generator is important because each pair of peers' channels has their delay randomly set. However, although the random number generator can be chosen, its purpose is no more than to set the channel's delays.

By design, QUANTAS does not respect repeatable randomness and should not be used with that in mind. There is no way to provide the simulator with a fixed seed. Upon start, the simulator generates a seed calculated based on the system's clock summed with a hash of the current thread's identifier. Due to the seed being based on the clock value, which will vary in each simulation, it will always be different. Therefore, each consequent simulation will yield different results.

If the simulator were modified to have a fixed seed, it still would not respect repeatable random in case threads are used. This limitation applies to the generation of the channel's delays and every randomly generated value in the simulation.

3.4 SimGrid

SimGrid [6] is an open-source simulation framework designed to support the development of custom simulators. Rather than being a fixed-purpose simulator, SimGrid provides an interface for implementing distributed systems simulation. It is implemented using the C++ programming language and is, at the time of writing, in version 4.0, which was released on the 11th of March 2025, with a history of regular releases. The interface to interact with the framework is called S4U (SimGrid for you) and is provided in multiple programming languages, those being C, C++, *Python* and *Java*. While the S4U interface is well-supported in C++, the other languages' bindings documentation is incomplete, including some outdated bindings.

3.4.1 Environment Simulation

SimGrid, out of the three simulators studied, is the one which most accurately simulates the functioning of a network.

SimGrid represents machines using *hosts*. Each *host* can have multiple processes called *actors*.

The simulation environment can be implemented manually using fixed values or a configuration file. They include each *host*'s processing power, specified by floating point operations per second, and each link's bandwidth and latency, specified using bytes per second and seconds, respectively. How the configuration files are specified will be further explained in Section 3.4.3.

Although SimGrid's network configuration is more realistic, the simulator abstracts every network communication using a SimGrid concept called `MailBox`. Each *actor* can register a unique mailbox in order to communicate with other *actors*.

This simulator carries out the simulation sequentially (in a single thread), even though there are multiple actors, each representing a process. Due to SimGrid's unique approach to simulation.

Instead of having predefined functions with logic that are called when an event triggers them, like the other two simulators. SimGrid runs a process's code until there is an instruction to stop and wait for an event. Whenever an event regarding the *actor* in question is triggered, it will resume the simulation. Take, for example, the function `get()` from the `MailBox` class. Whenever an *actor* calls the function for a specific mailbox, it stops and is only resumed once the mailbox in question receives a message. When the wanted message arrives, the `get()` function returns the message, and the process resumes. This example can be visualised using the bindings for *Python* in Listing 3.10.

```
1 mailbox = Mailbox.by_name(self.name)
2
3 new_msg = mailbox.get()
4
5 print("The new message is", new_msg)
```

Listing 3.10: Example of message receiving in SimGrid using the Python bindings

Actors can be paused for more reasons than waiting for messages. The researcher can inform SimGrid that an *actor* wants to yield to allow other *actors* to process their logic, tell an *actor* to sleep for some time, and tell the simulation to pause the *actor*'s execution to simulate time spent processing some instructions. When paused, an *actor* gives the control back to the simulation, with the simulation choosing which *actor* resumes their processing.

It is the only simulator that allows for the simulation of processing time (as the researcher can tell the simulation that an *actor* should execute a specific amount of instruction, which will result in the process being paused for the amount of time that the *host* takes to run those instructions, keeping in mind that the amount of instructions per second is included in the *host*'s configuration).

SimGrid also allows the simulation of memory access and concurrency. One of this simulator approach's weaknesses is that it can lead to deadlocks (for example, an *actor* pauses waiting for a message from another, which is stopped waiting for a message from the first).

SimGrid attempts to simulate the environment as rigorously as possible, making it possible to assess the performance characteristics of an algorithm in a specific environment. It also includes network models which more accurately represent different types of communication protocols⁴.

3.4.2 Metric Visualisation

Unlike Sinalgo, SimGrid does not provide a graphical user interface for metric visualisation. However, it does provide with logging capabilities natively.

Because the simulation can involve multiple *hosts* and *actors*, SimGrid includes logging capable of informing the researcher of the simulation time, *host* and *actor* of a log.

This logging feature is available in both the *C++* and the *Python* programming languages, having more customisation options in the *C++* bindings. The *Python* bindings contain four different priority levels, those being, according to the documentation: "debug", "info", "warning" and "error". An example of how to output different log messages in SimGrid's *Python* bindings can be seen in Listing 3.11.

```

1 from simgrid import Actor, this_actor
2
3 def logging_example():
4     this_actor.debug("This is a debug message.")
5     this_actor.sleep_for(1)
6     this_actor.info("This is an info message.")
7     this_actor.sleep_for(1)
8     this_actor.warning("This is a warning message.")
9     this_actor.sleep_for(1)
10    this_actor.error("This is an error message.")

```

Listing 3.11: Example of an actor logging to console in SimGrid

⁴<https://simgrid.org/doc/latest/Models.html#the-simgrid-models>

The code in Listing 3.11 outputs the logs in Listing 3.12. The "debug" log does not appear because the log threshold must be set when executing the simulation. By default, the "debug" priority is the only one below the priority threshold.

```
[HostName:logging_example:(1) 1.000000] [python/INFO] This is an info message.
[HostName:logging_example:(1) 2.000000] ../../src/bindings/python/simgrid_python.cpp
:121: [python/WARNING] This is a warning message.
[HostName:logging_example:(1) 3.000000] ../../src/bindings/python/simgrid_python.cpp
:123: [python/ERROR] This is an error message.
```

Listing 3.12: Output of SimGrid's logging

SimGrid's logging consists of strings and outputs to the console. However, it is possible to imitate QUANTAS's logging approach by writing information into a single data object and outputting it to either a file or the console. Logs will not become out of order or even interleaved by default. However, the researcher can implement multithreading in his approach at his own risk.

3.4.3 Scenario Configuration

The scenario configuration is an important step when creating a simulator in SimGrid. The simulation requires a platform, which is the configuration of the simulated hardware, and a deployment, which specifies the simulation's *actors*. Both the platform and the deployment can be modelled using XML files or directly through code.

The platform configuration informs the simulator of how the network is modelled. It includes information about the *hosts*, specifying an identifier and their processing capacity using floating point operations per second, about the *links*, specifying an identifier, their bandwidth using bytes per second and their latency using seconds, and about the *routes* specifying the source and destination *hosts* using their identifiers and specifying the *links* that connect the two hosts.

The deployment configuration is loaded into the simulation after the platform configuration. However, before loading the deployment configuration, the researcher must register the code's functions corresponding to the *actors*' logic as can be seen, using the bindings for *Python*, in line 20 of Listing 3.13. The *actors*' logic can either be a class or a method. It defines which *actors* are included in the simulation, specifying for each one the *host* it is assigned to (using the *host*'s identifier defined in the platform configuration beforehand). Additionally, it indicates the logic each *actor* should be bound to and provides any custom parameters needed for execution.

An example of how to configure a simulation can be seen in Listing 3.13. The example includes the use of a platform configuration file, the use of a deployment configuration file and the use of modelling both *hosts* and *actors* directly through code.

```
1 import sys
2 from simgrid import Engine, Actor, Host, this_actor
3
4 class Peer:
```

```

5     def __init__(self):
6         self.name = this_actor.get_host().name
7
8         # This is called right away
9     def __call__(self):
10        # Will output "My host is Fafard!"
11        this_actor.info(f"My host is {self.name}!")
12
13 def say_name(name):
14     this_actor.info(f"My name is {name}") # Will output "My name is John"
15
16 if __name__ == "__main__":
17     e = Engine(sys.argv)
18
19     e.load_platform("./platform.xml")
20     e.register_actor("peer", Peer)
21     # Includes the following line <actor host="Fafard" function="peer"/>
22     e.load_deployment("./actors.xml")
23
24     # Creates a new host called server providing processing power
25     e.netzone_root.add_host("server", 25e6)
26     # Creates an actor that uses the method say_name
27     Actor.create("actor_name", Host.by_name("server"), say_name, "John")
28
29     e.run()

```

Listing 3.13: Example of a platform and deployment configuration in Python

The disadvantage of SimGrid's approach is that the configuration is very detailed. Requiring many details that, when aiming at evaluating the behaviour of an algorithm, may not be relevant.

3.4.4 Fault Injection

The SimGrid framework does not include any fault injection capabilities natively. Much like the QUANTAS simulator, it is up to the researcher to implement fault injection.

It can be done by using the `suspend()` function on a *host's actors* to simulate having the *host* offline. In order to have a suspended *actor* become online again, another *actor* must call the function `resume()` on the suspended *actor*. The simulation could have a specific *actor* to handle fault tolerance. An example of an *actor* responsible for suspending and resuming other *actors* can be found in Listing 3.14.

```

1 from simgrid import Actor, this_actor
2
3 def faulty_actor_logic():
4     ... # Actor logic
5
6 def fault_tolerance_monitor():
7     # Instantiates a faulty actor
8     faulty_actor = Actor.create("Lazy", this_actor.get_host(), faulty_actor_logic)
9
10    # Verify if the faulty_actor is suspended

```

```
11     if faulty_actor.is_suspended():
12         faulty_actor.resume() # Resume the faulty_actor
13
14     faulty_actor.suspend() # Suspend the faulty_actor
15     this_actor.sleep_for(10) # Wait for 10 seconds
16     faulty_actor.resume() # Then resume up the faulty_actor again
```

Listing 3.14: Example of an actor that suspends and resumes other actors in SimGrid

3.4.5 Reproducibility

Unlike the other two simulators, SimGrid does not include a built-in random number generator. However, its flexible architecture allows researchers to implement and manage randomness mechanisms as needed.

SimGrid deterministically schedules which *actor* to process next within a single sequential logical process. Therefore, repeatable randomness is achievable if the same random number generator is used and the researcher does not introduce non-determinism. Researchers can ensure that the simulations yield consistent results across multiple runs by using a seed on their preferred random number generator.

3.5 Comparison

The previous sections already compared some of the characteristics of each simulator. However, this section will focus on comparing the characteristics of the simulators and explaining if, how, and why each one is suitable for evaluating distributed statistical learning.

3.5.1 Environment Simulation

Each of the simulators mentioned executes their simulations sequentially. However, their execution model differs. QUANTAS supports only synchronous simulation, meaning that its simulations advance through rounds, enabling the use of multiple threads to speed up the simulation. Sinalgo supports synchronous and asynchronous simulation but does not use multi-threading when running synchronously. SimGrid supports only asynchronous simulation and thus does not have multi-threading as an option.

Although allowing multi-threading speeds up the simulation, synchronous simulation limits the variety of systems a researcher can simulate. As mentioned in Section 3.1.1, a system that can be modelled for a synchronous simulation can also be modelled for an asynchronous simulation, but the opposite is not always true. Due to this limitation, QUANTAS is unsuitable for evaluating distributed statistical learning algorithms.

While both Sinalgo and SimGrid support asynchronous simulation, they handle it differently. A simulation is modelled in SimGrid through a coroutine-like approach where `Processes` execute logic until specifically told to stop, while in Sinalgo's asynchronous simulation, there are

specific methods which are called by the simulator (for example, `handleMessages`, which is called when the node receives a message).

From a researcher's point of view, Sinalgo's approach to asynchronous simulation is more straightforward because the behaviour of the simulator is predictable as opposed to SimGrid's approach, which can lead to *deadlocks* if every *process* is stopped waiting for an event which does not happen (for example, waiting for a message which was not sent).

QUANTAS's latency is decided when the simulation starts by assigning a random value using a distribution (provided in the configuration). However, it is tied to the rounds and does not attempt to represent realistic latency values (only the number of rounds it takes for a message to reach the destination).

SimGrid and Sinalgo handle latency differently by nature. SimGrid attempts to be more accurate out of the box, demanding that the researcher describes the connections between each *actor* by adding to the environment *links* and informing the simulator of which *links* are used between each connection. This simulator utilises routing paths and bandwidth to calculate latency. The network model modifies the calculated latency to achieve realism, but it may add unpredictability to the simulation (not randomness). However, the `raw` network model can be chosen, which does not modify the expected latency value. Sinalgo, on the other hand, is less strict. It allows the researcher to implement its own `MessageTransmissionModel`, which informs the simulator of the time it takes from one node to another. There is no concept of bandwidth or routing built-in.

The method of calculating latency offered by Sinalgo is more flexible than the one provided by SimGrid and more intuitive due to not requiring an entirely realistic network stack, allowing the researcher to set up a simulation environment quickly. However, there is a downside to the flexibility given: the responsibility of accurately modelling delay falls directly on the researcher.

It is important to note that all three simulators compute latency only if the communicating peers are connected. Sinalgo supports dynamic topologies. If the simulation is synchronous, using the connectivity model, Sinalgo updates the connectivity every round. If the simulation is asynchronous, the connectivity model must be triggered manually. QUANTAS and SimGrid require that the network is configured before the simulation starts, but allow the topology to be modified during execution.

Allowing the researcher to modify the connectivity of a system during the execution of the simulation allows for more scenarios to be modelled.

3.5.2 Metric Visualisation

Of the three simulators studied, only SimGrid does not include a built-in method of visualising metrics. However, it allows the researcher to implement its own.

Sinalgo includes logging and a graphical user interface, while QUANTAS provides a unique approach by allowing the researcher to output information to a JSON file. The JSON is a popular and standard file format that can be easily read by any tool the researcher has or creates to interpret the results. Logging is also possible in QUANTAS under certain conditions. Studying and debugging the simulation through log files is simpler than through JSON files. Therefore, JSON

files facilitate the study of the data provided by the simulation, while the log file facilitates the analysis and debugging of the simulation's behaviour.

Another unique characteristic that can be found in Sinalgo is the graphical user interface, which is another intuitive method of visualising the behaviour of the system being modelled. However, since log files provide a persistent and analysable record of the simulation's behaviour, the graphical user interface is less critical when evaluating distributed statistical learning algorithms.

While relying on the researcher to implement its metric visualisation ensures that the researcher can tailor the visualisation to its needs, it also diverts time and effort from the algorithm and simulation analysis toward modifying the simulation process.

3.5.3 Scenario Configuration

The three simulators include configuration files to configure simulation scenarios. Both QUANTAS and Sinalgo's configuration files are required, including many settings that modify the simulator's behaviour and allow the researcher to add custom variables to the simulated algorithms, altering their behaviour. In contrast, SimGrid's configuration files are optional and set up the simulation's environment and *processes*.

A feature unique to QUANTAS is the concept of experiments. In QUANTAS, experiments are defined directly in the configuration file. Each experiment is the configuration of a simulation with specific conditions. Therefore, a single configuration file can be used to execute multiple simulations, allowing the researcher to perform various studies of an algorithm in different environments and conditions. This feature can be handy for evaluating distributed statistical learning algorithms.

3.5.4 Fault Injection

The simulator must include any fault injection capabilities to evaluate how an algorithm behaves in scenarios where faults occur. Sinalgo is the only simulator which includes fault injection built-in. However, unless manually implemented by the researcher, it can only create message-related problems, not problems regarding a peer's normal functioning. The other two simulators require that the researcher implement faults manually. Therefore, Sinalgo is the most suitable for fault injection, although it still lacks peer-related faults.

3.5.5 Reproducibility

Apart from QUANTAS, the simulators allow for reproducibility as long as the researcher does not introduce non-determinism. The simulations obtain the same results by executing in the same conditions by providing a seed in the configuration file.

3.5.6 Conclusions

Each simulator analysed offers distinct advantages, catering to the different requirements for evaluating distributed statistical learning algorithms. QUANTAS stands out for its structured experimental configuration and metric logging using the `JSON` format. Sinalgo stands out for its extensibility, simulation behaviour visualisation and fault injection support. SimGrid stands out for having the highest degree of flexibility and fidelity, especially in modelling asynchronous environments and simulating realistic communication behaviour through bandwidth-aware network topologies.

However, none of the existing simulators can fully satisfy all the requirements for a robust evaluation of the algorithms in distributed environments without the researcher manually implementing some of them. QUANTAS is ruled out because it does not support asynchronous simulation. Sinalgo utilises a node-based approach to the network topology, which, while being a user-friendly abstraction, trades realism for simplicity. This tradeoff is due to the message transmission model calculating the latency dynamically, which means that not only does the responsibility of implementing an accurate message transmission model fall to the researcher, but the latency from one node to another cannot be pre-defined, which rules out the simulation of specific scenarios. Additionally, Sinalgo, unlike SimGrid, does not include bandwidth in the topology, so it does not consider the size of a message when calculating the time it takes for that message to arrive from one destination to another. SimGrid abstracts the logic behind network communication while providing realism. However, it does not include many wanted features built-in, requiring the researcher to implement them.

Given the features and limitations of the studied simulators, a new simulator could be designed to cater to the requirements using the desirable features listed. The new simulator should be tailored specifically to the evaluation of distributed statistical learning algorithms, prioritising realism, extensibility, reproducibility and metric visualisation.

To assess the validity and effectiveness of the proposed simulator, it is essential to compare it against an existing well-established simulator, either Sinalgo or SimGrid. Among the simulators analysed, SimGrid stands out since it still gets regular updates, has a unique approach to the simulator modelling (based on coroutine-like processes) and includes detailed modelling of the network resources (such as latency and bandwidth). These features make it particularly suitable as a reference platform for evaluating distributed statistical learning algorithms. SimGrid will be adopted as the baseline for the experimental evaluation of the new simulator. The comparison will allow for the determination of whether the simulator achieves its intended goals.

Chapter 4

Designing a Discrete Event Simulator

The previous chapter outlined the requirements for a simulator designed to model distributed networks. It examined how three existing simulators met these requirements (if they did at all) and compared the simulators, highlighting the strengths and weaknesses of their respective approaches to discrete event simulation and how useful they are to the evaluation of distributed statistical learning algorithms.

This chapter presents the design and development of a new simulator which attempts to implement the advantages of the studied simulators with the evaluation in mind. The code definitions referenced throughout this chapter can be found in Appendix A, specifically in Section A.2.

4.1 Design Philosophy and Requirements

This new simulator will be built with future research in mind. Because the characteristics and requirements of future research that could take advantage of this simulator are unknown and unpredictable, the simulator should allow the researcher to extend and customise the simulation in any way they see fit. However, the simulator will not take an approach similar to SimGrid, which is very flexible but laborious. It will take an approach closer to Sinalgo, which allows the researcher to model the behaviour by extending pre-defined methods and data structures. This approach enables the researcher to create a simulation and evaluate it faster, as it includes the commodities it needs out of the box. Sinalgo was developed using the object-oriented programming language *Java*, which leverages the concept of inheritance to enable the researchers to customise the simulations. By having abstract classes, the simulator allows the researcher to extend its functionality *overriding* pre-defined functions. This leads to intuitive and predictable extensibility.

The simulator's performance is not a priority but can lead to faster and more complex simulations. Therefore, the *Rust* programming language was chosen as it is fast and memory efficient [29]. However, *Rust* does not include inheritance the same way that *Java* does. The extensibility feature that *Java* offers can be similarly implemented in *Rust* using *Trait Objects* [15], a

concept which defines common behaviour for each data structure that implements it. Due to how *Rust* enforces memory safety, traits bring setbacks to the development of the simulator, requiring that the researcher pays attention to caveats that result from workarounds used.

For example, the *Java* code presented in Listing 4.1. The `Animal` class can easily be used to create multiple classes, such as `Dog`, and each of them will have an identifier which can not be modified, making the `Animal` class fully responsible for it.

```
1 abstract class Animal {
2     private int id;
3
4     Animal(int id) {
5         this.id = id;
6     }
7
8     int getId() {
9         return id;
10    }
11
12    abstract void speak();
13 }
14
15 class Dog extends Animal {
16     Dog(int id) {
17         super(id);
18     }
19
20     @Override
21     void speak() {
22         System.out.println("Woof");
23     }
24 }
```

Listing 4.1: Extending behaviour using inheritance in Java

In *Rust*, however, as can be seen in Listing 4.2, the trait `Animal` can not include the identifier like its *Java* counter-part (it is closer to an interface) and therefore, the identifier must be implemented in each of the structs that implement `Animal`, making the `Dog` class fully responsible for the identifier.

```
1 trait Animal {
2     fn get_id(&self) -> i32;
3     fn speak(&self) -> ();
4 }
5
6 struct Dog {
7     id: i32,
8 }
9
10 impl Dog {
11     pub fn new(id: i32) -> Self {
```

```
12     Self { id }
13 }
14 }
15
16 impl Animal for Dog {
17     fn get_id(&self) -> i32 {
18         self.id
19     }
20
21     fn speak(&self) -> () {
22         println!("Woof")
23     }
24 }
```

Listing 4.2: Extending behaviour using traits in Rust

This requires a workaround because a variable could be crucial to the functioning of the simulator, and therefore, the extended struct should not have complete control over it. The workaround consists of having a new struct called `AnimalInfo`, which contains the identifier, and then passing the responsibility to interact with the identifier to the defined trait. For the `Dog` struct to be unable to access the identifier, the `AnimalInfo` struct must be from another module, and the identifier must be private. Having the trait in the same module as `AnimalInfo` allows the trait to access the identifier even though it is private inside the struct. The new implementation of the module can be seen in Listing 4.3.

```
1 mod animal {
2     pub struct AnimalInfo {
3         id: i32,
4     }
5
6     impl AnimalInfo {
7         pub fn new(id: i32) -> Self {
8             Self { id }
9         }
10    }
11
12    pub trait Animal {
13        fn get_info(&self) -> &AnimalInfo;
14        fn get_info_mut(&mut self) -> &mut AnimalInfo;
15
16        // Default implementation, but can be overwritten
17        fn get_id(&self) -> i32 {
18            self.get_info().id
19        }
20        fn speak(&self) -> ();
21    }
22 }
```

Listing 4.3: The module `animal` which contains the trait and its data

As a result, the `Animal` trait requires that the methods `get_info` and `get_info_mut` are defined for each of the structs that implement the trait. Consequently, the `AnimalInfo` struct must exist as a member of the struct that implements `Animal`. As seen in Listing 4.4, `Dog` includes `AnimalInfo` as `info` and then implements the `get_info` and `get_info_mut` functions.

```
1 use animal::{Animal, AnimalInfo};
2
3 struct Dog {
4     info: AnimalInfo,
5 }
6
7 impl Dog {
8     pub fn new(id: i32) -> Self {
9         Self {
10             info: AnimalInfo::new(id),
11         }
12     }
13 }
14
15 impl Animal for Dog {
16     fn get_info(&self) -> &AnimalInfo {
17         &self.info
18     }
19
20     fn get_info_mut(&mut self) -> &mut AnimalInfo {
21         &mut self.info
22     }
23
24     fn speak(&self) -> () {
25         println!("Woof")
26     }
27 }
```

Listing 4.4: Custom implementation using the `Animal` trait

While this workaround works, it does lead to a lot of repetitive code. Fortunately, *Rust* also includes *macros* [15], a feature that allows repetitive code to be repeated without needing to type everything every time. A *macro* that avoids repetitive code in the example can be created as shown in Listing 4.5, resulting in significantly less code, as is presented in Listing 4.5. The downside of using a *macro* for these declarations is that it abstracts the logic and becomes less readable.

```
1 macro_rules! define_animal {
2     ($structname:ident, $speak:block) => {
3         impl Animal for $structname {
4             fn get_info(&self) -> &AnimalInfo {
5                 &self.info
6             }
7
8             fn get_info_mut(&mut self) -> &mut AnimalInfo {
9                 &mut self.info
10            }
```

```

10         }
11
12         fn speak(&self) $speak
13     }
14 };
15 }

```

Listing 4.5: Rust macro for custom struct that implements Animal

```

1 struct Dog {
2     info: AnimalInfo,
3 }
4
5 impl Dog {
6     pub fn new(id: i32) -> Self {
7         Self {
8             info: AnimalInfo::new(id),
9         }
10    }
11 }
12
13 define_animal!(Dog, {
14     println!("Woof");
15 });

```

Listing 4.6: Custom implementation using the Animal trait and a macro

The focus is on statistical learning. Therefore, the programming language in which the simulator is implemented should support the use of libraries for developing statistical learning algorithms. In this case, the simulator will use the *Rust* programming language, which can make use of a *crate* (the term for libraries that can be imported using *Rust*'s build tool and package manager, *Cargo* [28]) called *faer-rs*, providing both low-level linear algebra routines and a user-friendly high-level interface, all implemented in pure *Rust* [10].

For the evaluation, the researcher will be provided with the tools needed to examine and evaluate the simulation and its results. The simulator will include metric visualisation focusing on the behaviour and the data resulting from the simulation. The metric visualisation will be further expanded on Section 4.6.

4.2 Core Architecture Overview

The simulator is a library that exposes the data structures and functions that the researcher can use. A researcher can implement its custom simulation by creating and modifying specific files. With that in mind, the simulator is divided into three parts.

The first part is the `internal` module, which is the simulator library and where the logic regarding the simulator is implemented. The researcher does not need to tamper with this module.

The second part is the `scenarios` module, where the custom implementations made by the researcher should reside. They need not necessarily be created in this module as long as they are available for the next part.

The third part is the `main.rs` file, where the `Simulator` struct is instantiated and used to register which custom implementations the simulator should consider. It acts as a bridge between the custom and the internal logic while also being responsible for starting the simulator. The researcher should modify this file to register the custom implementations.

The structure of the simulator can be seen in Figure A.1, found in Appendix A. However, note that two different elements are inside the `scenarios` module. These are examples of scenarios which are to be implemented by the researcher and, therefore, are not part of the simulator.

Regarding the `internal` module, it is divided into three modules, `simulator`, `core` and `cli`. The `simulator` module contains the already mentioned `Simulator` struct, the `cli` module includes logic regarding the usage of command line arguments and the `core` module consists of the logic regarding the simulator's functioning and its components. The `Simulator` struct receives information about the simulation through the `cli` module and then starts the simulation by interacting with the `core` module. How these components interact can be seen in Figure 4.1, which presents the researcher, the components mentioned, and two other components `Scenario` and `engine`. The components not described yet can be found in the `core` module and are key to the simulation. The `Scenario` component is a trait which the simulator provides for the researcher to extend by adding the logic that should be simulated. The `engine` component is a module which contains the `run` function, which is responsible for the simulation loop.

Regarding the `scenarios` module, it should include the scenarios which are to be simulated. For a better organisation, it is recommended that the researcher divides those scenarios into modules, as seen in Figure A.1. The simulator can run the scenarios defined only if the researcher registers them in the `main.rs` file. To be registered, the scenario must be defined using the `Scenario` trait which requires the implementation of three functions, those being `name()`, which returns a string that names the scenario, `description`, which returns a string that describes the scenario, and `start(context, simulator, opts)` which does not return anything but is the entry point of the simulation for that scenario. The `Simulator` struct includes a function `run_scenario(name, context, simulator, opts)` which calls the `start` method on the scenario whose `name()` method returns the `name` argument provided. As seen in Figure 4.1, the `Scenario` trait, by itself, does not start the simulation. The researcher must call the appropriate method `engine::run(context, hooks, deadline_opt)` so that the simulation loop starts. Section 4.3 explains how the simulation loop works.

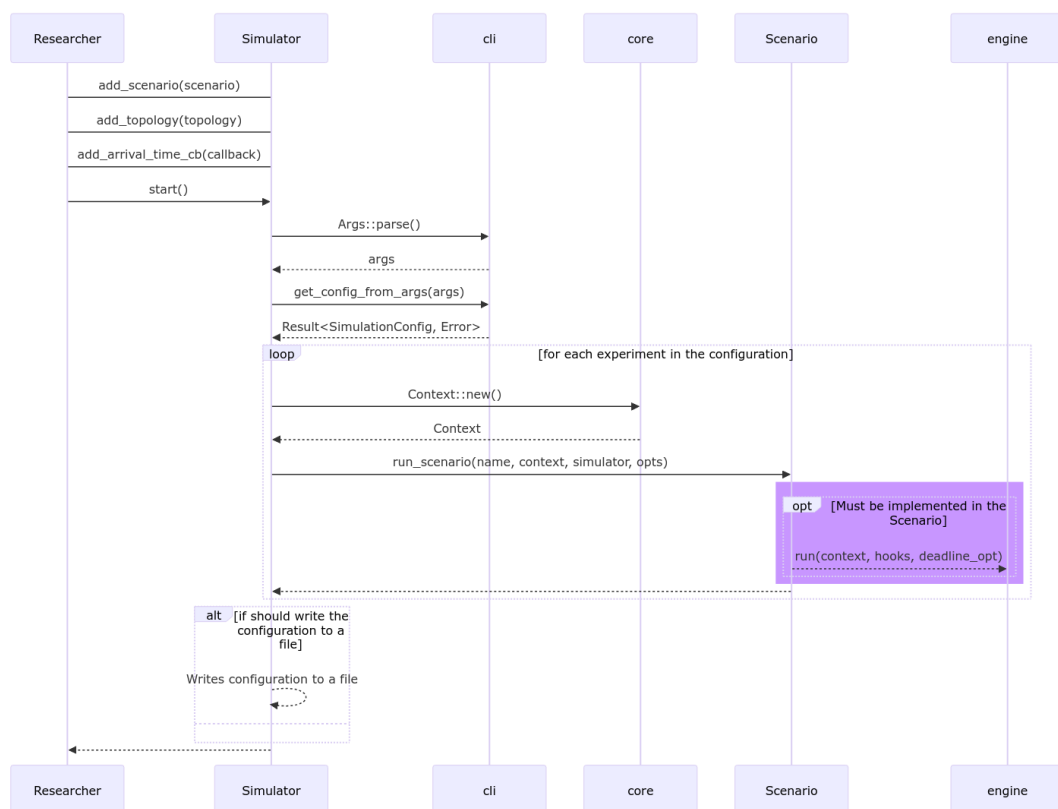


Figure 4.1: The simulator's logic upon starting

4.3 Event System Design

4.3.1 Events

In a discrete-event simulator, changes in the state are the consequence of the occurrence of events. Instead of advancing time continuously, the simulator progresses time by processing the earliest unprocessed event. Events are then dynamically introduced into the system.

Because events are such a crucial part of the simulation, the type of events that the simulator contains are reserved for the functionalities it offers. In this case, the simulator wants to simulate the communication between peers and, therefore, must include an event related to the communication.

Although the occurrence of events can be perceived during the simulation, the researcher should not be responsible for directly creating the events during the implementation of an algorithm. For that purpose, the simulator abstracts the creation of the events by providing methods. For example, sending a message is done through a method called `send_message_to`, which creates a `MessageDeliveryEvent` event.

The simulator contains only two types of events, those being `MessageDeliveryEvent` and `TimerEvent`, which will be explained in further detail in Section 4.4 and Section 4.5 respectively.

4.3.2 Event Queue

As mentioned, the simulator progresses time by processing the earliest unprocessed event, and for that to happen, the simulation loop must be able to calculate which event should be processed next and know how to process it. With that in mind, a trait called `Event` was created, which includes the methods called `timestamp`, `process`, `id` and `set_id`. The first should return the time at which the simulator should process the event. The second does not return anything but instead includes the logic to execute when (and if) the simulation's clock reaches the event's timestamp. The third and fourth are complementary because two events can have the same timestamp. Before an event is added to the simulation, the simulator assigns it an identifier (`set_id(id)`), which informs the simulator that, between two events with the same timestamp, the event with the lowest identifier is the earliest. Each event the simulator includes implements the trait `Event`.

To keep track of the events to be processed and to know which one is the earliest unprocessed event, the simulator requires a data structure that acts like a queue where events can be added and later removed. If the events were added in order, then a queue would suffice. However, the order in which the events are introduced into the data structure can not be predicted. Seeing as the two primary operations that the data structure will execute are to add events and to remove the event with the smallest timestamp, data structures like `Min-Heap` and `Max-Heap` are a good fit for the task because they have $O(1)$ time complexity for the `push` operation and $O(\log(n))$ time complexity for the `pop` operation. *Rust*'s standard library does not include two different structures for both `Min-Heap` and `Max-Heap`, instead it provides only the `BinaryHeap` structure,

which, by default, is similar to the `Max-Heap`. Since the simulator requires the earliest unprocessed event, the event with the smallest timestamp, the `Min-Heap` would be the best option. To use a `Min-Heap`, the `BinaryHeap` can be adapted by using the helper struct `Reverse`, which makes the `BinaryHeap` behave like a `Min-Heap`. As a result, the structure, which will be called `event_q` (event queue), is defined as `BinaryHeap<Reverse<EventType>>`. When an event is created, it is added into the `event_q`. Once the simulation loop wants to retrieve the earliest unprocessed event, it removes the event with the smallest timestamp from the `event_q`.

The mentioned definition of the event queue includes a comparable type that has not been discussed yet, which is called `EventType`. Events were told to have a common trait `Event`, and while *Rust* does allow the use of a trait inside a data structure which holds other structures, such as the `BinaryHeap`, through the use of the `Box` struct, it is not as efficient as using an enum. Using the `Box` struct with the trait is not as efficient because the event queue becomes polymorphic, which means that the compiler does not know the types that can be present in the queue at compile time. Not knowing the types (and therefore not knowing the sizes beforehand) leads the compiler to use virtual dispatch instead of the faster static dispatch. Using an enum that implements the `Event` trait allows the collection to be monomorphic, resulting in the compiler using static dispatch while still exhibiting polymorphic behaviour. This is only possible because there is a fixed variety of events (in this case, only two), so when, for example, the method `timestamp` is called on an element of the event queue, the implementation of the `Event` trait uses pattern matching to call the method `timestamp()` on the correct struct. If the types are unknown, using the boxed trait is best. Considering the advantages of using an enum instead of a trait, the `EventType` enum was created and used in the event queue.

4.3.3 Simulation State

Having defined the events and the event queue necessary for the simulation loop, only the simulation state is missing. In this case, the simulation state is a struct that contains all the information about the simulation and, therefore, must be passed into every function call used by the simulator that attempts to either visualise or modify the state. This struct has been mentioned in some of the function calls which have been discussed previously and can also be seen in Figure 4.1. It is called `Context` and is created before a scenario is run. The contents of the struct will be further discussed in the following chapters but can be seen in Listing A.1

With the simulation state and the event queue established, the simulator can now execute the simulation loop. Figure 4.2 presents a diagram illustrating the simulation loop.

As can be seen in the figure, the simulation loop starts when the function `run(context, hooks, deadline_opt)` from the `engine` module is called. This function introduces the simulation state to the simulation loop and includes two other arguments, which are called based on the simulation loop. The `deadline_opt` parameter is an optional value which, if defined, does not allow the simulation loop to process events after its value. The `hooks` parameter is a struct of type `SimulationHooks` containing two functions. Its definition can be seen in Listing A.2. The hooks are two functions that the researcher can set beforehand. However, only one of them is used

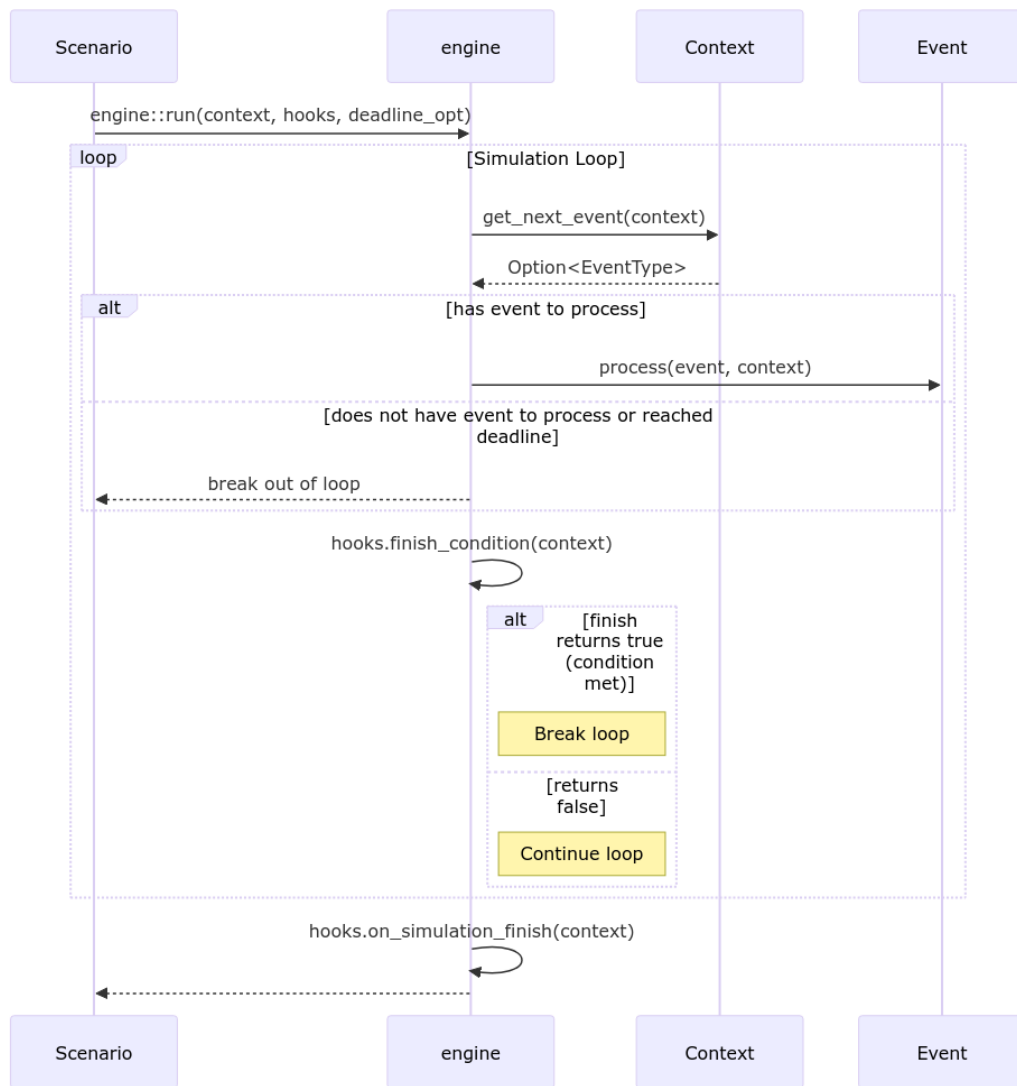


Figure 4.2: Selected items' reference graph

during the simulation loop. That hook is the one that provides the `finish_condition` function, which is called in each iteration of the simulation loop and can stop the loop if the conditions it contains are met (if it returns true). If neither the finish condition hook nor the deadline is defined, the simulation loop will continue until the event queue is empty.

It is worth noting that events can not be added to the simulation with timestamps lower than the simulation clock. If that happens, the simulator ignores the event, informing the researcher that an event's timestamp was lower than the simulation clock.

4.4 Network Topology and Communication

The simulator already has events, an event loop and a state, but it still does not simulate a distributed network. To simulate a network that contains many peers that can communicate with each

other, the simulator must include the peers, the links between the peers, and message-sending mechanisms.

4.4.1 Peers

The simulator takes an approach closer to Sinalgo, which has an abstract class called `Node` that provides methods that the researcher can override. The approach taken by Sinalgo takes advantage of Java's inheritance to provide extensibility. To be able to replicate the behaviour, the simulator will use *Trait Objects* as explained in Section 4.1. The name used for the trait with a function similar to the `Node` abstract class from Sinalgo will be `CustomPeer`.

A peer must include the `PeerInfo` struct whose definition can be seen in Listing A.3. This struct contains information necessary for the simulator's functioning but not as crucial for the researcher's custom logic. It includes information such as an identifier (`id`), whether the simulation is aware of its existence (`instantiated`), if the peer is alive (`alive`), its position (`position`) and the function that must be called every time the peer receives a message (`on_message_receive`).

To define a new peer, the researcher must create a struct containing whatever data it wants the peer to have and the `PeerInfo` struct with the name `peer_info`. Then, a *macro* can simplify its creation. Take Listing 4.7 as an example. A peer is created in the listing with `peer_info` and a number value. The macro `define_custom_peer` takes care of implementing the trait `CustomPeer` with its functions `get_peer` and `get_peer_mut`.

```

1  #[derive(Default)]
2  pub struct ExamplePeer {
3      pub peer_info: PeerInfo,
4      pub value: u64,
5  }
6
7  define_custom_peer!(ExamplePeer);

```

Listing 4.7: Implementation of a custom peer in the simulator

Creating the struct that implements the `CustomPeer` trait is enough to define a new peer. However, to add the peer to the simulation, the researcher must manually create and add a new peer instance to the simulation state (context). This process is done in the `run` method in the chosen `Scenario`. Creating a new peer instance requires the `peer_info` field to be constructed. As mentioned, the `PeerInfo` struct contains fields necessary to the simulator's functioning and should not be manually initialised. Instead, the researcher should use the provided methods to initialise the field, which requires intervention. The methods provided are `new(x, y, z)` which instantiates a new `PeerInfo` struct on the position provided using fields `x`, `y` and `z`, and `with_on_message_receive` which sets the function callback which must be called when the

peer in question receives a message. The message receival functionality will be further explained in Section 4.4.4. Using these methods and even assigning values to them is optional.

To add a peer to the simulation, the researcher must call the method `add_peer(context, custom_peer)`, which belongs to the `engine` module. This method adds the peer to a vector that holds every peer in the simulation and creates its respective links `IndexMap` whose purpose and functionality will be explained in Section 4.4.2. The peer vector is stored in the `Context` struct. When a peer is added to the context, the method named `instantiate` belonging to the `CustomPeer` trait is called with an argument representing the peer's identifier. As a result, the peer's identifier is set, and the `instantiated` field is set to true. A peer's identifier is its position in the context's vector of peers.

4.4.2 Connections

For peers to communicate, they must be aware of which other peers they can send messages to. For that purpose, the simulation state keeps track of every peer connection. The connections between peers are called links, and when a peer is connected to another, they are considered neighbours. However, links can be unidirectional. Therefore, if peer A is connected to peer B, but B is not connected to A, then B is A's neighbour, and A is not B's neighbour.

Each peer in the simulation state must contain its respective links `IndexMap` (a structure which behaves the same as a `HashMap`). The `IndexMap` has the identifier of the peer's neighbours as key and a variable of a type called `LinkInfo` as value. The `LinkInfo` type is an optional `LinkKind` enum (`Option<LinkKind>`), with the `LinkKind` enum consisting of the three enumerated types `Latency(f64)`, `Bandwidth(f64)` and `Full{bandwidth: f64, latency: f64}` (their definition of the type and enum can be seen in Listing A.4). The value associated with the link between a pair of peers is used to calculate the time it takes for a message to go from the sending peer to the receiving peer. The sending of messages and the effects of the `LinkInfo` value will be further discussed in Section 4.4.4.

In order to connect two peers, the simulator provides two methods from the `engine` module. Those methods are the `add_oneway_link(context, from, to, info)` method, which adds a new link from the peer with identifier `from` to the peer with identifier `to`, and the `add_twoway_link(context, from, to, info)` method, which does the same as the first method, but adds an additional link from the peer with identifier `to`, to the peer with identifier `from`. As a result, the two-way link is the same as a bidirectional connection between two peers. Therefore, both are neighbours of each other.

If a researcher wants to study the behaviour of an algorithm in a particular topology, they can create the topology by calling the provided methods to connect the peers. However, suppose the researcher wants to study the algorithm's behaviour in multiple topologies. In that case, they can create a custom reusable function that uses the provided methods to connect peers programmatically. This reusable function will be further explained in Section 4.9.2.

4.4.3 Messages

When a peer wants to communicate with another peer, it sends a message. Messages are holders of information that travel through the network. Since the network is only simulated, messages are holders of information assigned to an event and then delivered to the destination peer when the event they belong to is processed.

The researcher can create custom messages by implementing the trait `Message` in a custom struct. This trait requires that the researcher implements a method `size_bytes` which returns the size of the message in bytes. The `Message` trait includes a default implementation of the `size_bits` method that takes the value from the `size_bytes` method call and multiplies it by 8. This makes it so that the `size_bytes` method becomes mandatory. However, the researcher can implement the `size_bits` method for more fine-grain control over the returned value. If the researcher overrides the default method, the value returned by the `size_bytes` method is unimportant. The definition of the `Message` trait is presented in Listing A.5. A custom implementation of a message can be seen in Listing 4.8.

```

1 #[derive(Debug, Clone)]
2 pub struct FlowUpdatingPairwiseMessage {
3     pub sender: usize,
4     pub flow: f64,
5     pub estimate: f64,
6 }
7 impl Message for FlowUpdatingPairwiseMessage {
8     fn size_bytes(&self) -> u64 {
9         1 + 8 + 8
10    }
11 }

```

Listing 4.8: Custom Message implementation

4.4.4 Communication

The researcher is given access to a method called `send_message_to`, as was mentioned in Section 4.3.1, which encapsulates the simulator’s logic for emulating network communication. When the researcher calls the `send_message_to(context, from, to, message)` method, they must provide it with an already initialised message struct. The method makes sure the peer with identifier `from` has a connection to the peer with identifier `to`, calculates the time which the message should take to reach the destination peer and finally creates a `MessageDeliveryEvent` that contains the message delivery timestamp, the identifiers of the sender and the receiver, and the provided message instance. The definition of the `MessageDeliveryEvent` struct can be seen in Listing A.6. The calculation of the time it takes for a message to reach its destination depends on the link between the sender peer and the receiving peer.

If there is no link between the peers, the `MessageDeliveryEvent` is not created, and therefore, the message is not sent. In case the link exists and `LinkInfo` contains a value of the enum type `Latency(f64)`, then the time it takes to reach the destination is equal to the value the enum type contains. Otherwise, whether `LinkInfo` does not include a value or its value's enum type is `Bandwidth(f64)`, the simulator calculates the time it takes to reach the destination using a function from a field called `message_delay_cb` (message delay callback), which is part of the `Context` struct. The callback is defined before the simulation loop starts and will be mentioned again in Section 4.9.3. This callback allows the researcher to implement a custom dynamic method of calculating the delay between peers. The `message_delay_cb` field is of type `MessageDelayCallback`, which is an alias for a function that receives the simulation state and the identifiers of the two peers involved. After the delay is calculated through the callback if the `LinkInfo` included a value with enum type `Bandwidth(f64)`, then the time it takes to reach the destination gets added to the size of the message divided by the link's bandwidth. This way, the data transfer physical limit of a connection is simulated. Before the `MessageDeliveryEvent` is created, the simulator also allows the researcher to add jitter to the message delay. In a network, jitter is a property that indicates how uneven or inconsistent the delay is between consecutive packets reaching their target. The `Context` struct includes a field called `jitter`, which is of type `Jitter`, a struct that contains a distribution and a multiplier. This distribution and multiplier are used to calculate the jitter value that then gets added to the message delay. The `jitter` field must be defined by the researcher, or else its value will always be zero.

After calculating the time it takes to reach the destination, in order to add the message to the simulation, the simulator creates a new `textttMessageDeliveryEvent` instance. The timestamp provided when creating a `MessageDeliveryEvent` instance is the sum of the simulation state's clock time with the recently calculated message delay value. To add the new event to the simulation, the simulator uses the method `add_event(context, event)`, which is part of the engine module.

When the simulator processes a `MessageDeliveryEvent`, it invokes a callback named `on_message_receive` belonging to the destination peer. The message receipt method is of type `OnMessageReceiveCallback`, which receives the context, the sender and the receiver's identifiers, and the sent message. As mentioned in Section 4.4.1, the researcher can set this callback, allowing him to customise how peers deal with message receiving. If the callback is not set, by default, it does nothing.

As a result of this procedure, the message-sending mechanism was abstracted to a single event, which, when it gets instantiated, decides when it should be processed and instantly delivers its content to the moment that the simulation loop processes.

4.5 Timers and Asynchronous Logic

Sinalgo includes timers, a unique characteristic allowing the researcher to create events that execute custom logic outside the peer's logic. As it provides flexibility and extensibility to the

simulator, it was adapted to *Rust*.

It was mentioned before that the simulator has two types of events. The event type not yet discussed is the `TimerEvent`. Its definition can be seen in Listing A.8.

A `TimerEvent` struct contains two fields: the timestamp, which the simulator `Event` trait requires and that indicates when the simulation loop should process the event and a field called `timer` of type `Box<dyn Timer>` which means that it includes a boxed trait. The trait `Timer` is used to define new timers and requires only that the struct that implements it has a method `fire(timer, context)`. This method is called when the simulation loop processes a `TimerEvent`.

To create a new timer, the researcher must create a new struct that can hold any necessary data. The struct then must implement the `Timer` trait and its `fire` method. An example timer implementation is shown in Listing 4.9. Looking at the listing, it is also possible to notice how timer events are added to the simulation. The `add_timer(context, timestamp, timer)` method of the `engine` module is responsible for abstracting the procedure of adding an event to the simulation loop. By repeatedly adding the same timer to the simulation, it is possible to create events that periodically execute custom logic, noting that they have access to the simulation state and, therefore, can be used to visualise it.

```

1  #[derive(Debug, Clone)]
2  pub struct ExampleTimer {
3      pub interval: f64,
4  }
5
6  impl Timer for ExampleTimer {
7      fn fire(&self, ctx: &mut Context) {
8          println!("ExampleTimer fired!");
9          engine::add_timer(ctx, ctx.clock + self.interval, self.clone());
10     }
11 }

```

Listing 4.9: Implementation of a custom timer in the simulator

Timers can also be used to run custom logic on specific peers. They allow the simulator to simulate asynchronous logic, not interrupting the main flow of the simulation. A peer could perform logic independently of whether it is receiving messages. For example, in a real-world scenario, a peer could have two threads, one dealing with communication and another performing calculations. Asynchronous logic is practical when the peers are supposed to execute logic every tick as is done in the pairwise and collect all versions of the flow updating algorithm [1].

4.6 Observability and Monitoring

Understanding how a system behaves and responds to different conditions and environments is essential for improving its performance and reliability and assessing its theoretical foundation.

Observability and monitoring are critical in the context of simulating distributed environments for evaluating statistical learning algorithms. These mechanisms in a discrete event simulator allow researchers to track what, when, how, and why events occur.

Monitoring tracks specific, predefined metrics to assess the system's performance. Observability refers to the system's ability to expose the internal state through outputs (such as logs, metrics and traces). Therefore, monitoring helps the researcher understand what is happening in the system, while observability allows the researcher to understand why a particular behaviour is observed. By incorporating timestamps into the logging mechanisms, the simulator provides insight into when specific behaviours take place. Together, these mechanisms allow the researcher to determine how a behaviour emerged, enabling comprehensive evaluation and debugging of distributed statistical learning algorithms.

The observability and monitoring mechanisms were implemented in the `log` module. This module contains methods which can be accessed and used anywhere in the simulation. However, some of them require that a `Context` instance is initialised. This requirement is due to the ability to set options for the mechanisms.

These mechanisms are configured through a struct called `Logger`, whose definition can be consulted in Listing A.16. This struct is initialised when the `Context` struct is and can be configured using the struct implementation methods. The configuration of the `Logger` struct will be further explained in Section 4.10.

Before explaining the different types of logging that the simulator provides, a specific hook was created for the researcher to visualise information. The hook in question is a function named `on_simulation_finish`, which is one of the hooks mentioned in Section 4.3.3 and the one that was not called during the simulation loop. This hook is a function that takes the simulation state (`context`) as an argument. It is executed after the simulation loop is finished and has access to the final simulation state, allowing the researcher to perform a final examination. It is useful, for example, to calculate whether the algorithms simulated were successful. The other hook mentioned in Section 4.3.3, although not meant for logging, since it is executed every iteration of the simulation loop, can be repurposed to inspect the simulation state periodically.

4.6.1 Logging Architecture

The simulator takes an approach inspired by both Sinalgo and QUANTAS. Sinalgo contains a global logger object that writes plain text into either the console or a file. Plain text logs allow the researcher to follow the simulation logic intuitively. Sinalgo also includes different log functions which separate each text log by priority. This is a helpful feature, as it provides the researcher with a method of filtering which logs they want to analyze, thereby enabling the study of the simulation's behaviour. QUANTAS, on the other hand, has a unique feature: the use of a `JSON` object to register data. Since `JSON` is a widely adopted and intuitive format for storing structured data, it enables metrics storage for later consulting and analysis.

Building on the advantages of Sinalgo's and QUANTAS' approaches, the simulator's `Logger` struct provides the researcher with behaviour and metric logging. The logs and metrics can be

output directly to the console or stored in the file system, depending on the researcher's choice. Since plain text and JSON are distinct formats, behaviour and metric logs are written to separate files. Moreover, while JSON is a widely adopted format for structured data and can be interpreted programmatically, it is not as easily read as plain text.

4.6.2 Behaviour Logging

The plain text logs correspond to the behaviour logging. These logs are meant to describe what is occurring in the system. But because some occurrences are more relevant to the researcher than others, the `LogLevel` enum is created to differentiate between logs. The definition of this enum can be seen in Listing A.17 and includes the following levels, ordered by priority: `Trace`, `Debug`, `Info`, `Warn`, `Error` and `Internal`.

When the `Logger` struct is instantiated, it is provided with a level from the `LogLevel` enum because, in case the logger is meant to output the plain text logs into the console, the logs which are not as relevant are ignored and, therefore, do not occupy screen space, allowing the relevant logs to distinguish themselves. In case the plain text logs are set to be stored in a file, every log is written independently of its priority.

The `log` module, which contains the `Logger` struct and the `LogLevel` enum, includes multiple methods that the researcher can use to write logs. There are six priority levels, each with two corresponding log methods. The existence of two methods per level is because, to be able to filter the logs written, the log method should have access to the `Logger` struct instance (because it stores the priority level of the logger and the destination files) that belongs to the context, and therefore, it must have access to the context. As a result, the methods were divided into regular and global log methods. For example, for priority level `info`, the regular log method is `log::info(context, text)` while the global log method is `log::global_info(text)`. Global log methods can be used anywhere in the simulation and are written only to the console. This type of log is never stored in a file because the method cannot access the context's logger.

When plain text logs are written, they are divided into three parts: The first part shows the simulation clock's timestamp at which the log method was called. If the log is global, since it does not have access to the simulation clock, the timestamp number is replaced by the text "GLOBAL". The second part shows the logger level of the log in question. The third part is where the provided text is displayed. Some example logs can be seen in Listing 4.10.

```
[0.4974363502356993] [TRACE] ExamplePeer with id 2 received a message
[0.5186263475744203] [SIMUL] SIMULATION LOOP FINISHED
[0.5186263475744203] [INFO] Are the calculated values correct? yes
[GLOBAL] [SIMUL] FINISHED SIMULATION LOOP, SEED IS "1234567"
```

Listing 4.10: Example of simulation logs output to console

The different priority levels serve various purposes. The `Trace` level (seen in the logs as `[TRACE]`) is intended for tracking function calls and logic-related operations executed by the

simulator. The `Debug` level (`[DEBUG]`) is used for logs related to the researcher's implementation. The `Info` level (`[INFO]`) is used for informing the researcher that something occurred. The `Warn` level (`[WARN]`) brings to attention that an action did not have the expected outcome, but its consequences are not problematic. The `Error` level (`[ERROR]`) is reserved for unwanted behaviour which impacts the simulation and, therefore, should be addressed. The `Internal` level (`[SIMUL]`) informs the researcher of significant simulator events.

When logs are written to a file, the researcher can inspect a specific priority level by executing the following command on the command line:

```
cat filename.log | grep "\[TRACE\]"
```

In this case, the command would retrieve every log of the priority level `Trace`.

4.6.3 Metrics Logging

The `JSON` format logs correspond to the metrics logging. These are meant to store analytical information about the simulation.

The `QUANTAS` simulator implements this feature by providing a global object that can be written to and, as soon as the simulation is finished, is written to a file. A problem with the `QUANTAS` approach is that metrics are only written once the simulation stops, if for any reason the simulation comes to a halt before finishing, the metrics are lost. The simulator designed solves this problem by, instead of having a global `JSON` object, providing a log method `metrics(context, title, data)` which receives a `JSON` object (the parameter named `data`), serialises it, and instantly writes it to a file (note that, as mentioned earlier, the metrics are not output to the console, only to a file). The act of writing to the file as soon as possible allows the researcher to inspect its values even if the simulation stops before finishing.

The log method `metrics(context, title, metrics)` receives the following parameters: the simulation state (`context`), a string (`title`), and a `JSON` object (`metrics`). When provided the arguments, the method in question creates a `JSON` object that contains the keys: `timestamp`, `title` and `metrics`. The first's value is the timestamp of when the log was created, the second's value is a title to be associated with that log, allowing the researcher to divide this type of log into categories, and the third key's value is the `JSON` object provided by the researcher. An example of creating a metrics log can be seen in Listing 4.11.

```
1 let data = json!({
2   "name": "John",
3   "age": 23,
4   "grades": vec![15.5, 17.7, 18.0],
5 });
6 log::metrics(ctx, "student", &data);
```

Listing 4.11: Logging a `JSON` object example

Listing 4.12 presents the output of the code in Listing 4.11.

```
{
  "metrics": {
    "age": 23,
    "grades": [15.5, 17.7, 18],
    "name": "John"
  },
  "timestamp": 0.24199828773564375,
  "title": "student"
}
```

Listing 4.12: Example of an output of a metrics log

The *Python* programming language is popular for data analysis and graph creation. Seeing as the metrics' logs are stored in a file which contains `JSON` objects, as can be seen in Listing 4.13, these logs can be loaded with minimal effort into a python script using just a few lines of code. The listing, which for simplicity's sake lacks guard rails, not only parses a metrics file but also filters the logs by the key "title" and orders the logs by the key "age" from the `JSON` object present in key "metrics".

```
1 import json
2
3 filename = "metrics.jsonl"
4 title_filter = "student"
5 entries = []
6
7 with open(filename) as f:
8     for line in f:
9         entry = json.loads(line)
10         if entry.get("title") == title_filter:
11             entries.append(entry)
12
13 entries.sort(key=lambda x: x.get("metrics", {}).get("age", 0))
```

Listing 4.13: An example of a Python3 script which parses a metrics log file

The metrics logs can also be parsed using the command line by executing the `jq` program as follows:

```
jq -s 'map(select(.title == "student")) | sort_by(.metrics.age) | .[]' data.jsonl
```

4.7 Fault Injection

Fault injection is a testing technique used to assess a system's robustness and fault tolerance. In this case, the simulator should provide the researcher with tools that deliberately introduce faults to the system so that its reaction can be observed and studied.

The capability of introducing faults to the system is a characteristic that the other simulators studied did not include out-of-the-box, apart from Sinalgo. However, Sinalgo's built-in fault injection capabilities are limited to the messages. It does not simulate situations where a peer becomes unreachable because it stops processing logic. Even though peer-based faults can be simulated in the other simulators studied, they can not be done without manual intervention by the researcher.

The simulator allows the introduction of both message-based faults and peer-based faults.

4.7.1 Message-Based Faults

4.7.1.1 Dropped Messages

The researcher can introduce a message-based fault where messages are not delivered by providing a message drop rate (a value from zero to one) as input. This rate is stored as a field in the `Context` struct and is evaluated when the `send_message_to(context, from, to, message)` method in the `engine` module is used.

When the simulation attempts to send a message from one peer to another, the `drop_rate` field is compared to a randomly generated number in the interval $[0, 1)$. If this random number is less than the drop rate value, `MessageDeliveryEvent` is not created, resulting in the message not being sent, effectively simulating a message drop.

4.7.1.2 Duplicate Messages

The researcher can also introduce another message-based fault, which is the duplication of messages, also implemented in the method named `send_message_to` from the `engine` module. This trait requires that the researcher provides a message duplication rate (a value from zero to one) as input. This rate is stored as a field in the `Context` struct and is evaluated when the `send_message_to(context, from, to, message)` method in the `engine` module is used.

When the simulation attempts to send a message from one peer to another, and the message is not dropped, the `duplicate_rate` field is compared to a randomly generated number in the interval $[0, 1)$. If this random number is less than the duplication rate value, two `MessageDeliveryEvent` structs which contain the same message are created, effectively delivering two identical messages to the same peer. Resulting in a duplicate message.

4.7.1.3 Jitter

The last message-based fault the researcher can introduce into the simulation is jitter. It indicates how uneven or inconsistent the delay is between consecutive packets reaching their target. It does

not introduce faults as directly as the other message-based faults. Since the simulation adds the jitter value to the expected message arrival time, messages can be processed later than expected, which may lead to the simulation of out-of-order messages.

This value is also determined based on a field stored in the `Context` struct and is evaluated when the `send_message_to(context, from, to, message)` method from the engine module is used. However, the `jitter` field is neither a rate nor a number. It is a struct which contains a distribution and a multiplier. The distribution is one of the types in the `Distribution-Wrapper` enum, which can be consulted in Listing A.18. The multiplier is a floating point value.

When the simulation attempts to send a message from one peer to another, the simulator calculates a jitter value and adds it to the previously estimated arrival time before creating the corresponding `MessageDeliveryEvent` event (it can create two events if the message is duplicated). Therefore, when a message is duplicated, the original and its duplicate have different arrival times.

The jitter value is a floating point number that is calculated by invoking `Context` struct's `get_jitter_value()` method which, in turn, multiplies the `multiplier` field by a random value obtained by providing the `distribution` field to the method `get_value(context, distribution)` from the `distributions` module. After calculating the jitter value, it is added to the previously estimated arrival time. Therefore, it adds variation and uncertainty to the communication process.

When the simulation attempts to send a message from one peer to another, and the message is not dropped, right before creating the `MessageDeliveryEvent` event, the jitter value is calculated by calling the `Context` struct's method `get_jitter_value()` which multiplies the multiplier with a random value obtained by the distribution. After calculating the jitter value, it is added to the previously estimated arrival time. Therefore, it adds variation and uncertainty to the communication process.

If the researcher wants to include jitter but does not know which value to input, the simulator has an option to use a default jitter value. The default jitter value was calculated by performing 3 million pings to the 8.8.8.8 address and measuring the round trip time. Afterwards, the round trip times were loaded into a *Python* script, halved, and subtracted the minimum value. This process leads to the values now representing only the travel time variation of the network. The frequency distribution of values up to the 95th percentile can be seen in the plot in Figure 4.3. Using the *Python* library `scipy`, the function `weibull_min.fit` was then used to find the values for the shape and scale of a Weibull distribution. The values were 1.064 for the shape and 2.872 for the scale, which means that the default value is equivalent to the researcher inputting `Weibull(1.064, 2.872)` for the distribution. Compared to the frequency of the values, the distribution fit can be seen in Figure 4.4. The multiplier value is 0.001 because the values output by the distribution are in seconds. Therefore, the multiplier converts them into milliseconds.

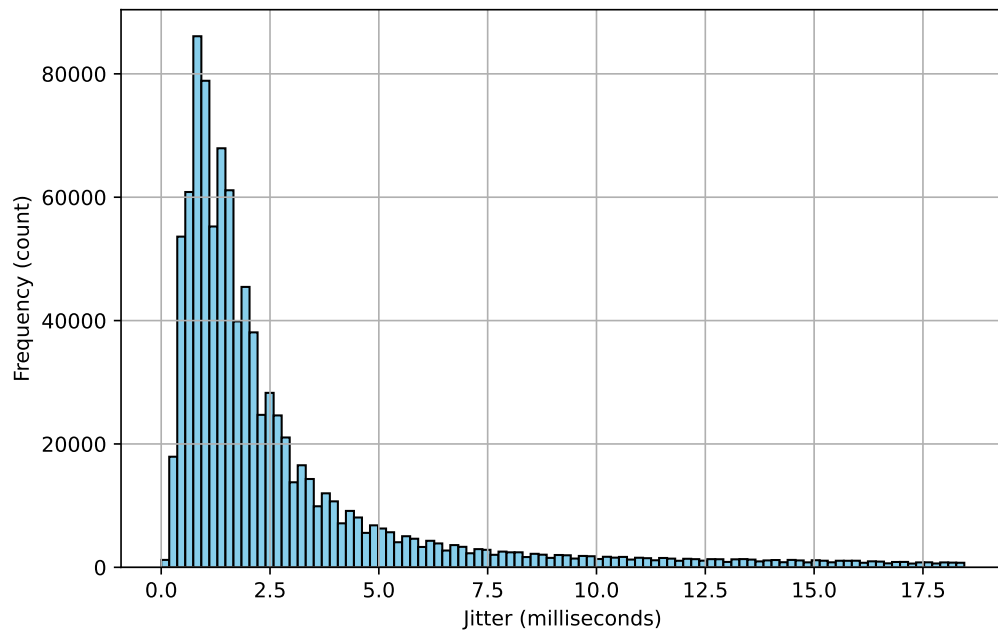


Figure 4.3: Histogram of jitter values up to the 95th percentile

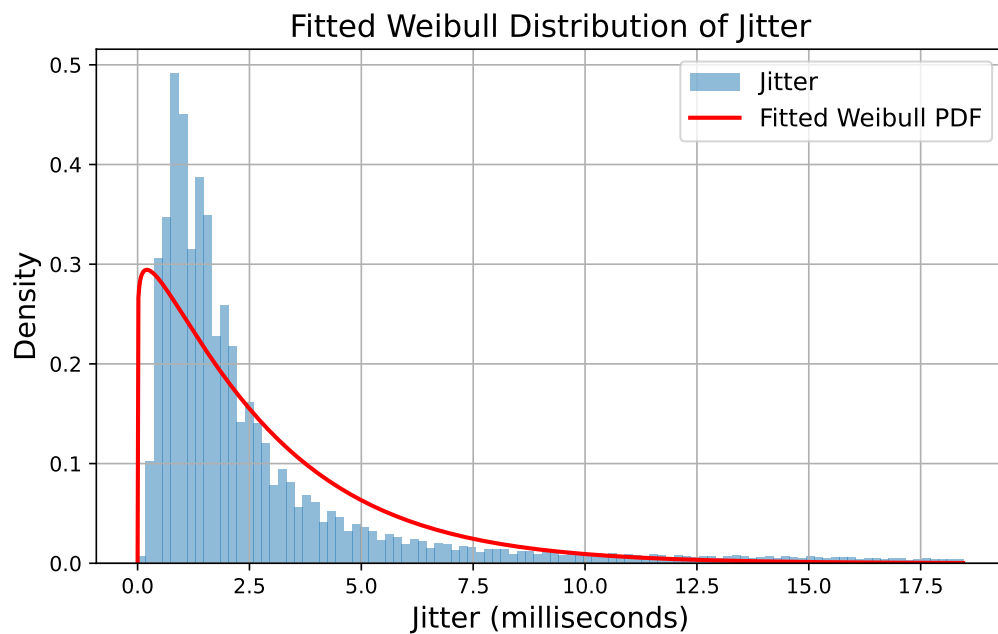


Figure 4.4: Weibull fit to jitter data

4.7.2 Peer-Based Faults

The existence of peer-based faults is one factor that differentiates the simulator from the simulators studied before. In this case, it is implemented using the `alive` flag, which is found in the `PeerInfo` struct (whose definition can be consulted in Listing A.3).

Whenever the researcher wants to simulate a peer being unable to process requests, they can call the `kill()` method found in the `CustomPeer` trait. This method sets the `alive` bool to false and can only be set back to true using the `revive()` method, also found in the `CustomPeer` trait.

The `alive` flag is used when the `MessageDeliveryEvent` event is processed to check if the receiving peer can process the message being delivered. The message delivery is ignored if the peer in question is not alive.

To schedule failures and recoveries, the researcher can use timers. For example, a researcher could create a timer that, once fired, kills a peer and immediately creates another timer that is responsible for reviving the peer. Listing 4.14 shows an example timer that solely kills a predefined peer.

```

1  #[derive(Debug, Clone)]
2  pub struct KillTimer {
3      pub target: usize,
4  }
5
6  impl Timer for KillTimer {
7      fn fire(&self, ctx: &mut Context) {
8          if let Some(target) = ctx.peers.get_mut(self.target) {
9              target.kill();
10             log::info(ctx, format!("Peer {} killed.", self.target));
11         }
12     }
13 }
14
15 // The code below is added in the scenario 'start' method
16 ...
17
18 // Get random value from interval [1, 10]
19 let activation = OrderedFloat(ctx.rng.random_range(1.0..=10.0));
20 // Kill peer with id 0 on timestamp 'activation'
21 engine::add_timer(ctx, activation, KillTimer { target: 0 });

```

Listing 4.14: Custom timer that kills peers

If the researcher implements custom logic using timers, which a peer should not process if he is not active (alive), then the responsibility to check whether the peer in question is alive falls on the researcher. The `CustomPeer` trait provides a function `is_alive()` that returns a bool indicating whether the peer is active or not.

4.8 Reproducibility

Simulations can be complex, leading to slight differences snowballing into significant changes to the simulation's result. Because of this, it becomes increasingly difficult for the researcher to inspect or debug a scenario if each run of said scenario has different results. A unique situation could drastically change the simulation's results. For example, if an out-of-order message is rare

but possible, the researcher could have a specific simulation run where the randomly generated jitter value results in a single out-of-order message. As a result, the researcher could fail to pinpoint the origin of the fault because each consecutive simulation run does not include any out-of-order message.

The simulator solves this problem by providing a single source of randomness that can be configured to output the same values in each simulation. This behaviour is obtained by providing a seed, which is a number of type `u64`, to the random number generator.

Guaranteeing determinism is only possible if the researcher does not introduce non-determinism into the simulation. To avoid that, the simulator must provide the single source of randomness for the researcher to use, too. Therefore, the random number generator resides in the `Context` struct as the `rng` field (the definition of the `Context` struct can be found in Listing A.1). Making sure that it is available to the researcher.

The introduction of a seed to the random number generator is optional. If the researcher does not provide a seed number, the simulator creates a random one. Upon the end of a simulation, the seed is logged so that the researcher can provide it for later use if they want to explore the same simulation conditions again.

To guarantee determinism, the simulator also considers other factors that can not use the single source of randomness. For example, the simulator uses the `IndexMap` data structure instead of the `HashMap` data structure because, for the same input, in different executions, the order of the keys of the `HashMap` data structure changes (due to *Rust*'s implementation). In contrast, the same does not happen in the `HashMap`.

4.9 Registry Systems

The simulator has been mentioned to require that the researcher register the scenarios in order to run them. The simulator should be provided with the name of the scenario it is to run. This way, the simulator allows the researcher to run different scenarios quickly without modifying the code. The mechanism that allows the researcher to indicate which scenario to run without modifying the code is the registry design pattern.

The registry design pattern is functionally similar to a `HashMap`. When registering, for example, a function in the registry, a key must also be provided. This key is then used to be able to call the registered function. The registry is helpful for the simulator because the researcher can implement a scenario once, register it into a scenario registry and give it a name in the form of a string, for example: "example_scenario". When the researcher wants to execute a simulation on the example scenario, they need not modify the code to run the scenario. He can provide the scenario name as input and use the registry to run the scenario. Another reason for using the design pattern for this purpose is that due to *Rust* requiring that the simulator is compiled before running, having to modify each simulation's code can be time-consuming.

The simulator can receive input when executing. The input receipt will be explained further in Section 4.10. For now, consider that the input is passed as command line arguments and are, therefore, strings.

The simulator includes three registries: one for scenarios, one for topologies, and one for arrival time callbacks. The researcher interacts with these registries through the `Simulator` struct. The `Simulator` struct's definition can be consulted in Listing A.9.

4.9.1 Scenario Registry

The `ScenarioRegistry`, whose definition can be examined in Listing A.10, is a struct that contains a single field named `scenarios` which is an `IndexMap` with strings for keys and two-element tuples for values. The first element of the two-element tuple is of the `ScenarioFn` type, which is an alias for a function whose parameters are of the same types as the `Scenario` trait's `start` method. The second element of the two-element tuple is another string.

Although the scenario registry should register scenarios implemented using the `Scenario` trait, the registry does not store instances of structs that implement the `Scenario` trait. Instead, the values of each trait's methods are used in the `IndexMap`. The output of the `name()` method is used as the key, the output of the `description()` method is the second element of the two-element tuple, and the method named `start` is the first element of the two-element tuple (not the output, but the function as a whole).

The `Scenario` trait, whose definition can be seen in Listing A.11, had its definition explained in Section 4.2. Therefore, it won't be presented once again.

A custom scenario can be defined by creating a struct that implements the `Scenario` trait. Take, for example, the `ExampleScenario` struct that is presented in Listing 4.15.

```

1 pub struct ExampleScenario {}
2
3 impl Scenario for ExampleScenario {
4     fn name() -> &'static str
5     where
6         Self: Sized,
7     {
8         "example"
9     }
10
11     fn description() -> &'static str
12     where
13         Self: Sized,
14     {
15         "An example simulation."
16     }
17
18     fn start(_ctx: &mut Context, _simulator: &Simulator, _opts: ExperimentOptions) {
19         // This scenario does not start the simulation loop unless it
20         // it adds events to the simulation and starts it by calling
21         // the 'engine' module's 'run' method.
22     }

```

23 }

Listing 4.15: Implementation of a custom scenario

After the custom scenario is defined, it must be imported into the `main.rs` file and added to the simulator using the `add_scenario::<S>()` method, where `S` is a struct that implements the `Scenario` trait (as can be consulted in Listing 4.16). The simulator then calls the `register::<T>()` method implemented in the `ScenarioRegistry` struct, registering the scenario in its `scenario_registry` field.

```

1 fn main() {
2     Simulator::default()
3         .add_scenario::<ExampleScenario>()
4         .start();
5 }
```

Listing 4.16: Adding a custom scenario to the simulator

When the simulator starts, it receives the name of a scenario as input. It then provides the scenario's name to the scenario registry when calling the method named `run_scenario`. If the scenario is registered in the registry, then it is executed.

4.9.2 Topology Registry

The topology registry works similarly to the scenario registry. This registry is used to store different topology functions. A topology function is, as mentioned in Section 4.4.2, a custom reusable function that takes advantage of the link-related methods provided by the `engine` module to connect peers programmatically.

The topology registry is represented by the `TopologyRegistry` struct, whose definition is available in Listing A.12. This registry also contains an `IndexMap`. However, the values of this data structure are not tuples but functions because the `Topology` trait requires a name and a function but not a description.

The `Topology` trait's definition can be seen in Listing A.13. It contains a method named `connect`, which receives the context, the number of peers to connect, and an optional list. The optional list will be further explained in Section 4.10.

Like scenarios, a custom topology can be defined by creating a struct that implements the `Topology` trait. The simulator includes a macro named `define_custom_topology`, which allows the researcher to implement the `connect` function and assign it to the macro while also providing the name for the struct and the name that should be returned by the `name()` function. Take, for example, the implementation of the star topology presented in Listing 4.17.

```

1 fn star_topology(ctx: &mut Context, n_peers: usize, _custom_list: Option<Vec<
    ConnectionInfo>>) {
2     let center_idx = 0;
3     for i in 1..n_peers {
4         engine::add_twoway_link(ctx, i, center_idx, None);
5     }
6 }
7
8 define_custom_topology!(StarTopology, "star", star_topology);

```

Listing 4.17: Implementation of a custom topology

After the custom topology is defined, much like the scenario example used above, it must be imported into the `main.rs` file and added to the simulator using the `add_topology<T: Topology>()` method, where `T` is a struct that implements the `Topology` trait (as can be seen in Listing 4.18). The simulator then calls the `register::<T>()` method implemented in the `TopologyRegistry` struct, registering the topology in its `topology_registry` field.

```

1 fn main() {
2     Simulator::default()
3         .add_scenario::<ExampleScenario>()
4         .add_topology::<StarTopology>()
5         .start();
6 }

```

Listing 4.18: Adding a custom scenario to the simulator

When a scenario is executed, it receives a struct with information about the topology chosen, including an optional string field that indicates the name of the topology. Then, if the researcher implements it in the scenario's `start` method, the information about the topology is passed to the topology registry's method named `connect_peers`. This method checks whether the topology name is defined and executes it if it exists in the registry.

4.9.3 Arrival Time Registry

The arrival time registry is used to store different arrival time callbacks. An arrival time callback is, as mentioned in Section 4.4.4, the callback that allows the researcher to implement a custom dynamic method of calculating the delay between peers.

The arrival time registry, which is defined using the `ArrivalTimeRegistry` struct, whose definition is available in Listing A.14. Like the topology registry, this registry contains a data structure of type `IndexMap` with the callback names as keys and their corresponding functions as values.

The `ArrivalTimeCallback` trait's definition can be observed in Listing A.15. It contains a method named `callback`, which takes the context and two peer identifiers as arguments. A custom arrival time callback can be defined by creating a struct that implements the

ArrivalTimeCallback trait. Similar to the topology, the simulator also includes a macro that takes the struct's name, the arrival time callback, and the body of the callback function. Take, for example, the implementation of a callback that calculates the time a message should take to travel from one peer to another based on the peer's positions, presented in Listing 4.19.

```

1 // optical fiber latency per kilometer in seconds
2 const OPTICAL_FIBER: f64 = 0.350e-6;
3
4 // In kilometers
5 fn distance_between_points(a: (f64, f64, f64), b: (f64, f64, f64)) -> f64 {
6     ((a.0 - b.0).powi(2) + (a.1 - b.1).powi(2) + (a.2 - b.2).powi(2)).sqrt()
7 }
8
9 fn distance(
10     ctx: &mut Context,
11     from: usize,
12     to: usize
13 ) -> Option<OrderedFloat<f64>> {
14     let (from_peer, to_peer) = (ctx.peers.get(from)?, ctx.peers.get(to)?);
15
16     let dist = distance_between_points(
17         from_peer.get_peer().position,
18         to_peer.get_peer().position
19     );
20
21     Some(OrderedFloat(dist * OPTICAL_FIBER))
22 }
23
24 define_custom_arrival_time_callback!(
25     DistanceBasedArrivalTime,
26     "distance",
27     |ctx, from, to| { distance(ctx, from, to) }
28 );

```

Listing 4.19: Implementation of a custom arrival time callback

After the custom arrival time callback is defined, it must be imported into the `main.rs` file and added to the simulator using the `add_arrival_time_cb::<A>()` method, where `A` is a struct that implements the `ArrivalTimeCallback` trait (as can be seen in Listing 4.20).

```

1 fn main() {
2     Simulator::default()
3         .add_scenario::<ExampleScenario>()
4         .add_topology::<StarTopology>()
5         .add_arrival_time_cb::<DistanceBasedArrivalTime>()
6         .start();
7 }

```

Listing 4.20: Adding a custom arrival time callback to the simulator

When a scenario is executed, it receives an optional string field that indicates the name of the arrival time callback to be used during the simulator. Then, if the researcher implements it in the scenario's `start` method, the optional string is passed to the arrival time registry's method named `get_callback`. This method is responsible for returning a callback function. If the passed arrival time callback name is not defined or does not exist in the registry, the method returns a default function that the simulator uses as a fallback and whose callback always returns the number 1. Otherwise, if the callback name passed is defined and exists, the method returns it. This returned callback is then assigned to the context field `message_delay_cb`, where it is accessed and used in the `send_message_to` method of the `engine` module.

4.10 Configuration Management

The simulator must be compiled before being run. When the simulator binary is executed a help message is shown in the command line. This is because the simulator supports two ways of running a simulation, either by having the researcher provide the information necessary through command line arguments or through a configuration file.

Some settings can not be easily defined as command line arguments, therefore, the configuration file should be used to have access to every feature as it is more flexible.

4.10.1 Command Line Argumentes

The simulator uses a crate called `clap` to interpret the arguments provided through the command line. This crate allows the definition of a struct whose fields translate to arguments. The definition of this struct, which is called `Args` and belongs to the `cli` module, can be consulted in Listing A.19.

There are five arguments that can execute tasks in the simulator by themselves, those are:

- `config` - requires the path to a configuration file.
- `scenario` - requires the name of the scenario to be run.
- `list_scenarios` - lists the names and corresponding descriptions of the scenarios registered in the simulator.
- `list_topologies` - lists the names of the topologies registered in the simulator.
- `list_arrival_times` - lists the names of the arrival time callbacks registered in the simulator.

When the program receives one of the last three arguments, it outputs their content (obtained from the registries mentioned in Section 4.9) to the console and then exits. An example is shown in Listing 4.21. The first two arguments, however, can be provided along with other arguments.

```
./target/release/tdes --list-scenarios
```

Listing 4.21: Example run of the simulator with the list-scenarios argument

The `config` argument can be used with the following arguments:

- `flush_threshold` - requires a number that selects the number of logs the logger holds before writing to the file. A higher number translates to a lower simulation time, but if the simulation comes to a halt unexpectedly, more logs can be lost. If not set, its value is 200.
- `write_config` - a flag that, if set, makes the simulator override the configuration file used with one generated by the simulator that represents the simulation executed. If not set, its value is `false`.

The `scenario` argument can be used with the following arguments:

- `seed` - a number that is used for seeding the simulation's source of randomness. If not set, the simulation generates a random one.
- `repetitions` - a number used to repeat the simulation a provided amount of times. If not set, its value defaults to 1.
- `use_jitter` - a flag that, if set, tells the simulation to use the default jitter configuration `Weibull(1.064, 2.872) * 0.001` mentioned in Section 4.7.1.3. If not set, the jitter value will always be 0.
- `arrival_time` - a string which should be the name value of one of the arrival time callbacks registered in the simulator. If not set, the arrival time callback used will be a built-in one called 'constant' which always returns 1.0.
- `n_peers` - a number that indicates how many peers should be created in the scenario simulation. If not set, its value is 5.
- `topology` - a string which should be the name value of one of the topologies registered in the simulator. If not set, no topology function is called.
- `drop_rate` - a number that should be from 0 to 1 and indicates which percentage of messages should be dropped before sending. If not set, its value is 0.0.
- `duplicate_rate` - a number that should be from 0 to 1 and indicates which percentage of messages should be duplicated when sending. If not set, its value is 0.0.
- `deadline` - a number that tells the simulation loop at which timestamp should stop processing events if still iterating. If not set, the simulation loop only stops when there are no more events to process or when the finish condition is met.

- `dir` - the path to a directory (the directory gets created if it does not exist) where the configuration file and the log results are written.

As can be deduced, the `scenario` argument allows the researcher to quickly run a simulation by providing options, being able to generate a configuration file for later use. While the `config` argument is supposed to execute the simulation described on a file.

A flag argument that can be used with both `scenario` and `config` is the `quiet` option, which suppresses the output of logs to the console.

4.10.2 Configuration File

A characteristic common to each of the studied simulators is the existence of configuration files (SimGrid does not have a specific configuration file, but it includes files that configure the network and the elements involved in it). A configuration file is an intuitive and quick method of repeating a simulation and documenting the conditions for later inspection.

The simulator includes a single configuration file that takes inspiration from the configuration files in the simulators studied. However, it uses a format different from that of the other simulators. The format used is the `YAML` data serialisation language. This format was chosen because it is designed to be easy to read and write and allows for the use of aliases and anchors. Aliases and anchors enable the researcher to avoid having to repeat the same values over and over again in a file. Take, for example, the `YAML` file in Listing 4.22. Anchors are identified by the `&` character, while aliases are identified by the `*` character. The listing shows that simple values can be declared, such as the ones for key `district`, as well as more complex values, such as those for key `info`. The aliases are replaced with the anchor's values. Therefore, the parser interprets the listing as the Listing 4.23. Therefore, if the researcher wants to reuse a specific value multiple times, they can do so using `YAML` anchors and aliases. This not only makes the file more readable but also ensures that updates to the value can be made in one place, reducing the risk of typos causing inconsistencies or errors.

```

1 azores: &azores Azores
2 porto: &porto Porto
3 dog_type: &dog
4   type: Dog
5   sound: bark
6 cat_type: &cat
7   type: Cat
8   sound: meow
9
10 pets:
11   - name: Simba
12     age: 1
13     district: *azores
14     info: *dog
15   - name: Yona
16     age: 8
17     district: *porto

```

```

18     info: *dog
19   - name: Ollie
20     age: 4
21     district: *porto
22     info: *cat

```

Listing 4.22: YAML's aliases and anchors example

```

1 pets:
2   - name: Simba
3     age: 1
4     district: Azores
5     info:
6       type: Dog
7       sound: bark
8   - name: Yona
9     age: 8
10    district: Porto
11    info:
12      type: Dog
13      sound: bark
14   - name: Ollie
15     age: 4
16     district: Porto
17     info:
18       type: Cat
19       sound: meow

```

Listing 4.23: YAML's aliases and anchors replaced example

Using the `scenario` argument allows the researcher to run a single scenario with some provided options. However, if the researcher wants to tweak a single setting, such as, for example, the arrival time callback, to be able to compare the results, they need to execute two commands, one at a time. A configuration file can make it easier for the researcher to tweak the simulation. Still, if the file only sets up a single scenario and its options, the problem is not solved, leading to multiple configuration files (and, consequentially, one command for each file). With that in mind, the simulator takes inspiration from the QUANTAS simulator and introduces the notion of experiment.

An experiment describes a scenario and its settings and can be found in a configuration file. It solves the problem presented because each configuration file supports multiple experiments. Therefore, if the researcher wants to compare the results of various variations of a single scenario, they can create a single simulation file that describes each variation as an experiment. As a result, the researcher only needs to execute a single command in the command line to run multiple scenarios.

The simulator represents each experiment internally through the `Experiment` struct, whose definition can be consulted on Listing A.20. This struct includes 10 fields, all of which will be

explained.

Starting with the `name` field, this value is only required to differentiate between experiments. The logger uses this value to store the logs in folders representing each experiment, enabling the identification of the logs. Logs are stored in a folder called *results* inside the folder where the configuration file is stored, the structure of the folder containing the `config.yaml` file can be seen in Listing 4.24 which was generated by running the command `tree -charset=ascii` example.

```
example
|-- config.yaml
'-- results
    '-- experiment1
        '-- 1750204180
            |-- experiment1_0.jsonl
            '-- experiment1_0.log

3 directories, 3 files
```

Listing 4.24: The output of the `tree` program on a folder containing a simulation configuration file and its logs

The `scenario` field points to the scenario's name that the researcher wants to execute and, as such, must have been previously registered. The field is a string corresponding to one of the keys in the `ScenarioRegistry`. If the researcher wants to consult which scenarios they can pick, they can use the previously mentioned `list-scenarios` argument. The simulator calls the `run_scenario` method present in the `ScenarioRegistry` struct, which will, if a scenario by the name provided exists, invoke the `start` method of the scenario in question. If the scenario does not exist, no scenario is simulated, and the researcher is warned.

The `seed` field is optional and indicates whether the simulation's random number generator should use a pre-defined seed or if it should generate a new one. This field's value is passed to the `Context` struct's initialisation method called `new`. The simulation's context is instantiated before the scenario is run because other fields from the experiment's configuration may depend on it.

Field `arrival_time`, apart from being optional, is similar to the `scenario` field, pointing to a name used in a registry. In this case, the registry responsible for the arrival time callback is represented by the `ArrivalTimeRegistry` struct. However, unlike the scenario registry, the simulator is not responsible for calling the registry's activation function. The researcher could want the arrival time callback to be a specific function (and no other) for that scenario. Suppose the researcher wants the callback to be chosen using the configuration file. In that case, he can do so as the `arrival_time` field gets passed as an argument to the scenario's `start` method (using the `ExperimentOptions` whose definition can be observed in Listing A.21). Therefore, the researcher could do as shown in Listing 4.25. Note that the `Context` struct's `on_message_cb` field defines a simulation's arrival time callback. As can be deduced from the listing, assigning the

arrival time callback is the researcher's responsibility. In case the researcher implements the code shown in the listing but does not provide the `arrival_time` field in the configuration file, the registry returns, as a default response, the built-in arrival time callback `ConstantArrivalTime`.

```
1 ctx.message_delay_cb = simulator
2   .arrival_time_registry
3   .get_callback(opts.arrival_time);
```

Listing 4.25: Manual assigning of a custom arrival time callback

The `topology` field is of type `TopologyInfo`, this is an object that can have four keys: `n_peers`, `name`, `connections` and `positions`. Only the first key is required and corresponds to the number of peers that the simulation should instantiate. This field is required, but the implementation is up to the researcher, therefore, not only can the researcher ignore the number of peers, an additional field called `positions` can be used to provide pre-defined positions. This additional field is a list of tuples of whether two, or three numbers. The topology has to be implemented by the researcher in each scenario, therefore, it is also provided to the `start` method through the `ExperimentOptions` struct. With fields `n_peers` and `positions` in mind, the researcher can create peers as shown in Listing 4.26 which shows an example of the configuration on Listing 4.27. This listing instantiates as many peers as the `n_peers` field indicates, assigning their corresponding position from the configuration file if defined, generating a random position if not.

```
1 for i in 0..opts.topology.n_peers {
2   let (pos_x, pos_y, pos_z) =
3     match opts.topology.positions.as_ref().and_then(|v| v.get(i)) {
4       Some(&(px, py, pz)) => (px, py, pz.unwrap_or(0.0)),
5       None => (
6         ctx.rng.random_range(-10.0..=10.0),
7         ctx.rng.random_range(-10.0..=10.0),
8         0.0,
9       ),
10    };
11
12    engine::add_peer(ctx, ExamplePeer::new(pos_x, pos_y, pos_z));
13 }
```

Listing 4.26: Manual assigning of a peers' positions

```
1 experiments:
2 - name: experiment1
3   scenario: example
4   topology:
5     n_peers: 14
6     positions: [
```

```

7      [1.0, 2.0, 0.4],
8      [-10.0, 5.0, 0.0],
9      [3.0, 0.0, ~],
10     ]

```

Listing 4.27: Configuration with number of peers and some of the peers' positions

Having instantiated the peers, the simulation needs to know how to connect them. The `name` field is optional and represents the name used by the `TopologyRegistry`. Similarly to how the scenario's implementation of the arrival time callback was responsibility of the researcher, the topology is too. The researcher can call a custom topology function by adding the code on Listing 4.28 to the scenario's `start` method. In case no topology name is provided, or the topology name is not registered in the corresponding registry, the simulator does not connect the peers, warning the researcher in the second case.

```

1 simulator
2     .topology_registry
3     .connect_peers(ctx, opts.topology);

```

Listing 4.28: Manual invocation of the custom topology function

The invoked topology function contains three parameters: the simulation state (context), the number of peers (provided in the configuration file) and an optional list of connections. The number of peers is provided to the topology connection function so the simulator knows how many they must connect (which peers, too, as peers' identifiers are in ascending order). The optional list of connections allows the researcher to create pre-defined connections between peers, adding flexibility to the simulation.

The simulator includes two built-in topology connection functions, which, along with the optional list of connections, allow the researcher to fully customise the network. Each element of the optional list of connections must include three values. The first two are the identifiers of the peers, while the last one is of type `LinkInfo`. The elements of the list are used as arguments in the `add_oneway_link` and `add_twoway_link` functions from the `engine` module. The researcher can not choose which of the functions to use for each connection, but using the built-in functions, the researcher can add the links unidirectionally (`onewaycustomtopology`) or bidirectionally (`twowaycustomtopology`). An example of the usage of the declaration of custom unidirectional connections can be seen in Listing 4.29. Note that by inverting the identifiers and maintaining the same `LinkInfo`, the simulator creates two unidirectional links that correspond to a single bidirectional link. The listing shows the declaration of a list of links that are connected using the `onewaycustom` function. This list translates into the use of bandwidth for the unidirectional link from 0 to 1, the use of latency for the unidirectional links from 0 to 2 and 2 to 0, meaning that it is equivalent to a single bidirectional link between 0 and 2, the use of both

bandwidth and latency for the unidirectional link from 0 to 3, and lastly, the use of a single unidirectional link from 0 to 4 that use neither bandwidth nor latency. The simulator calculates latency using the arrival time callback function when it is not provided.

```

1 experiments:
2 - name: experiment2
3   scenario: example
4   topology:
5     n_peers: 8
6     name: onewaycustom
7     connections:
8       - [0, 1, !Bandwidth 100]
9       - [0, 2, !Latency 0.1]
10      - [2, 0, !Latency 0.1]
11      - [0, 3, !Full {bandwidth: 100, latency: 0.4}]
12      - [0, 4, ~]

```

Listing 4.29: Configuration file containing the declaration of connections

Additionally, the simulator includes the topology functions `full`, `star` and `ring` that are included into the simulator by default.

Fields `drop_rate`, `duplicate_rate` and `jitter` are related to the introduction of message-based faults, having been explained in Section 4.7.1. The first two fields are numbers that should be between 0.0 and 1.0. The simulator will clamp the value and warn the researcher if the value provided is outside of the allowed interval. On the other hand, the `jitter` field is an object that must contain the distribution to be used and the multiplier. The first is one of the enum types from the `DistributionWrapper` enum, and the last is a number. An example of the `jitter` field being declared in the configuration file can be seen in Listing 4.30. Each of the three fields, `drop_rate`, `duplicate_rate` and `jitter`, is optional. They are provided to the `Context` struct using the struct's `set_drop_rate` and `set_duplicate_rate` methods immediately after the struct is instantiated. If not explicitly set, their default values are 0.0 for both `drop_rate` and `duplicate_rate`, and the `Nothing` enum type combined with a multiplier value of 0.0 for the `jitter` field. Listing 4.30 also includes the declaration of the `drop_rate` and `duplicate_rate` message-based faults' fields.

```

1 experiments:
2 - name: experiment3
3   scenario: example
4   topology:
5     n_peers: 5
6     drop_rate: 0.05
7     duplicate_rate: 0.001
8     jitter:
9       distribution: !Weibull [1.064, 2.872]
10      multiplier: 0.001

```

Listing 4.30: Configuration file containing the declaration of message-based faults

The `deadline` field is an optional number representing a timestamp that tells the simulation loop not to process events after that timestamp. It is passed to the scenario as an argument through the `ExperimentOptions` struct and then passed as an argument to the `run(context, hooks, deadline_opt)` method in the `engine` module. Since it is optional, its value (internally) will be `Option<f64>`, and therefore, when not set, the simulation loop checks that its value is `None` and, thus, stops. The existence of a deadline does not mean that the simulation will stop at that specific timestamp. It can finish earlier if the finish condition is met or there are no more events to process. The deadline guarantees that the simulation does not finish after the specified timestamp.

The `repetitions` field is an optional number that specifies how many times an experiment will execute. If omitted, it defaults to 1. This field allows researchers to run the same experiment multiple times to collect results to compare. Each repetition will use a different random seed if the experiment's `seed` field is not provided.

The last field is called `extra_args`. It is an optional value of type `serde_yaml::Value` that allows the researcher to pass custom information into the simulation. The simulator passes it as an argument to the scenario's `start` method through the `ExperimentOptions` struct. Then, it is the researchers' responsibility to parse the information. The researcher can take advantage of the fact that the field's value is of type `serde_yaml::Value` to create a specific struct that would then be deserialised, resulting in a format that is easily understood by whoever reads the code and wants to understand which extra options there are. Deserialising is not obligatory, however, and therefore, the researcher can do as shown in Listing 4.31 that reads a flag from the `extra_args` field and creates a timer that introduces a peer-based fault. A configuration file for the previous listing can be found in Listing 4.32.

```

1 if let Some(custom) = opts.extra_args {
2     if let Some(Value::Bool(true)) = custom.get("kill_peer") {
3         engine::add_timer(ctx, OrderedFloat(0.1), KillTimer::new(0));
4     }
5 }

```

Listing 4.31: Using the configuration's extra arguments to introduce a peer-based fault

```

1 experiments:
2 - name: experiment3
3   scenario: example
4   topology:
5     n_peers: 5
6   extra_args:
7     kill_peer: true

```

Listing 4.32: Configuration file containing the declaration of message-based faults

As has been shown, although the simulator includes many features, the researcher still needs to connect them to the simulation. Some other features mentioned in this chapter do not take part of the configuration file, for example, the hooks mentioned in Section 4.6 and Section 4.3.3. However, using the `extra_args` field, the researchers can extend the simulator's functionality, enabling, for example, variable matching to select which hooks to use.

4.11 Implementation Considerations

The simulator is designed to balance extensibility and flexibility with a set of strict rules that control its behaviour. These rules facilitate the researcher's interaction with the simulator by reducing the need to understand how it works internally. Thereby, the researchers can focus on creating simulation scenarios instead of having to implement mechanisms to evaluate them.

Extensibility is primarily achieved through the use of traits. Researchers implement these traits to inject custom logic into the simulation while adhering to its rules. This approach ensures a consistent structure while allowing the researcher to extend the simulator's functionality.

Traits are similar to Sinalgo's inheritance as a means to provide extensibility. However, the simulator offers more flexibility through how the researchers can tailor the specification and initialisation of new simulations.

To create a new simulation, researchers must assemble a struct that implements the `Scenario` trait. While the simulator abstracts many of its features to simplify the development, it also allows researchers to customise scenarios by modifying or omitting certain functionalities and adding new ones.

Additionally, the simulator provides *macros* to reduce boilerplate code, leading to quicker development. However, abstracting the simulator's behaviour may alienate researchers from understanding how the simulator's internal mechanisms work. The simulator aims to reduce the amount of logic the researchers need to know to create a simulation, not attempt to hide how it works internally.

Importantly, the simulator aims to minimise the amount of internal logic researchers need to comprehend in order to be able to create simulation scenarios rather than hiding its internal operation entirely.

Chapter 5

Experimental Evaluation

After studying the existing simulators and designing a new one, this chapter presents an experimental evaluation of the developed simulator.

The chapter is divided into two parts. The first part validates the developed simulator by comparing its functioning to an already-established simulator (Section 5.1).

The second part leverages the simulator to evaluate distributed statistical learning algorithms, showing how an algorithm is implemented and studied and what can be done after the evaluation (Section 5.2).

5.1 Simulator Validation

The designed simulator will be validated by comparing its functioning with SimGrid. Multiple experiments will be conducted to assess whether the two simulators share the same network simulation functionality and obtain the same results. The results of these experiments should show that the designed simulator can achieve the same results as the already-established simulator.

First, the simulator's basic message capabilities will be validated in a straightforward scenario. This scenario involves having a single peer responsible for sending messages to its neighbours. It shall assess if the communication delay is correctly calculated. The scenario consists of four experiments, which will be discussed in Section 5.1.1.2.

After validating that messages are sent correctly, the interaction between peers will be validated using a more complex scenario. The complex scenario involves implementing one of the Flow Updating [1] algorithms, particularly the pairwise asynchronous version. The implementation of this algorithm in both simulators will be described in Section 5.1.2.1 and Section 5.1.2.1 for the custom simulator and SimGrid, respectively.

The designed simulator does not attempt to emulate different types of communication protocols as does SimGrid with its network models. Therefore, the developed simulator will be compared with SimGrid using the `raw` network model.

5.1.1 Message Passing Scenario

As already mentioned, the message-passing scenario consists of having a single peer that sends a single message to its neighbours. A peer responsible for sending messages will be called *sender*, while a peer responsible for receiving a message will be named *receiver*.

5.1.1.1 Implementation

The implementations of the Message Passing scenario in both simulators can be consulted in Appendix B.1.1, particularly Appendix B.1.1.1 for the designed simulator and Appendix B.1.1.2 for SimGrid.

Designed Simulator The designed simulator implements this scenario by creating a new struct that implements the `Scenario` trait. This struct, whose implementation can be seen in Listing B.1, is named `SimpleMessageScenario`.

Registered as "simple_message", its `start` method begins by adding the peers to the simulation state. These peers are instances of the struct `SimplePeer` (whose definition can be seen in Listing B.2) and are given a position when being created by the simulator. The position is optional and provided by the configuration file. The default position (0, 0) is used if no value is found in the positions field. As can be seen in the listing of the definition of the peer struct, upon creation, the callback, which is called when the peer receives a message, is set to the method whose definition can be consulted in Listing B.3. This callback solely logs that the peer in question received a message.

After adding peers to the simulation, the scenario connects the peers using the topology provided by the configuration file. Then, it assigns the arrival time callback to the simulation, which is to be used to calculate the arrival time of messages if the configuration file provides the name of the callback.

Before being able to start the simulation, the scenario obtains the size of the messages passed by the configuration file using the `extra_args` option. In case the size of the message is not passed, it will default to 0. This value is then used for the start event, which consists of a timer named `StartTimer` that fires on timestamp 0.0. The timer in question (whose definition can be observed in Listing B.4) is provided with the size of the messages it should send. It sends the messages from the *sender* peer to the *receiver* peers. The message is a struct that implements the `Message` trait and can have its definition consulted in Listing B.5.

The designed simulator's implementation assigns the first peer as the *sender* and the rest of the peers as the *receivers*.

Finally, the scenario is set up, and the experiments can be executed.

SimGrid The SimGrid implementation is straightforward. It consists of only three parts, those being the `sender` method, the `receiver` method, and the *Python* program's entry point.

The program's entry point (whose definition can be seen in Listing B.6) is responsible for setting up the simulation environment. During this part, the methods `sender` and `receiver` are registered as the methods that contain the logic of the actors, *sender* and *receiver*, respectively.

The SimGrid implementation does not require a message object. Therefore, messages are a single string. The size that should be assigned to the messages being sent is provided by the arguments that the `sender` method (whose definition can be found in Listing B.7) receives. These arguments are part of the `actors` configuration file.

The `receiver` method (whose definition can be seen in Listing B.8) is only responsible for receiving messages and logging that it has received a message.

The SimGrid's implementation differentiates the *sender* and *receiver* roles by assigning their respective method using the `actors` configuration file.

Having implemented the three parts, the experiments can be executed.

5.1.1.2 Experiments

Four experiments will be put into practice and then analysed to validate the simulator's behaviour. The four experiments will compare how the network communication characteristics of the two simulators compare. The developed simulator is expected to obtain the same practical results as the already-established simulator.

Latency: Empty Message Exchange The first experiment validates whether a message takes the same time from when it is sent by the *sender* to when it is received by the *receiver*, not considering the connections' bandwidth. The topology used for this experiment can be seen in Figure 5.1. It consists of two nodes connected by a bidirectional link. The link's latency implies that each message transmitted through the link experiences a delay of 40 milliseconds in both directions. The simulators will not consider the link's bandwidth because the messages that will be sent will have a size of 0.

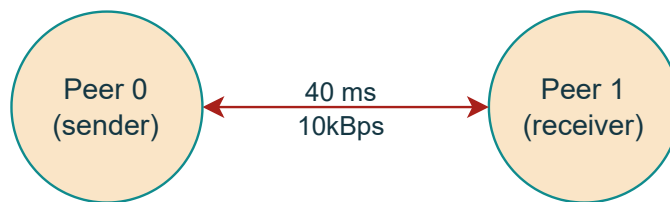


Figure 5.1: Two peers connected by a bidirectional link

Designed Simulator's Configuration Considering the scenario implementation previously explained for the developed simulator, the experiment configuration must specify the topology shown in Figure 5.1 and the parameters required to run the simulation. The resulting configuration file can be seen in Listing 5.1. In this listing, it can be observed that the experiment has been named "empty_message", it uses the "simple_message" scenario, has a topology that represents the one seen in the figure and includes an extra argument that informs the simulation of the message size

to be used, in this case, 0. Note that the bandwidth value in the developed simulator is in bits, and the latency value is in seconds.

```

1 - name: empty_message
2   scenario: simple_message
3   topology:
4     n_peers: 2
5     name: twowaycustom
6     connections:
7       - [0, 1, !Full {bandwidth: 80000, latency: 0.04} ]
8   extra_args:
9     msg_size: 0

```

Listing 5.1: The designed simulator's first experiment configuration

SimGrid's Configuration For the SimGrid simulator scenario implementation, two configuration files are required: the `platform` file and the `actors` file. Listing 5.2 presents the `platform` configuration file. It is responsible for informing the simulator of the hosts, the links, and the routes the simulation includes. Therefore, it represents the topology seen in Figure 5.1 and, as such, consists of two hosts (`Peer_0` and `Peer_1`), a single link with 10 kilobytes (kB) per second bandwidth and a latency of 40 milliseconds, and a single route that connects the two hosts using the link. Listing 5.3 illustrates the `actors` configuration file. It is responsible for informing the simulator of what actors it must instantiate. In this case, there are two actors, the *sender* and the *receiver*, which belong to the actors `Peer_0` and `Peer_1`, respectively.

```

1 <?xml version='1.0'?>
2 <!DOCTYPE platform SYSTEM "https://simgrid.org/simgrid.dtd">
3 <platform version="4.1">
4   <zone id="zone0" routing="Full">
5     <host id="Peer_0" speed="1Gf"/>
6     <host id="Peer_1" speed="1Gf"/>
7
8     <link id="0" bandwidth="10kBps" latency="40ms"/>
9
10    <route src="Peer_0" dst="Peer_1">
11      <link_ctn id="0"/>
12    </route>
13  </zone>
14 </platform>

```

Listing 5.2: SimGrids's first experiment platform configuration

```

1 <?xml version='1.0'?>
2 <!DOCTYPE platform SYSTEM "https://simgrid.org/simgrid.dtd">
3 <platform version="4.1">

```

```

4  <actor host="Peer_0" function="sender">
5    <argument value="Peer_1"/>
6    <argument value="0" />
7  </actor>
8  <actor host="Peer_1" function="receiver">
9    <argument value="1"/>
10 </actor>
11 </platform>

```

Listing 5.3: SimGrids's first experiment actors configuration

Results Executing the experiment in both simulators results in the logs shown in Listing 5.4 and Listing 5.5, the first being from the designed simulator and the second from SimGrid.

```

[0] [SIMUL] SIMULATION LOOP STARTED
[0] [INFO] Peer 0 sent message to 1
[0.04] [INFO] Peer 1 received a message
[0.04] [SIMUL] SIMULATION LOOP FINISHED

```

Listing 5.4: The designed simulator's first experiment log output

```

[Peer_0:sender:(1) 0.000000] [python/INFO] Sender started.
[Peer_0:sender:(1) 0.000000] [python/INFO] Sending message to Mailbox(Peer_1): Message
content
[Peer_1:receiver:(2) 0.000000] [python/INFO] Receiver started with Mailbox(Peer_1)
[Peer_1:receiver:(2) 0.040000] [python/INFO] Received: Message content
[0.040000] [python/INFO] Simulation finished

```

Listing 5.5: SimGrid's first experiment log output

As can be concluded, both simulators sent the message at timestamp 0.0, and it arrived at timestamp 0.04, which corresponds to the 40 milliseconds that were provided to the link that connects the two peers.

Bandwidth: 100 Bytes Message Exchange Having concluded that considering only the latency, sending messages obtains the same result in both simulators, bandwidth will be assessed by introducing it into the experiment. Therefore, the second experiment is similar to the first, but the message size is 100 bytes.

The topology used will be the same (presented in Figure 5.1). But in this experiment, considering that the link has a bandwidth value of 10 kilobytes per second and that the message being sent has a size of 100 bytes, the message should take an extra 10 milliseconds. The extra 10 milliseconds are due to how both simulators calculate the time for a message to arrive. Both simulators calculate it based on the formula shown in Equation 5.1. Note that SimGrid calculates

it using this equation when using the `raw` network model¹. Using other network models leads the simulator to calculate the time for a message to arrive differently.

$$latency + \frac{size}{bandwidth} \quad (5.1)$$

Designed Simulator's Configuration This experiment's configuration file only differs from the one presented in Listing 5.1 in its name and the extra argument provided to the simulation. In this case, as can be seen in Listing 5.6, it is called `100_bytes_message` and the `msg_size` field is 100 instead of 0.

```

1 - name: 100_bytes_message
2   scenario: simple_message
3   topology:
4     n_peers: 2
5     name: twowaycustom
6     connections:
7       - [0, 1, !Full {bandwidth: 80000, latency: 0.04} ]
8   extra_args:
9     msg_size: 100

```

Listing 5.6: The designed simulator's second experiment configuration

SimGrid's Configuration Similarly to the developed simulator, this experiment's configuration is similar to the first experiment's. In SimGrid's case, the only configuration file that needs to be changed is the `actors` file. As can be seen in Listing 5.7, the second argument from actor `Peer_0` has a value of 100, instead of 0 presented in Listing 5.3.

```

1 <?xml version='1.0'?>
2 <!DOCTYPE platform SYSTEM "https://simgrid.org/simgrid.dtd">
3 <platform version="4.1">
4   <actor host="Peer_0" function="sender">
5     <argument value="Peer_1"/>
6     <argument value="100" />
7   </actor>
8   <actor host="Peer_1" function="receiver">
9     <argument value="1"/>
10  </actor>
11 </platform>

```

Listing 5.7: SimGrid's second experiment actors configuration

Results Executing the experiment in both simulators results in the logs shown in Listing 5.8 and Listing 5.9, the first being from the designed simulator and the second from SimGrid.

¹<https://simgrid.org/doc/latest/Models.html#raw-network-model>

```
[0] [SIMUL] SIMULATION LOOP STARTED
[0] [INFO] Peer 0 sent message to 1
[0.05] [INFO] Peer 1 received a message
[0.05] [SIMUL] SIMULATION LOOP FINISHED
```

Listing 5.8: The designed simulator's second experiment log output

```
[Peer_0:sender:(1) 0.000000] [python/INFO] Sender started.
[Peer_0:sender:(1) 0.000000] [python/INFO] Sending message to Mailbox(Peer_1): Message
content
[Peer_1:receiver:(2) 0.000000] [python/INFO] Receiver started with Mailbox(Peer_1)
[Peer_1:receiver:(2) 0.050000] [python/INFO] Received: Message content
[0.050000] [python/INFO] Simulation finished
```

Listing 5.9: SimGrid's second experiment log output

As shown in the logs, both simulators sent the message at timestamp 0.0, and it arrived at timestamp 0.05. Considering that the latency is 40 milliseconds, the resulting 50 milliseconds correspond to the latency added to the expected extra 10 milliseconds. In conclusion, the designed simulator handles bandwidth as the already-established simulator does.

Topology: Star Topology The third experiment introduces multiple peers into the simulation in a star topology. Each of the links that connect the peers will have different values of bandwidth and latency. The topology for this experiment can be seen in Figure 5.2.

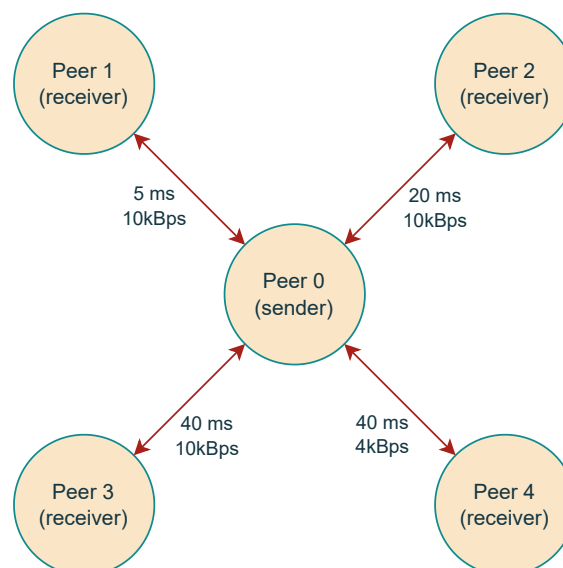


Figure 5.2: Five peers connected by bidirectional links in a star topology

Designed Simulator's Configuration This experiment's configuration file introduces five peers and four connections in order to describe the topology in Figure 5.2. It can be observed in Listing 5.10. Every peer is connected to the first, and each of these connections has different values for the bandwidth and latency. Since the bandwidth is considered, the size of the messages is set using the `extra_args` field to 100.

```

1 - name: star_topology
2   scenario: simple_message
3   arrival_time: distance
4   topology:
5     n_peers: 5
6     name: twowaycustom
7     connections:
8       - [0, 1, !Full {bandwidth: 80000, latency: 0.005} ]
9       - [0, 2, !Full {bandwidth: 80000, latency: 0.02} ]
10      - [0, 3, !Full {bandwidth: 80000, latency: 0.04} ]
11      - [0, 4, !Full {bandwidth: 32000, latency: 0.04} ]
12   extra_args:
13     msg_size: 100

```

Listing 5.10: The designed simulator's third experiment configuration

SimGrid's Configuration Because the number of peers has increased, compared to the previous scenarios, the platform configuration file must be changed to append the new hosts, links and routes. As a result of this change, the configuration found in Listing 5.11 is created. It presents five hosts, four links and four routes. The first three links have a fixed bandwidth of 10 kilobytes per second, and the last has a bandwidth of 4 kilobytes. The first link has a latency of 5 milliseconds, the second has a latency of 20 milliseconds, and the last two have a latency of 40 milliseconds.

```

1 <?xml version='1.0'?>
2 <!DOCTYPE platform SYSTEM "https://simgrid.org/simgrid.dtd">
3 <platform version="4.1">
4   <zone id="zone0" routing="Full">
5     <host id="Peer_0" speed="1Gf"/>
6     <host id="Peer_1" speed="1Gf"/>
7     <host id="Peer_2" speed="1Gf"/>
8     <host id="Peer_3" speed="1Gf"/>
9     <host id="Peer_4" speed="1Gf"/>
10
11     <link id="0" bandwidth="10kBps" latency="5ms"/>
12     <link id="1" bandwidth="10kBps" latency="20ms"/>
13     <link id="2" bandwidth="10kBps" latency="40ms"/>
14     <link id="3" bandwidth="4kBps" latency="40ms"/>
15
16     <route src="Peer_0" dst="Peer_1"><link_ctn id="0"/></route>
17     <route src="Peer_0" dst="Peer_2"><link_ctn id="1"/></route>
18     <route src="Peer_0" dst="Peer_3"><link_ctn id="2"/></route>

```

```

19     <route src="Peer_0" dst="Peer_4"><link_ctn id="3"/></route>
20   </zone>
21 </platform>

```

Listing 5.11: SimGrids's third experiment actors configuration

As a consequence of updating the platform configuration file, the actors file must also be changed. The new hosts will be assigned to new *receiver* actors, and the *sender* actor will update its list of peers to include all four other peers. The resulting configuration file is shown in Listing 5.12.

```

1 <?xml version='1.0'?>
2 <!DOCTYPE platform SYSTEM "https://simgrid.org/simgrid.dtd">
3 <platform version="4.1">
4   <actor host="Peer_0" function="sender">
5     <argument value="Peer_1,Peer_2,Peer_3,Peer_4"/>
6     <argument value="100" />
7   </actor>
8   <actor host="Peer_1" function="receiver"></actor>
9   <actor host="Peer_2" function="receiver"></actor>
10  <actor host="Peer_3" function="receiver"></actor>
11  <actor host="Peer_4" function="receiver"></actor>
12 </platform>

```

Listing 5.12: SimGrids's third experiment actors configuration

Results Executing the experiment in both simulators results in the logs shown in Listing 5.13 and Listing 5.14, the first being from the designed simulator and the second from SimGrid.

```

[0] [SIMUL] SIMULATION LOOP STARTED
[0] [INFO] Peer 0 sent message to 1
[0] [INFO] Peer 0 sent message to 2
[0] [INFO] Peer 0 sent message to 3
[0] [INFO] Peer 0 sent message to 4
[0.015] [INFO] Peer 1 received a message
[0.03] [INFO] Peer 2 received a message
[0.05] [INFO] Peer 3 received a message
[0.065] [INFO] Peer 4 received a message
[0.065] [SIMUL] SIMULATION LOOP FINISHED

```

Listing 5.13: The designed simulator's third experiment log output

```

[Peer_0:sender:(1) 0.000000] [python/INFO] Sender started.
[Peer_0:sender:(1) 0.000000] [python/INFO] Sending message to Mailbox(Peer_1): Message
content
[Peer_1:receiver:(2) 0.000000] [python/INFO] Receiver started with Mailbox(Peer_1)

```

```
[Peer_2:receiver:(3) 0.000000] [python/INFO] Receiver started with Mailbox(Peer_2)
[Peer_3:receiver:(4) 0.000000] [python/INFO] Receiver started with Mailbox(Peer_3)
[Peer_4:receiver:(5) 0.000000] [python/INFO] Receiver started with Mailbox(Peer_4)
[Peer_0:sender:(1) 0.000000] [python/INFO] Sending message to Mailbox(Peer_2): Message
content
[Peer_0:sender:(1) 0.000000] [python/INFO] Sending message to Mailbox(Peer_3): Message
content
[Peer_0:sender:(1) 0.000000] [python/INFO] Sending message to Mailbox(Peer_4): Message
content
[Peer_1:receiver:(2) 0.015000] [python/INFO] Received: Message content
[Peer_2:receiver:(3) 0.030000] [python/INFO] Received: Message content
[Peer_3:receiver:(4) 0.050000] [python/INFO] Received: Message content
[Peer_4:receiver:(5) 0.065000] [python/INFO] Received: Message content
[0.065000] [python/INFO] Simulation finished
```

Listing 5.14: SimGrid's third experiment log output

Both simulators output the same behaviour, which, calculating the expected arrival time for each message, corresponds to the values shown in the logs. Therefore, it can be concluded that sending multiple messages in topology to different peers and in various conditions exhibits the same behaviour in the developed and already-established simulators.

Distance Based Latency The fourth (and last) experiment differs from the first two because it does not compare the designed simulator's functionality with its corresponding functionality in SimGrid. The developed simulator allows the researcher to assign a spatial position to each peer and to use those positions to calculate the latency of the links that connect them. SimGrid does not offer that functionality. Therefore, pre-defined latency values will be used to compare. The experiment's topology will be planned as before and adapted to the designed simulator to compare the two.

This experiment's topology is presented in Figure 5.3. The figure shows that four peers must exist, one *sender* and three *receivers*. Every link has the same bandwidth of 10 kilobytes per second and the same latency of 40 milliseconds, except for one link, which has a latency of 56.57 milliseconds. Because the focus of this experiment is the latency, the bandwidth will, once again, not be considered.

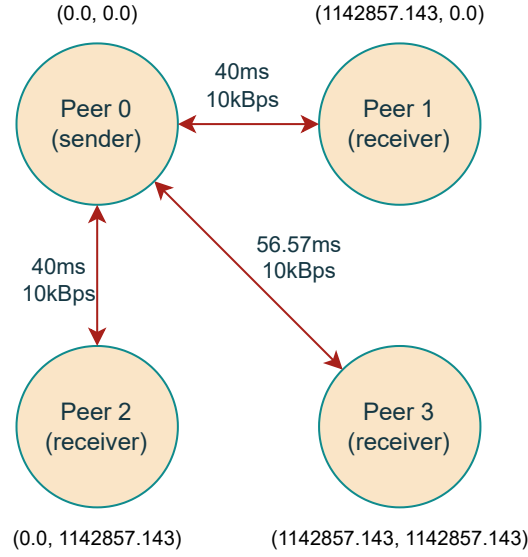


Figure 5.3: Four peers connected by bidirectional links in a star topology

In the topology figure, above the nodes on top and below the nodes on the bottom, the coordinates of the nodes can be found. These were calculated based on how the designed simulator calculates the distance between each peer when using the built-in arrival time callback named "distance" (whose implementation was already mentioned in Section 4.9.3 and can be found in Listing 4.19). The arrival time callback calculates the latency by multiplying a coefficient that represents the optical fiber delay with the distance between the two peers, as can be seen in Equation 5.2.

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} \times c \quad (5.2)$$

where $c = 0.350 \times 10^{-6}$ is the optical fiber delay coefficient in seconds per meter, and the coordinates are in meters.

The simulators are expected to handle multiple events with the same timestamp. Therefore, two message delivery events should take 40 milliseconds, and a third one should take 56.57 milliseconds. The messages that arrive at the same timestamp should be in the same order as they were sent. Minor numerical inaccuracies are expected since the simulator computes latency from spatial data using floating-point arithmetic. Nevertheless, the result should remain approximately correct.

Designed Simulator's Configuration This experiment's configuration file must describe the topology shown in Figure 5.3 and the parameters required to run the simulation. The resulting configuration file can be inspected in Listing 5.15. For this experiment, the configuration file includes four peers whose positions have been declared using the `positions` field. Since the third coordinate is optional, and this simulation does not consider it, the value in the configuration

file is the ~ symbol, which the parser interprets as an unspecified value. These coordinates will be used in the arrival time callback, which the configuration file indicates to the simulator using the `arrival_time` field and the "distance" value. The bandwidth values are provided, but the extra argument declared means they will not affect the simulation's outcome.

```

1 - name: distance_topology
2   scenario: simple_message
3   arrival_time: distance
4   topology:
5     n_peers: 4
6     name: twowaycustom
7     connections:
8       - [0, 1, !Bandwidth 80000]
9       - [0, 2, !Bandwidth 80000]
10      - [0, 3, !Bandwidth 80000]
11     positions:
12       - [0.0, 0.0, ~]
13       - [1142857.143, 0.0, ~]
14       - [0.0, 1142857.143, ~]
15       - [1142857.143, 1142857.143, ~]
16   extra_args:
17     msg_size: 0

```

Listing 5.15: The designed simulator's fourth experiment configuration

SimGrid's Configuration There are no significant changes to make to SimGrid's configuration files to simulate this scenario since it will not calculate latency based on the distance.

The `platform` configuration file describes the topology in Figure 5.3. As seen in Listing 5.16, four hosts, three links and three routes are declared. Two of the links have a latency of 40 milliseconds, and the other one has a latency of 56.57 milliseconds. As the figure shows, the routes connect every peer to the first.

```

1 <?xml version='1.0'?>
2 <!DOCTYPE platform SYSTEM "https://simgrid.org/simgrid.dtd">
3 <platform version="4.1">
4   <zone id="zone0" routing="Full">
5     <host id="Peer_0" speed="1Gf"/>
6     <host id="Peer_1" speed="1Gf"/>
7     <host id="Peer_2" speed="1Gf"/>
8     <host id="Peer_3" speed="1Gf"/>
9
10    <link id="0" bandwidth="10kBps" latency="40ms"/>
11    <link id="1" bandwidth="10kBps" latency="40ms"/>
12    <link id="2" bandwidth="10kBps" latency="56.57ms"/>
13
14    <route src="Peer_0" dst="Peer_1"><link_ctn id="0"/></route>
15    <route src="Peer_0" dst="Peer_2"><link_ctn id="1"/></route>
16    <route src="Peer_0" dst="Peer_3"><link_ctn id="2"/></route>
17  </zone>

```

```
18 </platform>
```

Listing 5.16: SimGrids's fourth experiment platform configuration

The `actors` configuration file complements the `platform` file by indicating which roles each host will have. As seen in Listing 5.17, the first actor is a *sender* while the remaining are *receivers*. The *sender* actor is provided with the identifiers of the other peers and a value of 0 for the message size.

```
1 <?xml version='1.0'?>
2 <!DOCTYPE platform SYSTEM "https://simgrid.org/simgrid.dtd">
3 <platform version="4.1">
4   <actor host="Peer_0" function="sender">
5     <argument value="Peer_1,Peer_2,Peer_3"/>
6     <argument value="0" />
7   </actor>
8   <actor host="Peer_1" function="receiver"></actor>
9   <actor host="Peer_2" function="receiver"></actor>
10  <actor host="Peer_3" function="receiver"></actor>
11 </platform>
```

Listing 5.17: SimGrids's fourth experiment actors configuration

Results Executing the experiment in both simulators results in the logs shown in Listing 5.18 and Listing 5.19, the first being from the designed simulator and the second from SimGrid.

```
[0] [SIMUL] SIMULATION LOOP STARTED
[0] [INFO] Peer 0 sent message to 1
[0] [INFO] Peer 0 sent message to 2
[0] [INFO] Peer 0 sent message to 3
[0.40000000004999997] [INFO] Peer 1 received a message
[0.40000000004999997] [INFO] Peer 2 received a message
[0.5656854250199487] [INFO] Peer 3 received a message
[0.5656854250199487] [SIMUL] SIMULATION LOOP FINISHED
```

Listing 5.18: The designed simulator's fourth experiment log output

```
[Peer_0:sender:(1) 0.000000] [python/INFO] Sender started.
[Peer_0:sender:(1) 0.000000] [python/INFO] Sending message to Mailbox(Peer_1): Message
content
[Peer_1:receiver:(2) 0.000000] [python/INFO] Receiver started with Mailbox(Peer_1)
[Peer_2:receiver:(3) 0.000000] [python/INFO] Receiver started with Mailbox(Peer_2)
[Peer_3:receiver:(4) 0.000000] [python/INFO] Receiver started with Mailbox(Peer_3)
[Peer_0:sender:(1) 0.000000] [python/INFO] Sending message to Mailbox(Peer_2): Message
content
```

```
[Peer_0:sender:(1) 0.000000] [python/INFO] Sending message to Mailbox(Peer_3): Message
content
[Peer_1:receiver:(2) 0.040000] [python/INFO] Received: Message content
[Peer_2:receiver:(3) 0.040000] [python/INFO] Received: Message content
[Peer_3:receiver:(4) 0.056570] [python/INFO] Received: Message content
[0.056570] [python/INFO] Simulation finished
```

Listing 5.19: SimGrid's fourth experiment log output

As can be concluded from the logs, both simulators show the same behaviour. The messages expected to arrive at the same timestamp are received at the expected timestamp. Furthermore, both simulators process the message delivery events in the same order in which the events were created. The message that was supposed to arrive later than the others does, in fact, correctly arrive later. Therefore, the designed simulator exhibits the same behaviour as SimGrid when using the arrival time callback instead of a pre-defined latency number. Using an arrival time callback instead of a pre-defined latency value is helpful in easily generating new topologies and simulating dynamic latency calculations, for example, if the peers moved during the simulation.

5.1.2 Flow Updating Scenario

Flow Updating is an algorithm that allows multiple peers, each with a value, to communicate between themselves to obtain the average of their values. They do this without needing to share their value or store the values of every other peer in the network. The pairwise version is a variant of this algorithm that is around 30% faster but demands that from 3 to more 10 times as many messages are sent [1]. It was chosen to validate the developed simulator because it is a more message-intensive and complex algorithm with concurrent communication between multiple peers.

5.1.2.1 Implementation

The following two sections explain how the scenario was implemented in both simulators. The code definitions mentioned in the implementation sections are documented in Appendix B.1.2, particularly Appendix B.1.2.1 for the designed simulator and Appendix B.1.2.2 for SimGrid.

Designed Simulator The designed simulator implements this scenario by creating a new struct that implements the `Scenario` trait. This struct, whose implementation can be seen in Listing B.9, is named `FlowUpdatingPairwise`.

Registered in the simulator as "flow_updating", its `start` method begins by gathering the values for the peers that can be provided in the `extra_args` field with key values. These values are stored in a vector of numbers of type `i32`, and the size is irrelevant. It is irrelevant because, in the next step, the simulator creates every peer, assigning it a value. It retrieves the value to be used from the list of values provided. However, if no value is provided, the simulator generates a random number between 0 and 80. This will not be used during the experiment, but one could provide fewer values than the number of peers and have the simulator generate the

remaining values. Not only do peers require that the value is provided when being created, but they also require a position. A custom topology will be used during the experiment. Therefore, this part can be ignored as it is irrelevant. However, one could use an arrival time callback such as "distance" and generate simulations with different latencies between peers.

The peer in this scenario is a struct called `FlowUpdatingPairwisePeer` whose definition can be found in Listing B.10. It contains five fields, not counting with `peer_info`. Those fields are `value`, which corresponds to the integer value of the peer, `flows` which is a `HashMap` that corresponds to the flows of the neighbouring peers, `estimates` which is a `HashMap` that corresponds to the estimates of the neighbouring peers, `ticks_since_last_avg` which is a `HashMap` that corresponds to the number of ticks since the peer last performed the averaging operation, and lastly, `last_avg` which is a floating point number that corresponds to the previous value obtained after performing the averaging operation. Apart from `last_avg`, these fields are essential aspects of the algorithm and, therefore, are required to perform the calculations. Whenever a new peer of this type is created, the message received callback is set to the `on_message_receive` function. This function, whose definition can be consulted in Listing B.11, checks whether the message received is of type `FlowUpdatingPairwiseMessage` and if it is, updates the receiving peer's `estimate` and `flows` fields with the values (the flow value is negated) from the message received using the sender's identifier as key. After updating the peer's fields, it calls the `avg_and_send` function.

The `FlowUpdatingPairwiseMessage`, whose definition is shown in Listing B.12, is the struct that is used for every message in the scenario, it contains three fields: the sender's identifier (`sender`) a floating-point number representing the flow value (`flow`) and a floating-point number representing the estimated value (`estimate`).

The `avg_and_send` function, whose definition can be consulted in Listing B.13, receives two identifiers as parameters. The first points to the peer on which the calculations will be performed, and the second points to the neighbour to which the resulting message will be sent. In case the peers are, in fact, neighbours (one of them could be dead or not connected), the peer performing the calculations will calculate its new average value and use it to update the neighbouring peer's flow and estimate values, storing them in its fields `flow` and `estimate` respectively. After having calculated the values, the peer updates its `last_avg` field with the new average, sets the `ticks_since_last_avg` field's value for the neighbouring peer to 0, and sends a new `FlowUpdatingPairwiseMessage` to the target peer with its identifier as field `sender`, the flow calculated for the target peer as field `flow` and its new average as field `estimate`.

After instantiating every peer, the scenario connects the peers using the topology provided by the configuration file. Then, it assigns the arrival time callback to the simulation, which is to be used to calculate the arrival time of messages if the configuration file provides the name of the callback.

Before adding timers to the simulation, the scenario sets up the simulation hooks. Only the hook, called when the simulation loop finishes, is assigned a different function. This function, whose definition can be found in Listing B.14, retrieves the initial values and last averages from

every peer, calculating how far off it is from the actual average value. It logs the average of the values obtained by the simulation, the minimum last average, and the maximum.

Having added the hook to the simulation, the scenario introduces two timers required for the algorithm's logic and one timer for metric collection.

The first timer is called `StarTimer`, and its definition can be found in Listing B.15. This timer receives the identifier of a peer so that, when fired, it can obtain the value of the peer in question and broadcast it to its neighbours using the `FlowUpdatingPairwiseMessage` message struct.

The second timer is called `TickTimer`, and its definition can be found in Listing B.16. This timer receives an interval value set to 0.001 (1 millisecond). This is so that whenever it is fired, it can add itself to the simulation again using the interval time as an offset. Because it keeps adding itself to the simulation, the simulation loop will never reach the point at which it stops because there are no more events to process. Therefore, this simulation scenario must include a finish condition hook (which it does not) or a deadline. When fired, this timer calls the `tick` function for every peer belonging to the simulation. The `tick` function, whose definition can be observed in Listing B.17, checks, for the peer in question, whether it has communicated with its neighbours and, for each neighbour which the peer has not communicated for a certain threshold number of ticks, the peer calls the `avg_and_send` function. Whether a peer has communicated with its peers is possible to inspect using the peer's `ticks_since_last_avg` field, which includes a number of ticks for each neighbour. The threshold number of ticks is set as 50.

The third timer is called `MetricsTimer`, and its definition can be found in Listing B.18. Like the `TickTimer` timer, it also requires an interval value. In this case, it was also set to 0.001. This timer does not influence the simulation's outcome. Its purpose is solely to inspect the simulation state. When fired, it retrieves the `last_avg` value from every peer the simulation involves. It logs that value with the title "last_averages" using the developed simulator's `metrics` log method, which writes the values, along with the title and the timestamp at which it is called.

After introducing the timers into the simulation, the scenario calls the `run` method from the simulator's `engine` module, effectively initiating the simulation loop (and the simulation).

SimGrid SimGrid's implementation of this scenario, although replicating the same behaviour, requires a few adjustments to adapt to how the simulator functions. For example, there are no timers in SimGrid. Therefore, alternatives must be used to implement the tick and metrics timer's logic.

The *Python* program's entry point, which is shown in Listing B.19, starts by initializing the simulator engine, then it must receive two arguments from the system, those being the name of the scenario (`scenario`) and the deadline (`should_stop`). Another variable, which is called `metrics_interval`, is also declared. This is the same variable that was used in the designed simulator's implementation. However, the entry point sets up the simulation's environment before using that value.

The class `Peer`, whose declaration can be seen in Listing B.20, is registered to the name "peer". Both the `platform` and `actors` configuration files are in files whose name corresponds

to the scenario's name. Therefore, the `scenario` variable is used for loading the configuration files. The `actors` configuration indicates which actors are to be added to the simulation and which hosts they belong to. Therefore, indicating that an actor has function value "peer" means that an actor will be created using the `Peer` class. This class's constructor requires two arguments that must be provided in the `actor` configuration file. These arguments are the list of neighbours and the value of the peer. The constructor is then responsible for initializing the other internal variables that the simulation requires for the peer.

After registering the `Peer` class and loading the configuration files into the simulation, the entry point opens a new file for JSON formatted logs. Afterwards, the simulation creates a new host called "Observer", assigning it a new actor called "watcher". This actor corresponds to a method called `watcher` that receives two arguments, those being the previously declared variables: `should_stop` and `metrics_interval`.

The `watcher` method contains the logic responsible for inspecting the simulation state. Its declaration can be consulted in Listing B.21. The actor that executes this method's logic will output to the log file every millisecond until the deadline is reached. To log the simulation state into the log file, it uses a function (also shown in Listing B.21) called `print_global_values` which uses the global variables `log_file` and `global_values` to check whether the simulation has opened the log file and, if so, outputs a JSON object of the data available in the `global_values` dictionary. The moment that the `watcher` actor reaches the deadline, it terminates all other actors by executing `Actor.kill_all()`.

When the `watcher` actor finishes being instantiated, the simulation engine executes the simulation.

Upon starting the simulation, every actor added to it starts their logic. Methods start executing the logic they contain. However, classes, on the other hand, get their `__call__` method called. This method is responsible for initializing the tick loop and the message loop. Since the tick timer would be executed once every millisecond to call a method concerned with only one peer at a time, the tick function can be introduced to each peer's internal logic. This is done by creating another actor that executes the peer's `Peer` class method `tick_loop`. This loop executes another method called `tick` every millisecond, effectively replicating the tick timer. The actor executing the class's logic executes the `message_loop`, a loop responsible for checking if new messages of type `FlowUpdatingPairwiseMessage` arrive. The remaining methods of the class will not be explained as they contain the same logic that the designed simulator's corresponding methods contain, translated to *Python*.

The `FlowUpdatingPairwiseMessage`, whose implementation can be consulted in Listing B.22, contains three fields and a method, similar to the designed simulator's implementation of the message.

It is worth noting that, at the time of this implementation, the bindings for *Python* of the SimGrid simulator were different, and therefore, an `ActivitySet` is used to append every sent message communication. Otherwise, a segmentation fault error would occur.

This scenario's implementation uses the methods `get_async` and `put_async` to send and wait for messages without blocking the actor's progress.

The two distinct actors' purpose is to behave like two threads are being executed in the host. One is for receiving data, and the other is for tick logic.

5.1.2.2 Experiment Configuration

This scenario will be executed with the conditions shown in Figure 5.4. The figure shows seven peers that are fully connected and have pre-defined values. The latency and bandwidth values are presented on the label. The peers are in a circle, so their latency values reflect that. The label shows that the blue-coloured connections, such as the one between `Peer_0` and `Peer_1`, are the closest and, therefore, have a latency of 20 milliseconds. The orange-coloured connections are the farthest and thus have a latency of 47.68 milliseconds, and the black-coloured connections are in between and therefore have a latency of 36.04 milliseconds. The label also shows that every connection has a bandwidth of 10 kilobytes per second.

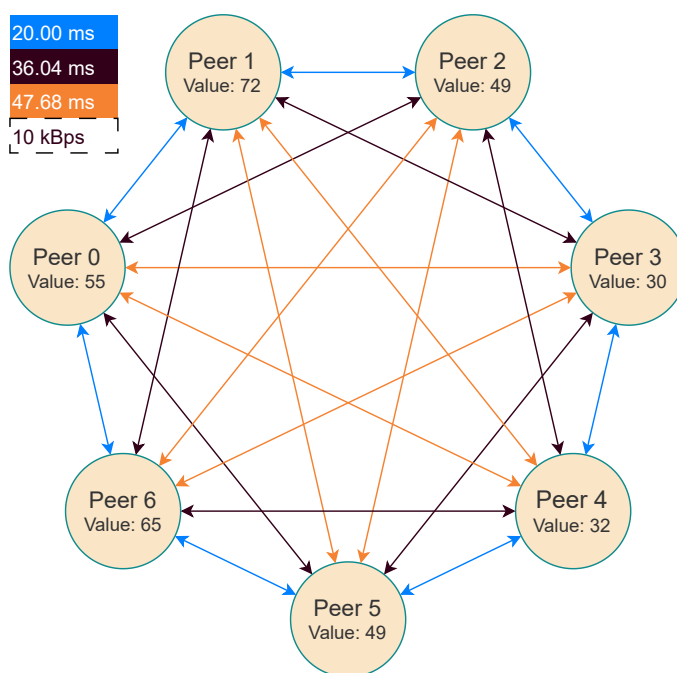


Figure 5.4: Fully connected experiment topology with values

Designed Simulator Given the scenario implementation described and the topology shown in Figure 5.4, the designed simulator will execute the simulation using a configuration file that contains the experiment's specification. The experiment configuration can be observed in Listing 5.20.

In this listing, it can be observed that the experiment has been named "flow_updating", it uses the "flow_updating_pairwise" scenario, has a topology that represents the one seen in the figure, has a deadline set to 1.75 seconds and includes an extra argument that provides the values to be assigned to the peers.

```

1 - name: flow_updating
2   scenario: flow_updating_pairwise
3   topology:
4     n_peers: 7
5     name: twowaycustom
6     connections:
7       - [0, 1, !Full {bandwidth: 80000, latency: 0.02} ]
8       - [0, 2, !Full {bandwidth: 80000, latency: 0.03604} ]
9       - [0, 3, !Full {bandwidth: 80000, latency: 0.04768} ]
10      - [0, 4, !Full {bandwidth: 80000, latency: 0.04768} ]
11      - [0, 5, !Full {bandwidth: 80000, latency: 0.03604} ]
12      - [0, 6, !Full {bandwidth: 80000, latency: 0.02} ]
13      - [1, 2, !Full {bandwidth: 80000, latency: 0.02} ]
14      - [1, 3, !Full {bandwidth: 80000, latency: 0.03604} ]
15      - [1, 4, !Full {bandwidth: 80000, latency: 0.04768} ]
16      - [1, 5, !Full {bandwidth: 80000, latency: 0.04768} ]
17      - [1, 6, !Full {bandwidth: 80000, latency: 0.03604} ]
18      - [2, 3, !Full {bandwidth: 80000, latency: 0.02} ]
19      - [2, 4, !Full {bandwidth: 80000, latency: 0.03604} ]
20      - [2, 5, !Full {bandwidth: 80000, latency: 0.04768} ]
21      - [2, 6, !Full {bandwidth: 80000, latency: 0.04768} ]
22      - [3, 4, !Full {bandwidth: 80000, latency: 0.02} ]
23      - [3, 5, !Full {bandwidth: 80000, latency: 0.03604} ]
24      - [3, 6, !Full {bandwidth: 80000, latency: 0.04768} ]
25      - [4, 5, !Full {bandwidth: 80000, latency: 0.02} ]
26      - [4, 6, !Full {bandwidth: 80000, latency: 0.03604} ]
27      - [5, 6, !Full {bandwidth: 80000, latency: 0.02} ]
28   deadline: 1.75
29   extra_args:
30     values: [55, 72, 49, 30, 32, 49, 65]

```

Listing 5.20: The designed simulator's experiment configuration

SimGrid To execute the experiment in SimGrid, two configuration files must be created: the platform and the actors configuration files.

The platform file can be observed in Listing 5.21. It declares the hosts representing the peers and the links and routes representing the connections. Note that a property in the links has not yet been mentioned. That property is called `sharing_policy`. SimGrid, unlike the developed simulator, simulates the consequences of congestion. Therefore, when the link's capacity is full, it adds additional delay to the message delivery process. Because this is not simulated in the developed simulator, the `sharing_policy` property is assigned the `FATPIPE` value to indicate to the simulator congestion is not to be considered. This property was required in this experiment but not in the previous experiments because no peer was receiving more than one message at a time. Therefore, no message's delivery time was impacted by another message being also delivered to the same peer at coincidental time frames.

```

1 <?xml version='1.0'?>
2 <!DOCTYPE platform SYSTEM "https://simgrid.org/simgrid.dtd">

```

```

3 <platform version="4.1">
4   <zone id="zone0" routing="Full">
5     <host id="Peer_0" speed="1Gf"/>
6     <host id="Peer_1" speed="1Gf"/>
7     <host id="Peer_2" speed="1Gf"/>
8     <host id="Peer_3" speed="1Gf"/>
9     <host id="Peer_4" speed="1Gf"/>
10    <host id="Peer_5" speed="1Gf"/>
11    <host id="Peer_6" speed="1Gf"/>
12
13
14    <link id="0" bandwidth="10kBps" latency="20ms" sharing_policy="FATPIPE"/>
15    <link id="1" bandwidth="10kBps" latency="36.04ms" sharing_policy="FATPIPE"/>
16    <link id="2" bandwidth="10kBps" latency="47.68ms" sharing_policy="FATPIPE"/>
17    <link id="3" bandwidth="10kBps" latency="47.68ms" sharing_policy="FATPIPE"/>
18    <link id="4" bandwidth="10kBps" latency="36.04ms" sharing_policy="FATPIPE"/>
19    <link id="5" bandwidth="10kBps" latency="20ms" sharing_policy="FATPIPE"/>
20    <link id="6" bandwidth="10kBps" latency="20ms" sharing_policy="FATPIPE"/>
21    <link id="7" bandwidth="10kBps" latency="36.04ms" sharing_policy="FATPIPE"/>
22    <link id="8" bandwidth="10kBps" latency="47.68ms" sharing_policy="FATPIPE"/>
23    <link id="9" bandwidth="10kBps" latency="47.68ms" sharing_policy="FATPIPE"/>
24    <link id="10" bandwidth="10kBps" latency="36.04ms" sharing_policy="FATPIPE"/>
25    <link id="11" bandwidth="10kBps" latency="20ms" sharing_policy="FATPIPE"/>
26    <link id="12" bandwidth="10kBps" latency="36.04ms" sharing_policy="FATPIPE"/>
27    <link id="13" bandwidth="10kBps" latency="47.68ms" sharing_policy="FATPIPE"/>
28    <link id="14" bandwidth="10kBps" latency="47.68ms" sharing_policy="FATPIPE"/>
29    <link id="15" bandwidth="10kBps" latency="20ms" sharing_policy="FATPIPE"/>
30    <link id="16" bandwidth="10kBps" latency="36.04ms" sharing_policy="FATPIPE"/>
31    <link id="17" bandwidth="10kBps" latency="47.68ms" sharing_policy="FATPIPE"/>
32    <link id="18" bandwidth="10kBps" latency="20ms" sharing_policy="FATPIPE"/>
33    <link id="19" bandwidth="10kBps" latency="36.04ms" sharing_policy="FATPIPE"/>
34    <link id="20" bandwidth="10kBps" latency="20ms" sharing_policy="FATPIPE"/>
35
36
37    <route src="Peer_0" dst="Peer_1"><link_ctn id="0"/></route>
38    <route src="Peer_0" dst="Peer_2"><link_ctn id="1"/></route>
39    <route src="Peer_0" dst="Peer_3"><link_ctn id="2"/></route>
40    <route src="Peer_0" dst="Peer_4"><link_ctn id="3"/></route>
41    <route src="Peer_0" dst="Peer_5"><link_ctn id="4"/></route>
42    <route src="Peer_0" dst="Peer_6"><link_ctn id="5"/></route>
43    <route src="Peer_1" dst="Peer_2"><link_ctn id="6"/></route>
44    <route src="Peer_1" dst="Peer_3"><link_ctn id="7"/></route>
45    <route src="Peer_1" dst="Peer_4"><link_ctn id="8"/></route>
46    <route src="Peer_1" dst="Peer_5"><link_ctn id="9"/></route>
47    <route src="Peer_1" dst="Peer_6"><link_ctn id="10"/></route>
48    <route src="Peer_2" dst="Peer_3"><link_ctn id="11"/></route>
49    <route src="Peer_2" dst="Peer_4"><link_ctn id="12"/></route>
50    <route src="Peer_2" dst="Peer_5"><link_ctn id="13"/></route>
51    <route src="Peer_2" dst="Peer_6"><link_ctn id="14"/></route>
52    <route src="Peer_3" dst="Peer_4"><link_ctn id="15"/></route>
53    <route src="Peer_3" dst="Peer_5"><link_ctn id="16"/></route>
54    <route src="Peer_3" dst="Peer_6"><link_ctn id="17"/></route>
55    <route src="Peer_4" dst="Peer_5"><link_ctn id="18"/></route>
56    <route src="Peer_4" dst="Peer_6"><link_ctn id="19"/></route>
57    <route src="Peer_5" dst="Peer_6"><link_ctn id="20"/></route>
58  </zone>
59 </platform>

```

Listing 5.21: SimGrid's experiment platform configuration

The `actors` file can be consulted in Listing 5.22. It declares which roles each actor will have and which host they are assigned to and provides the arguments needed to initialize them. For this experiment, each of the actors will have their `function` property set to `"peer"`, which corresponds to the `Peer` class mentioned in the SimGrid implementation (its declaration can be inspected in Listing B.20). The class requires two arguments, the value of the peer and its neighbours, which are declared in the file in accordance with the topology in Figure 5.4.

```

1 <?xml version='1.0'?>
2 <!DOCTYPE platform SYSTEM "https://simgrid.org/simgrid.dtd">
3 <platform version="4.1">
4   <actor host="Peer_0" function="peer">
5     <argument value="55" />
6     <argument value="Peer_1,Peer_2,Peer_3,Peer_4,Peer_5,Peer_6"/>
7   </actor>
8   <actor host="Peer_1" function="peer">
9     <argument value="72" />
10    <argument value="Peer_0,Peer_2,Peer_3,Peer_4,Peer_5,Peer_6"/>
11  </actor>
12  <actor host="Peer_2" function="peer">
13    <argument value="49" />
14    <argument value="Peer_0,Peer_1,Peer_3,Peer_4,Peer_5,Peer_6"/>
15  </actor>
16  <actor host="Peer_3" function="peer">
17    <argument value="30" />
18    <argument value="Peer_0,Peer_1,Peer_2,Peer_4,Peer_5,Peer_6"/>
19  </actor>
20  <actor host="Peer_4" function="peer">
21    <argument value="32" />
22    <argument value="Peer_0,Peer_1,Peer_2,Peer_3,Peer_5,Peer_6"/>
23  </actor>
24  <actor host="Peer_5" function="peer">
25    <argument value="49" />
26    <argument value="Peer_0,Peer_1,Peer_2,Peer_3,Peer_4,Peer_6"/>
27  </actor>
28  <actor host="Peer_6" function="peer">
29    <argument value="65" />
30    <argument value="Peer_0,Peer_1,Peer_2,Peer_3,Peer_4,Peer_5"/>
31  </actor>
32 </platform>

```

Listing 5.22: SimGrid's experiment actors configuration

The SimGrid implementation does not retrieve the simulation deadline timestamp from any of the configuration files. Instead, it must be provided to the *Python* program through command-line arguments. The value 1.75 will be provided as a command line argument when executing this experiment.

5.1.2.3 Results

The experiment was executed in both simulators' implementations. As a result both simulators output their behaviour through textual logs and their values through the JSON logs. SimGrid's implementation does not store the textual logs in a file, therefore, the console output is stored in a file to be examined.

Note that, because there are no operations in both simulations that require the generation of random numbers, there is no need to specify a seed.

This scenario's textual logs show that the simulators handle the events differently. SimGrid's *actors* yield whenever they send a new message, therefore, when starting, the first peer sends a message and yields, leading to another peer sending one of its messages before the first sends its second. The designed simulator however, has the first peer sending all the initial messages before the second is able to do so. Therefore, the order in which the events are processed differs between the two simulators. Listing 5.23 and Listing 5.24 show the first nine text logs about the messages that are sent when the simulation starts for the designed simulator and SimGrid respectively. As can be observed in the listings, in the designed simulator, *Peer_0* sends all its messages before *Peer_1* starts sending its own, while in SimGrid, each peer sends a message, so by the time *Peer_0* sends its second message, every other peer has already sent one message.

```
[0] [INFO] Peer_0 sending to Peer_1
[0] [INFO] Peer_0 sending to Peer_2
[0] [INFO] Peer_0 sending to Peer_3
[0] [INFO] Peer_0 sending to Peer_4
[0] [INFO] Peer_0 sending to Peer_5
[0] [INFO] Peer_0 sending to Peer_6
[0] [INFO] Peer_1 sending to Peer_0
[0] [INFO] Peer_1 sending to Peer_2
[0] [INFO] Peer_1 sending to Peer_3
```

Listing 5.23: The designed simulator's first nine text logs about messages sent

```
[Peer_0:peer:(1) 0.000000] [python/INFO] Peer_0 sending to Peer_1
[Peer_1:peer:(2) 0.000000] [python/INFO] Peer_1 sending to Peer_0
[Peer_2:peer:(3) 0.000000] [python/INFO] Peer_2 sending to Peer_0
[Peer_3:peer:(4) 0.000000] [python/INFO] Peer_3 sending to Peer_0
[Peer_4:peer:(5) 0.000000] [python/INFO] Peer_4 sending to Peer_0
[Peer_5:peer:(6) 0.000000] [python/INFO] Peer_5 sending to Peer_0
[Peer_6:peer:(7) 0.000000] [python/INFO] Peer_6 sending to Peer_0
[Peer_0:peer:(1) 0.000000] [python/INFO] Peer_0 sending to Peer_2
[Peer_1:peer:(2) 0.000000] [python/INFO] Peer_1 sending to Peer_2
```

Listing 5.24: SimGrid's first nine text logs about messages sent

By editing the log files so that both logs are in the same format, it is shown that the events with

the same timestamp are sent and received in different orders for the explained reason. Next, by ordering the logs using the `sort`² program, and then comparing them with the `diff`³ program (both programs are commonly available on the Unix command line), it is shown that the same events occur and that the same values are also observed (logs that are unique to one of the simulators' implementation were removed). Therefore, the order in which the simulators end up handling the events with the same timestamp differs. But, in contrast, the impact of that different order does not impact the overall results and calculations obtained and performed by the simulator and its peers.

To examine the metrics, a *Python* script was created to read the data and generate diagrams. Listing B.23 shows the *Python* code, which parses the metrics logs and stores them in a sorted array.

With the parsed data organized, the graphs are generated. Listing B.24 shows the *Python* code that creates a graph showing the average value calculated over time evolution. The resulting graph also includes a line showing the actual average value the system should obtain. The code from the listing results in Figure 5.5 and Figure 5.6, for the results obtained by the designed simulator and SimGrid, respectively.

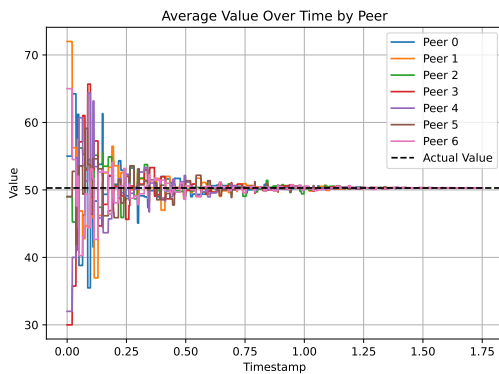


Figure 5.5: Average value over time by peer in the designed simulator

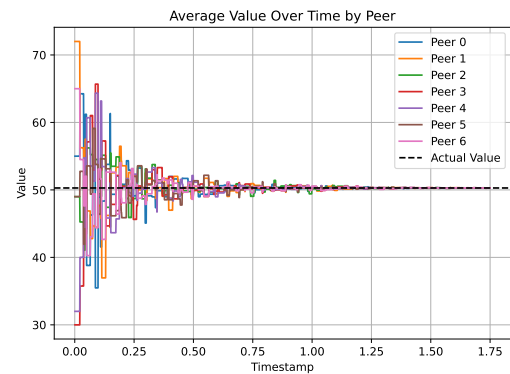


Figure 5.6: Average value over time by peer in SimGrid

Listing B.25 shows the *Python* code that creates a graph which, having calculated the sum of every calculated average value, shows the evolution of the difference between that value and the actual value over time. The code from the listing results in Figure 5.7 and Figure 5.8, for the results obtained by the designed simulator and SimGrid, respectively.

The JSON logs further confirm the conclusion when analysing the behaviour from the textual logs. Since the behaviour is the same, the metrics and the corresponding graphs are equal.

Faulty Experiment An advantage of the developed simulator is that it can quickly introduce faults into the system. Copying the experiment in the configuration file, renaming the copy and introducing the field `drop_rate` with a value of 0.2. The simulator can execute the two experiments in a single run, outputting the logs to be further examined. In this case, a new experiment

²<https://www.gnu.org/software/coreutils/>

³<https://www.gnu.org/software/diffutils/>

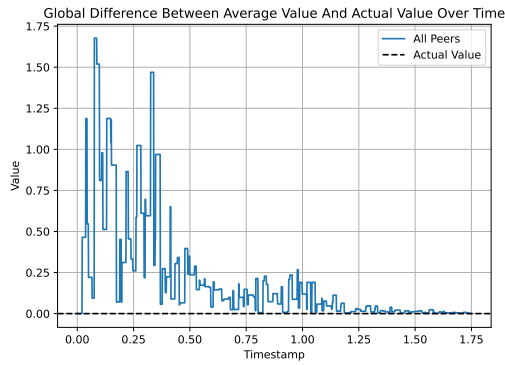


Figure 5.7: Global difference between the calculated and the actual average value over time in the designed simulator

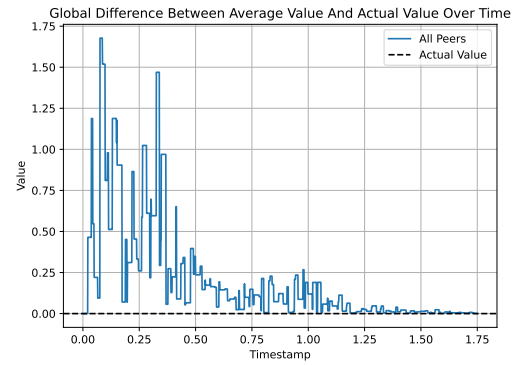


Figure 5.8: Global difference between the calculated and the actual average value over time in SimGrid

called "flow_updating_0_2" was executed with a drop rate of 0.2 (20% of the messages are not sent), and its configuration can be consulted in Listing B.26.

Using the code from Listing B.24, which draws the graph showing the average value calculated over time evolution. It can be seen when comparing Figure 5.9 (which is equal to Figure 5.5) to Figure 5.10 that the average values converge in a similar time frame, however, the values calculated in the faulty experiment suffer more drastic changes, an occurrence that is highlighted by the fact that, as seen in the figure, multiple peers had an average value lower than the minimum starting value.

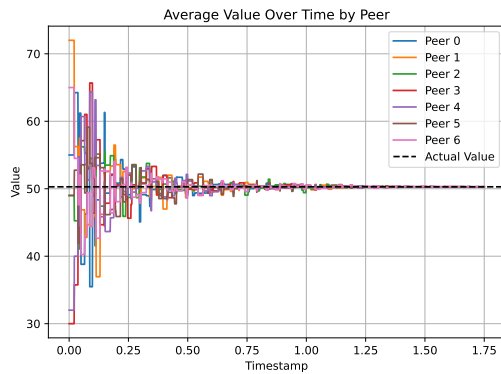


Figure 5.9: Average value over time by peer with no faults

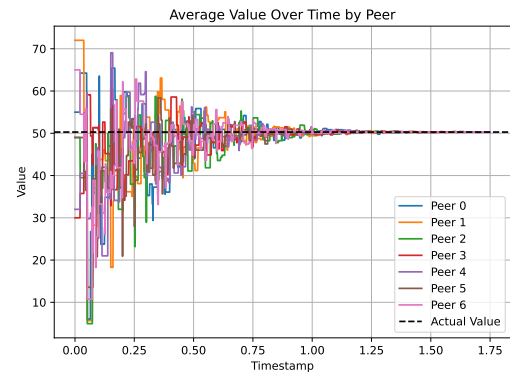


Figure 5.10: Average value over time by peer with dropped messages

Using the code from Listing B.25, which draws the graph showing the evolution of the difference between the total average and the actual average over time. It can be seen when comparing Figure 5.11 (which is equal to Figure 5.7) to Figure 5.12, that the difference is significant during the earliest stage of the simulation. Note that graphs do not have the same scale. The figure on the left goes from 0 to 1.75 on the y-axis, while the figure on the right goes from 0 to more than 30 on the same axis.

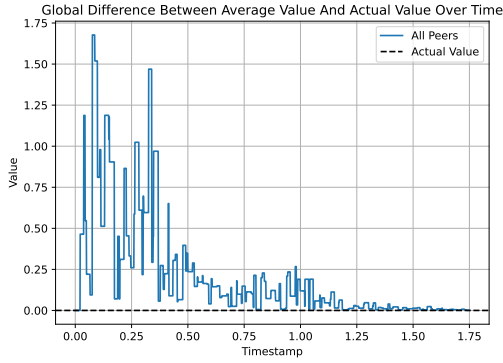


Figure 5.11: Global difference between the calculated and the actual average value over time with no faults

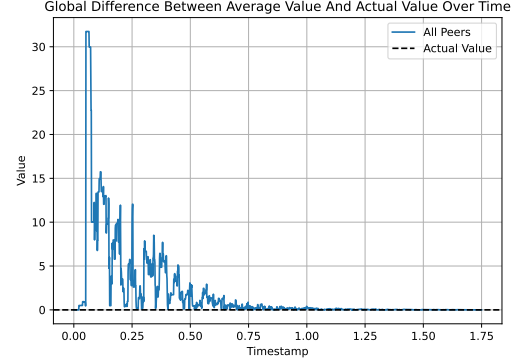


Figure 5.12: Global difference between the calculated and the actual average value over time with dropped messages

With these results in mind, this experiment shows that configuring additional experiments to compare results is not a complex task and only requires some configuration file changes.

Further, the JSON logs were instrumental in generating the graphs and tracking the evolution of the inspected metrics. Since the developed *Python* scripts created are reusable, this feature allows the researchers to build a pipeline to visualize the results of the experiments after they are executed. This capability is native to the designed simulator, eliminating the need for manual implementation. In contrast, SimGrid requires the functionality to be recreated for each scenario by, for example, as done in the Flow Updating scenario implemented, manually adding a global variable and an additional actor.

5.2 Statistical Learning Algorithm Evaluation

This section will implement a new scenario into the designed simulator. This scenario consists of implementing the distributed generalised linear model [31].

The scenario's statistical learning algorithm will be explained in Section 5.2.1, its implementation will be described in Section 5.2.2, experiments will be created in Section 5.2.3, and the conclusions of those experiments will be discussed in Section 5.2.4. Finally, a solution that attempts to solve the shortcomings highlighted in the studied algorithm will be implemented in Section 5.2.5.

5.2.1 Algorithm

The distributed generalised linear model (GLM) algorithm is a statistical learning algorithm that partitions the computational workload across multiple machines or processes to handle large datasets or to accelerate training through parallelization. It ensures scalability and adaptability to diverse data distributions while maintaining privacy-preserving properties [31]. However, fault tolerance was not part of its original design.

The distributed generalised linear model algorithm, initially implemented in the *R* programming language [30], will be translated to the *Rust* programming language in order to be implemented in the simulator.

In this algorithm, each of the peers will contain a portion of a dataset. All peers first calculate their local R matrix (R_{local}) and then broadcast their number of local observations (rows) from their initial portions of the dataset. Once a peer receives all counts of the local observations from other peers, it calculates the total number of observations and broadcasts its R_{local} matrix. When a peer receives another peer's R_{local} matrix (R_{remote} for the peer), the matrix is stored along with its corresponding iteration number. If a peer receives the R_{remote} matrices from all other peers in the network corresponding to its current iteration, the combined matrix is computed by vertically concatenating the local matrix and the remote matrices and used to obtain the global matrix (R_{global}), update the coefficients and check for convergence. If convergence is not obtained, the peer recalculates its R_{local} matrix using the updated coefficients and moves on to the next iteration. Otherwise, if convergence is obtained, the peer does not participate in the algorithm anymore as it has already acquired the final coefficients, which, in the end, for the intended purpose should be equal for every peer.

From this moment on, the algorithm performs multiple iterations until it converges. All peers broadcast their number of local observations (rows) from their initial datasets to enable proper global variance calculations.

The developed simulator should be able to simulate a scenario that implements this algorithm, effectively evaluating it. The evaluation should then help pinpoint the algorithm's flaws, allowing the algorithm to be adapted or mechanisms to be introduced in order to make it fault-tolerant.

5.2.2 Implementation

The distributed generalised linear model algorithm has been implemented in the *R* programming language [30]. That implementation will be translated to the *Rust* programming language in order to implement the scenario.

The designed simulator implements this algorithm by creating a new struct that implements the `Scenario` trait. Named `DistributedGeneralizedLinearModel`, this struct's implementation can be seen in Listing B.27. Registered on the simulator as "dglm", its `start` method begins by reading the input data from the file system using a method called `model_data` (whose definition is shown in Listing B.28), which returns an instance of the `ModelData` struct, which contains two matrices x and y . Then, another file is read from the file system using the function `model_beta` (whose definition can be observed in Listing B.29) and transformed into a matrix which corresponds to the coefficients that will later be used to verify if the simulation was successful. Before proceeding, the simulator asserts whether the data has a valid shape.

Having the input data loaded, the simulation divides it into as many portions as indicated by the number of peers declared. This division is performed by the function `chunk_nx` (whose definition can be consulted in Listing B.30) and each of the resulting chunks' x and y fields will be provided as an argument to the creation of a corresponding peer. Peers are instances of the `GlmPeer`

struct, which, when created, also require two-dimensional coordinates that, if not provided by the experiment's configuration, are randomly generated. Upon being created, the peers' instances are instantiated and added to the simulation state.

The `GlmPeer` struct (whose definition can be inspected in Listing B.31) holds the peer's internal information as well as its state, which, in itself, is another struct called `GlmState` (presented in the same listing). This struct contains seven fields, which are instantiated when the peer is created. These are:

- `model` - corresponds to an instance of the `GeneralizedLinearModel` struct whose initial values are calculated upon creating the peer holding it.
- `data` - stores the portion of the dataset assigned to this peer. It is an instance of the `ModelData` struct containing the `x` and `y` arguments received on peer creation.
- `r_n_rows` - a hash map that stores each peer's number of rows. The key is the peer's identifier.
- `r_remotes` - a nested hash map that stores the intermediate computation results from remote peers. The outer key is the peer's identifier, while the inner key is the iteration number.
- `total_nrow` - the total number of observations across every peer in the system. It is the sum of all values in `r_n_rows` plus the local data size.
- `nodes` - the identifiers of the peers belonging to the simulation that are also instances of the `GlmPeer` struct.
- `finished` - a flag that indicates whether the algorithm has been completed. If set to true, no more interactions are needed.

The `GeneralizedLinearModel` struct (whose definition can be observed in Listing B.32) holds four fields which are instantiated when the peer that holds this struct is created. Those fields are:

- `coefficients` - a column vector which will be updated at each iteration of the algorithm. Initially, the column vector is created with as many values as there are features (number of rows in the `x` argument), with every value set to zero.
- `r_local` - the local upper triangular matrix R from QR decomposition, specific to the peer's data partition. When the peer is created, its value is set to the value obtained from the first iteration of the local GLM computation using the initial zero coefficients. This iteration is performed using the `distributed_single_iter_n` function.
- `family` - specifies the probabilistic distribution family to be used. In this scenario, the family is set as Binomial.
- `iter` - the current iteration number for this peer, initially set to 0.

Creating a peer instance assigns it the callback called when a message is received. This callback's function is called `on_message_receive`, and its definition can be consulted in Listing B.33. It calls two different methods depending on the message it receives, as the message could be an instance of the message struct `GlmSumRowsMessage` or an instance of the message struct `GlmConcatMessage`, leading to the invocation of the function `receive_sum_rows_msg` or `receive_concat_r_msg` respectively. Both messages and their handling will be explained when their use is mentioned.

After creating all the peers involved in the simulation, the simulator, if provided with a topology in the experiment configuration, connects the peers using the topology registry and assigns the simulation's message delay callback using the arrival time registry.

With the peers created and connected, two timers are created. The first is responsible for the peers' start logic, while the second is purposely built to inject a peer-based fault into the simulation.

Due to its simplicity, the last timer will be explained first. It is called `KillTimer` (its definition can be found in Listing B.34) and requires an identifier (which will always be 0 in this scenario). If the experiment's configuration file has a boolean field called `kill_peer` set to true, an instance of this timer will be instantiated to be fired at timestamp 0.1. When fired, this timer calls the `kill` function of the peer with the given identifier, essentially removing him from the simulation. This timer is not part of the algorithm. Instead, its purpose is to allow the observation of the algorithm's behaviour after a peer is removed or can not communicate.

The first timer is called `StartTimer`, and its definition can be found in Listing B.35. In order to be created, an identifier is required. This value is used to indicate which peer the timer's logic targets. When fired, this timer invokes another method called `broadcast_sum_rows`, which receives the simulation context and the target peer identifier.

The `broadcast_sum_rows` method (whose definition can be inspected in Listing B.36) broadcasts the number of rows that the peer's local data has. This method obtains all the peers connected to the peer in question but also filters them, checking if they are alive and are of the same instance, storing a list of the filtered peers in a vector containing their identifiers. Checking if the peers are of the same instance is redundant since no other structs are used to introduce peers to the simulation. However, it was implemented in case future modifications introduce another peer type. The broadcasting is executed by calling the method `send_sum_rows` (whose definition can be observed in Listing B.37) to each of the filtered peers. The `send_sum_rows` method creates an instance of the `GlmSumRowsMessage` (whose definition can be consulted in Listing B.38) message struct, providing it with the sender's identifier and number of rows, and then sends the just created message to the target peer. Informing the target peer of the number of rows the sender peer has.

When a peer receives a message of instance `GlmSumRowsMessage`, its message received callback invokes the `receive_sum_rows_msg` function (whose definition can be seen in Listing B.39). This function checks if the receiving peer's state already has the number of rows from the sending peer. If it does, it simply outputs a warning because that is not supposed to

happen. If the receiving peer's state does not have the number of rows of the sending peer, the number is added. When a new number of rows is added, the receiving peer checks if he already received the number of rows of every peer. This is because the peers should start interchanging their local computation only after they know each other's existence and the total number of rows the result should have. This synchronization ensures all peers consistently know the global problem size before beginning the distributed computation. Therefore, if the receiving peer has received the number of rows from every peer mentioned in its list of neighbours (filtered during the `broadcast_sum_rows` call), it computes the total number of rows and calls the `broadcast_nodes` function.

The `broadcast_nodes` function (whose definition can be found in Listing B.40) calls the `send_concat_r` to each of the sending peer's neighbours. This second function (whose definition can be found in Listing B.41) creates an instance of the `GlmConcatMessage` (whose definition can be consulted in Listing B.42) message struct, providing it with the sender's identifier, his R_{local} matrix (`r_local` field) and his iteration number (which, if it is the first time being called, will be 0), and then sends it to the target peer.

When a peer receives a message of instance `GlmConcatMessage`, its message received callback invokes the `receive_concat_r_msg` function (whose definition can be seen in Listing B.43). This function checks if the receiving peer's state already includes the iteration that the message targets so that, if it does not, a new empty hash map is created on the peer's state `r_remotes` field, using the iteration identifier as the key. Then, after being sure that the iteration exists, another method named `handle_iter` is called.

The `handle_iter` method (whose definition can be consulted in Listing B.44) is a complex function which contains most of the algorithm's computations. Since this method is called after a `GlmConcatMessage` is received, its parameters are the receiver and sender's identifiers, the message's upper triangular matrix (R_{remote}) and the message's iteration number. This method requires that the receiving peer does not already have the sender's upper triangular matrix for that iteration. A warning is issued if it already has it and the method returns. In case it did not have it yet, the R_{remote} matrix received is added to the `r_remotes` field from the receiving peer's state with the key set as the iteration number.

If, when the `handle_iter` method adds a message's R_{remote} matrix to the receiving peer's state, the message's iteration is the same as the receiving peer state's iteration and the peer received every R_{remote} associated with the iteration from its neighbours, then every neighbour's upper triangular matrix, as well as the local upper triangular matrix, are stacked using vertical concatenation into a single global upper triangular matrix (R_{global}).

If, when the `handle_iter` method processes a received message containing an R_{remote} matrix, the message's iteration number matches the receiving peer's current iteration, and the peer has received an R_{remote} matrix from all of its neighbours, then the peer proceeds to aggregate these matrices. Specifically, it collects all received R_{remote} matrices for that iteration and appends its own R_{local} matrix. These matrices are then vertically concatenated into a combined matrix using the `mat_cat_vec` function, whose definition can be found in Listing B.45 because it was not

implemented in `faer-rs` [10] library.

After calculating, the combined matrix is used by the `distributed_single_solve_n` function (whose definition can be seen in Listing B.46) to calculate a single global upper triangular matrix (R_{global}), update the coefficients, and check for convergence. Following the function call, the receiving peer updates its state: the `r_local` is replaced by the calculated global upper triangular matrix, the coefficients are updated to the newly computed values, the iteration counter is incremented, and the `finished` field is set based on whether convergence was achieved.

If the coefficients have converged, the peer sets its `finished` flag to true because it does not need to participate in further iterations, as the distributed GLM algorithm has reached its solution (for this peer). Otherwise, in case the coefficients have not converged, the peer prepares for the next iteration by computing a new `r_local` (R_{local}) matrix using the updated coefficients via the `distributed_single_iter_n` function (whose definition can be found in Listing B.47), then broadcasts this information to other peers to continue the distributed computation.

Returning to the scenario's `start` method, only two processes are left. Setting up the hooks and starting the simulation loop.

This scenario contains the hook called when the simulation loop finishes. This hook is obtained by providing the coefficients that were parsed from the file system to a function called `on_simulation_finish_hook` (whose definition can be consulted in Listing B.48), which returns a boxed method with the logic to be executed when the simulation finishes. The hook is assigned through a function call instead of simply assigning it a function because the function call can introduce the value of the coefficients to the boxed method for them to be used in the future. When executed, the boxed method gathers the coefficients of every peer's state and compares them to the file system coefficients, logging the output.

Finally, the scenario's simulation loop is started.

5.2.3 Experiments

After implementing the distributed generalised linear model algorithm in a scenario, multiple experiments will be conducted in order to study the effects of various conditions. The experiments are designed to evaluate the algorithm, and therefore, the outcomes will be analysed. All experiments will use the same dataset used in the original implementation of the algorithm [30]. After executing all the experiments, if at least one of them shows that the algorithm has a flaw, which is expected since it is mentioned that it was not developed with fault tolerance in mind, an approach to resolving one of the flaws will be attempted.

The following sections include the experiments' configuration, results and analysis.

The experiments to be conducted are the following:

- `default_run` - does not introduce any side effects into the simulation.
- `run_100_peers` - introduces many peers into the simulation.
- `constant_latency` - the latency value is 1 second for every message sent.

- `custom_latency` - has pre-defined latency and bandwidth values.
- `ring_topology` - utilizes the built-in "ring" topology.
- `custom_ring_topology` - defines a ring topology with pre-defined latency and bandwidth values.
- `star_topology` - utilizes the built-in "star" topology.
- `custom_topology_oneway` defines a custom topology with unidirectional connections.
- `peer_killed` - deactivates a peer after some time.
- `drop_rate_0_2` - drops 20% of the messages sent.
- `duplicate_rate_0_2` - delivers 20% of the messages twice.
- `jitter` - introduces jitter into the simulation.

Because some of the characteristics that describe these experiments will be repeated, anchors and aliases were used for the common attributes `seed`, `arrival_time` and `topology` as can be seen in Listing 5.25. The information shown in the listing will be included in the same file as the experiments. The configuration file used for all the experiments which will be analysed can be found in Listing B.56.

```

1 seed: &default_seed 2586804559409604052
2 arrival_time: &default_arrival_time distance
3 topology: &default_topology
4   n_peers: 7
5   name: full

```

Listing 5.25: The configuration's defaults

5.2.3.1 default_run

Configuration This experiment demonstrates the behaviour of the distributed generalised linear model algorithm in normal conditions. Its configuration (Listing 5.26) uses the anchors defined before. However, this experiment, in particular, does not have a seed declared and is repeated one hundred times. Without a declared seed, the peers' positions are randomly generated, and because the arrival time is calculated using the "distance" callback, different distances between peers will be determined depending on the execution. Consequently, the messages are expected to have different latency values for each repetition.

```

1 - name: default_run
2   scenario: dglm

```

```

3 arrival_time: *default_arrival_time
4 topology: *default_topology
5 repetitions: 100

```

Listing 5.26: The "default_run" experiment configuration

Results and Analysis After executing this experiment, the logs show that each simulation has a different execution time and that every one of them is successful. This conclusion could be taken by running the following on the command line:

```
tail results/default_run/*/default_run_*.log | grep "central"
```

Which leads to filtering only the logs that are output by the `on_simulation_finish_hook`. A small excerpt of the result of this command can be found in Listing 5.27.

```

[0.46617561560275533] [INFO] Are the coefficients from every peer equal to central's?
      true
[0.450968826955373] [INFO] Are the coefficients from every peer equal to central's? true
[0.4206770810546292] [INFO] Are the coefficients from every peer equal to central's?
      true
[0.4120919763354216] [INFO] Are the coefficients from every peer equal to central's?
      true
[0.4246164237897534] [INFO] Are the coefficients from every peer equal to central's?
      true
[0.400742739199584] [INFO] Are the coefficients from every peer equal to central's? true
[0.37581636050422107] [INFO] Are the coefficients from every peer equal to central's?
      true
[0.38429957378668483] [INFO] Are the coefficients from every peer equal to central's?
      true
[0.4146768233538578] [INFO] Are the coefficients from every peer equal to central's?
      true

```

Listing 5.27: The designed simulator's first experiment log output

By running the following command:

```
tail results/default_run/*/default_run_*.log | grep "false"
```

The logs are filtered to include only the word "false", which should be present if the simulation obtains the incorrect value. Therefore, it can be confirmed that no execution leads to the wrong results.

5.2.3.2 run_100_peers

Configuration The second experiment is equal to the first but not only has many more peers. It also includes a seed so that consequent runs of the configuration file obtain the same results. However, the big difference between this experiment and the previous one is the number of peers.

```
1 - name: run_100_peers
2   scenario: dglm
3   seed: *default_seed
4   arrival_time: *default_arrival_time
5   topology:
6     n_peers: 100
7     name: full
```

Listing 5.28: The "run_100_peers" experiment configuration

Results and Analysis Having executed the experiment, the logs show that the algorithm is successful in handling a large number of peers. The textual logs show that each peer's number of rows was reduced to 13, but this did not affect the algorithm's final result.

5.2.3.3 constant_latency

Configuration It is a basic situation where the time between messages is always the same. This experiment replaces the default arrival time callback used by the previous experiments with the "constant" arrival time callback.

```
1 - name: constant_latency
2   scenario: dglm
3   seed: *default_seed
4   arrival_time: constant
5   topology: *default_topology
```

Listing 5.29: The "constant_latency" experiment configuration

Results and Analysis The logs show that having constant latency is not an obstacle to the algorithm, seeing as it obtained the right result.

5.2.3.4 custom_latency

Configuration The following experiment complicates the scenario configuration by introducing pre-defined latency and bandwidth. It repurposes the fully connected topology used in Figure 5.4 from Section 5.1.2.2, resulting in the topology shown in Figure 5.13.

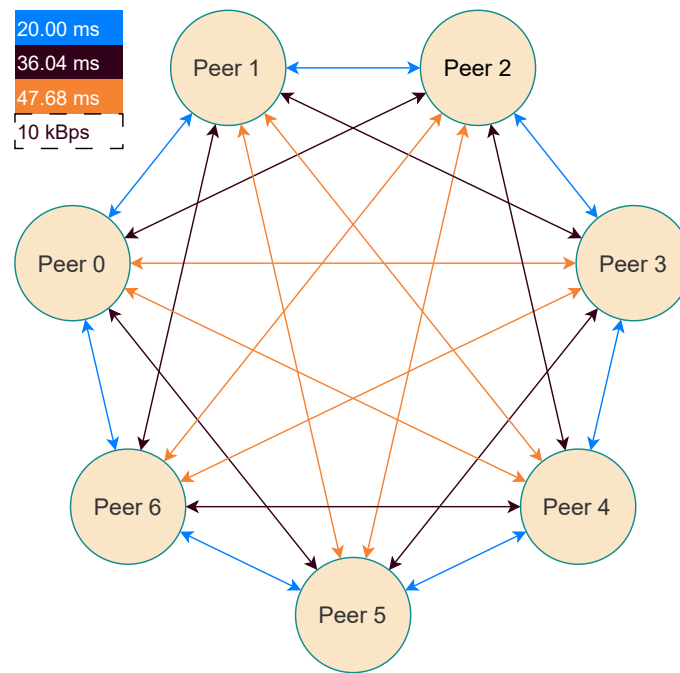


Figure 5.13: Fully connected experiment topology

```

1 - name: custom_latency
2   scenario: dglm
3   topology:
4     n_peers: 7
5     name: twowaycustom
6     connections:
7       - [0, 1, !Full {bandwidth: 80000, latency: 0.02} ]
8       - [0, 2, !Full {bandwidth: 80000, latency: 0.03604} ]
9       - [0, 3, !Full {bandwidth: 80000, latency: 0.04768} ]
10      - [0, 4, !Full {bandwidth: 80000, latency: 0.04768} ]
11      - [0, 5, !Full {bandwidth: 80000, latency: 0.03604} ]
12      - [0, 6, !Full {bandwidth: 80000, latency: 0.02} ]
13      - [1, 2, !Full {bandwidth: 80000, latency: 0.02} ]
14      - [1, 3, !Full {bandwidth: 80000, latency: 0.03604} ]
15      - [1, 4, !Full {bandwidth: 80000, latency: 0.04768} ]
16      - [1, 5, !Full {bandwidth: 80000, latency: 0.04768} ]
17      - [1, 6, !Full {bandwidth: 80000, latency: 0.03604} ]
18      - [2, 3, !Full {bandwidth: 80000, latency: 0.02} ]
19      - [2, 4, !Full {bandwidth: 80000, latency: 0.03604} ]
20      - [2, 5, !Full {bandwidth: 80000, latency: 0.04768} ]
21      - [2, 6, !Full {bandwidth: 80000, latency: 0.04768} ]
22      - [3, 4, !Full {bandwidth: 80000, latency: 0.02} ]
23      - [3, 5, !Full {bandwidth: 80000, latency: 0.03604} ]
24      - [3, 6, !Full {bandwidth: 80000, latency: 0.04768} ]
25      - [4, 5, !Full {bandwidth: 80000, latency: 0.02} ]
26      - [4, 6, !Full {bandwidth: 80000, latency: 0.03604} ]
27      - [5, 6, !Full {bandwidth: 80000, latency: 0.02} ]

```

Listing 5.30: The "custom_latency" experiment configuration

Results and Analysis This experiment's logs show that the algorithm obtains the correct result even when latency and bandwidth is introduced.

5.2.3.5 ring_topology

Configuration This experiment uses the built-in topology "ring" to assess whether the algorithm can obtain the correct results in another topology other than the fully connected topology. Note that, as shown in the configuration in Listing 5.31, the topology is named "ring".

```

1 - name: ring_topology
2   scenario: dglm
3   seed: *default_seed
4   arrival_time: *default_arrival_time
5   topology:
6     n_peers: 7
7     name: ring

```

Listing 5.31: The "ring_topology" experiment configuration

Results and Analysis This experiment's logs should indicate whether the algorithm functions correctly when providing the "ring" topology. However, when examining the textual logs, it is observable that the algorithm did not achieve the desired result, and it is not caused by the fact that the topology itself is of type "ring". The issue that led to no peer achieving convergence originated from peer 5 receiving its second (the peer has only two neighbours in this topology) GlmSumRowsMessage message only after it had already received a GlmConcatMessage message. As a result, it did not increase from iteration 0, leading to other peers being unable to move to other rounds. Therefore, as can be observed in Listing B.39, the `receive_sum_rows_msg` function resets the peer's `r_remotes` hash map, thereby losing the information that the first GlmConcatMessage message had provided. The logs which allowed this behaviour to be understood are shown in Listing 5.32.

```

[0.00414224053740488] [TRACE] peer 5 received a GlmSumRowsMessage message from peer 4
    with nrows: 188
...
[0.04889433597388251] [TRACE] peer 5 received a GlmConcatMessage from 4 on iteration 0
[0.04889433597388251] [TRACE] peer 5 handle_iter, received (1/2) messages on iteration 0
    (current iteration is 0)
...
[0.058658868546768946] [TRACE] peer 5 received a GlmSumRowsMessage message from peer 6
    with nrows: 191
[0.058658868546768946] [TRACE] Sent GlmConcatMessage from 5 to 4 (iteration 0) in
    0.00414224053740488
[0.058658868546768946] [TRACE] Sent GlmConcatMessage from 5 to 6 (iteration 0) in
    0.058658868546768946
...
[0.09364643141036014] [TRACE] peer 5 received a GlmConcatMessage from 4 on iteration 1

```

```
[0.09364643141036014] [TRACE] peer 5 handle_iter, received (1/2) messages on iteration 1
    (current iteration is 0)
...
[0.11731773709353789] [TRACE] peer 5 received a GlmConcatMessage from 6 on iteration 0
[0.11731773709353789] [TRACE] peer 5 handle_iter, received (1/2) messages on iteration 0
    (current iteration is 0)
...
[0.17597660564030684] [TRACE] peer 5 received a GlmConcatMessage from 6 on iteration 1
[0.17597660564030684] [WARN] peer 5 handle_iter, received all messages from iteration 1
    but current iteration is 0
```

Listing 5.32: Out-of-order messages in the ring_topology experiment

This issue occurred due to the latency between the peers. The peer with identifier 4 received its `GlmSumRowsMessage` messages and consequently sent a `GlmSumRowsMessage` message that was to be received by the peer with identifier 5 before that peer had received all of its `GlmSumRowsMessage`. This situation shows that the algorithm's first round, as it stands, does not deal with out-of-order messages.

5.2.3.6 custom_ring_topology

Configuration Following the previous experiment's result, because the topology's effects were not analysed, a new experiment was created to avoid the issue shown in the `ring_topology` experiment. This experiment has the ring topology shown in Figure 5.14.

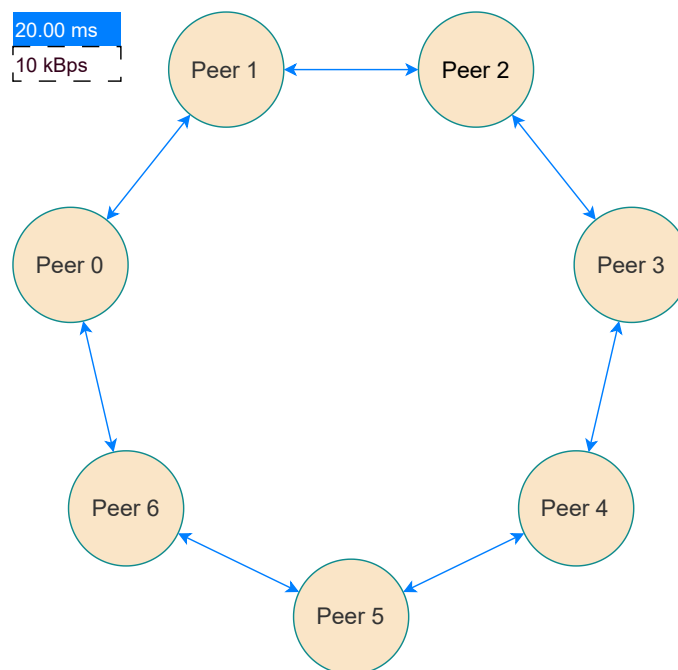


Figure 5.14: Custom ring topology

The configuration is shown in Listing 5.33.

```

1 - name: custom_ring_topology
2   scenario: *default_scenario
3   topology:
4     n_peers: 7
5     name: twowaycustom
6     connections:
7       - [0, 1, !Full {bandwidth: 80000, latency: 0.02} ]
8       - [1, 2, !Full {bandwidth: 80000, latency: 0.02} ]
9       - [2, 3, !Full {bandwidth: 80000, latency: 0.02} ]
10      - [3, 4, !Full {bandwidth: 80000, latency: 0.02} ]
11      - [4, 5, !Full {bandwidth: 80000, latency: 0.02} ]
12      - [5, 6, !Full {bandwidth: 80000, latency: 0.02} ]
13      - [6, 0, !Full {bandwidth: 80000, latency: 0.02} ]

```

Listing 5.33: The "custom_ring_topology" experiment configuration

Results and Analysis The resulting logs of this experiment show that the topology does not obtain the desired results. The peer with identifier 6 achieves convergence in round 10, and the peer with identifier 2 achieves it in round 11. Due to the peers not broadcasting when they achieve convergence, their neighbouring peers do not advance their current round and, therefore, do not achieve convergence. It can be concluded, therefore, that the algorithm does not obtain the desired results in the ring topology.

5.2.3.7 star_topology

Configuration This experiment uses the star topology given by the built-in "star" topology. Given that this experiment can be affected by the same issue that the `ring_topology` experiment was, it will be repeated ten times. Each run will be analysed to assess the topology's effect on the algorithm. Using multiple repetitions instead of pre-defined positions or pre-defined latency values enables the simulator to generate situations that may lead to unexpected outcomes. The configuration can be seen in Listing 5.34.

```

1 - name: star_topology
2   scenario: *default_scenario
3   arrival_time: *default_arrival_time
4   topology:
5     n_peers: 7
6     name: star
7     repetitions: 10

```

Listing 5.34: The "star_topology" experiment configuration

Results and Analysis The experiment's logs conclude that the algorithm does not obtain the correct values. Out of the ten runs, only two did not stall on the first iteration due to the peer that

connects to every other peer (the peer in the middle of the star shape) receiving an out-of-order message. The remaining two runs showed that every peer except for the middle peer achieved convergence on iteration 14. Therefore, only the middle peer did not. However, the JSON logs show that the peers that achieved convergence did not obtain the correct value. The result is incorrect because the peers that achieve convergence do so without the data from every other peer in the network.

5.2.3.8 peer_killed

Configuration To assess the algorithm's behaviour under peer-based faults, this experiment kills a peer during execution. As seen in Listing 5.35, an extra argument named `kill_peer` is introduced. This argument triggers the addition of a timer named `KillTimer` (whose definition can be seen in Listing B.34) to the simulation. The added timer is fired at timestamp 0.1, killing the peer with identifier 0. The algorithm is expected not to resist this fault as an unresponsive peer does not process nor send the required messages to the other peers, halting the algorithm because an iteration can not be advanced.

```

1 - name: peer_killed
2   scenario: *default_scenario
3   seed: *default_seed
4   arrival_time: *default_arrival_time
5   topology: *default_topology
6   extra_args:
7     kill_peer: true

```

Listing 5.35: The "peer_killed" experiment configuration

Results and Analysis The resulting logs show that the expected behaviour is indeed presented. The peer with identifier 0 (the killed peer) receives all its `GlmSumRowsMessage` messages and, consequently, sends its `GlmConcatMessage` messages. That allows the remaining peers to advance into iteration 1. However, because the killed peer becomes unresponsive before receiving all its corresponding `GlmConcatMessage` messages, it stops on iteration 0. As a result, every peer but the killed peer is on the second iteration while the unresponsive peer is on its first.

5.2.3.9 drop_rate_0_2

Configuration This experiment introduces a message drop rate to assess the algorithm's behaviour under message-based faults. This fault causes a pre-determined percentage of the messages the simulator sends not to be delivered. For this experiment, as seen in Listing 5.36, 20% of the messages sent will be dropped by the simulator. It is expected that the algorithm does not resist this adversity because a single missed message would result in the inability to advance iterations.

```

1 - name: drop_rate_0_2
2   scenario: dglm
3   seed: *default_seed
4   arrival_time: *default_arrival_time
5   topology: *default_topology
6   drop_rate: 0.2

```

Listing 5.36: The "drop_rate_0_2" experiment configuration

Results and Analysis Dropping messages lead to the expected result as the logs show that not every `GlmSumRowsMessage` is delivered. Because of this, the algorithm halts before some of the peers even start iterating. The logs show that only the peers with identifiers 1 and 3 start sending `GlmConcatMessage` messages.

5.2.3.10 duplicate_rate_0_2

Configuration Another message-based fault is the delivery of duplicate messages. Therefore, this experiments is configured to deliver some messages twice. As seen in Listing 5.37, 20% of the messages delivered will be delivered twice.

```

1 - name: duplicate_rate_0_2
2   scenario: dglm
3   seed: *default_seed
4   arrival_time: *default_arrival_time
5   topology: *default_topology
6   duplicate_rate: 0.2

```

Listing 5.37: The "duplicate_rate_0_2" experiment configuration

Results and Analysis The results obtained by executing this experiment show, through the logs, that the algorithm is resistant to duplicate messages. Since both the `receive_sum_rows_msg` method (whose definition is shown in Listing B.39) and the `handle_iter` (Listing B.44) method, which is called after `receive_concat_r_msg` (Listing B.43) method, check if the content of the received message has already been processed, the duplicate message is ignored and therefore does not impact the algorithm.

5.2.3.11 jitter

Configuration The last experiment introduces the jitter into the default experiment to attempt to simulate the network latency variation. Listing 5.38 shows the parameters used for the jitter, which are presented in Section 4.7.1.3. This experiment's configuration is equal to the `default_run` experiment, aside from the jitter introduced.

```

1 - name: jitter
2   scenario: dglm
3   arrival_time: *default_arrival_time
4   topology: *default_topology
5   jitter:
6     distribution: !Weibull [1.064, 2.872]
7     multiplier: 0.001
8     repetitions: 100

```

Listing 5.38: The "jitter" experiment configuration

Results and Analysis Although the `default_run` experiment had all its 100 runs obtain the correct result, it only obtained the correct result in 57 runs of this experiment. Note that the seeds from each run in both experiments are different. The introduction of jitter to the experiment influenced the result because it introduced out-of-order messages, and, as already seen, the algorithm halts when a `GlmSumRowsMessage` message arrives after a `GlmConcatMessage` message because the peer's state forgets the first to arrive.

5.2.4 Evaluation Overview

Following the execution of the experiments in the scenario designed to evaluate the distributed generalised linear model algorithm, the resulting logs highlight not only which situations the algorithm can handle but also the flaws and imperfections of its implementation. The experiments exposed the following flaws:

- **Out-of-order Messages** - The algorithm fails whenever a peer receives a `GlmConcatMessage` message before the arrival of the required `GlmSumRowsMessage` messages. This causes the peer's state to reset its `r_remotes` hash map, deleting the previously received computation results (R_{remote} matrices). This flaw was observed in the following experiments: `ring_topology`, `star_topology` and `jitter`.
- **Lack of Peer-Based Fault Tolerance** - The algorithm fails if one of the peers becomes unresponsive. Whenever a peer stops processing and receiving messages, it halts the algorithm's execution because the peer cannot advance its iteration and keeps others from advancing their own. In order to advance its iteration, a peer needs to receive a message from every other peer. If one of the other peers does not send its message, the peer can not proceed. This flaw was observed in the `peer_killed` experiment.
- **Message Delivery Failures** - The algorithm fails if a message is not delivered. Since the algorithm was thought without fault tolerance in mind, there are no mechanisms for defending the algorithm from message loss faults. The messages have an essential role not only in sharing data between peers but also in synchronizing them. Therefore, the loss of a message means not only will the affected peers lose information but also not notify them whether the

iteration should proceed. The consequences of this flaw are similar to the consequences of the out-of-order messages issue previously mentioned because, by deleting the information that the `GlmConcatMessage` message delivered earlier, it is as if the message had been dropped. This flaw was seen in the `drop_rate_0_2` experiment.

- **Reliance on Full Connectivity** - The algorithm requires that the peers know the other's computational results to iterate. However, the implementation does not have any way of sharing the needed data if two of the network's peers are not connected. Because of that lack of connection, the convergence results are incorrect. The algorithm succeeded only when the topology was fully connected in the experiments performed. This flaw was observed in the `ring_topology`, `custom_ring_topology` and `star_topology` experiments.
- **No Convergence Signaling** - Whenever a peer converges, the algorithm does not notify the remaining peers of this occurrence. Due to the remaining peers being unaware of this occurrence, they can become stuck waiting indefinitely in their current iteration if they need to wait for a message from the peer who has finished their iterations. This was particularly evident in the `custom_ring_topology` experiment.
- **Lack of Timeout Mechanisms** - Whenever the algorithm is impacted by one of the flaws described, it stops because messages stop being delivered or processed. The algorithm has no mechanism that detects when adversity is encountered. A mechanism that could be useful would be the existence of timeouts. It would allow a peer to detect whether a message is missing and be able to request it again. However, timeouts must be fine-tuned to avoid false positives in slow network conditions.

Finding flaws and pinpointing their origins is essential to improve the statistical learning algorithms. In this case, the distributed generalised linear model algorithm can be improved based on the identified flaws. The resulting changes to the algorithm can then be re-evaluated using the simulator. This simulator-enabled iterative process supports the complete research, development, and evaluation of distributed statistical learning algorithms.

5.2.5 Proposed Solution

Knowing which algorithm's characteristics can be improved, a solution will be considered and implemented for some of them. The solution will not be optimal as it was created to showcase how a researcher can use the designed simulator to improve and modify the evaluated algorithms.

The characteristic of the generalised linear model algorithm, which was thought to improve, was the **Reliance on Full Connectivity** mentioned previously. However, more characteristics were added as a consequence of the approaches taken.

Since the algorithm requires that the peers know other peers' R matrices, it only obtained the correct result in the experiments where the topology was fully connected. For example, a peer belonging to a ring topology will not be connected to every other peer and, therefore,

will only receive its neighbours' R_{remote} matrices. It leads to incorrect results, as shown by the `custom_ring_topology` experiment (described in Section 5.2.3.6). To obtain the correct calculations, the peer of said topology would need to receive somehow the R_{remote} matrices of peers who are not his neighbours.

The initially conceptualised solution was to have the peers propagate their messages to reach those not directly connected to them. However, this approach would require a mechanism to prevent messages from looping indefinitely. One idea was to store the identifiers of the peers that had already received the message within the message itself before propagating it. This strategy was ultimately discarded because if a peer broadcasts a message, it introduces multiple instances of the same message into the network. Another approach considered was to assign a unique identifier to each message before broadcasting. While avoiding the propagation of duplicated messages, peers would be required to maintain a record of all received message identifiers.

Since the current algorithm already stored some of the contents of the messages in the peers' state (R_{remote} matrices), the chosen solution was to have each peer retain all relevant information received and request missing data from its neighbours when needed. However, this approach increases the number of messages exchanged and raises a significant concern: what would happen if the neighbouring peer does not have the requested information? The issue highlighted by the question becomes more impactful if the peers are far away from each other. As shown in Figure 5.15, when Peer 0 requests Peer 2's R matrix, Peer 1 may not have it. In such cases, Peer 1 forwards the request to its neighbours. In the case presented in the figure, the neighbour is Peer 2 and, therefore, successfully provides the requested matrix. At this point, however, Peer 1 must remember that the request originally came from Peer 0. A table of operations could have been used, where the reception of Peer 2's matrix would trigger its forwarding to Peer 0. However, this approach would again increase the amount of information each peer needs to store.

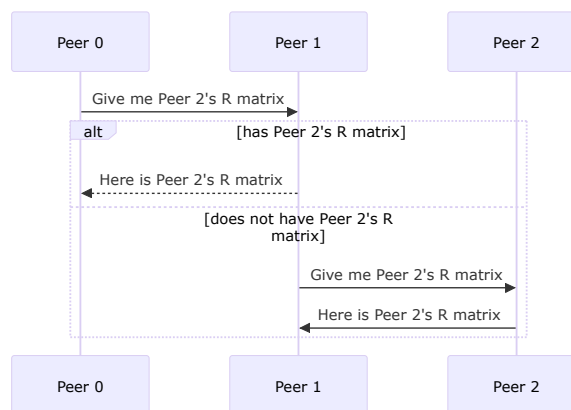


Figure 5.15: Peer requesting R matrix from its neighbour

The solution chosen was to use *timeouts*. This mechanism would allow peers to request the information once it finally reaches a neighbour. It is a costly mechanism timewise as data traverses the network as fast as the timeouts are called. Figure 5.16 illustrates this drawback by presenting

an example where the timeout mechanism eventually allows the peer to receive the requested information. However, as can be observed, due to `Peer 1` not sending the information requested immediately upon receiving it, the timeout duration directly impacts how early the information is obtained because `Peer 1` only responds to an explicit request. Consequently, longer timeouts delay the propagation of information across the topology. A side effect of this solution is that if a request message is lost, the peer can attempt to retrieve the information again the next time the timeout is triggered. Therefore, the introduction of *timeouts* allows another three problems identified on the algorithm to be mitigated: the **Lack of Timeout Mechanisms**, **Message Delivery Failures** and **Out-of-order Messages**. The last one is due to the algorithm being able to request the deleted messages from the peer affected.

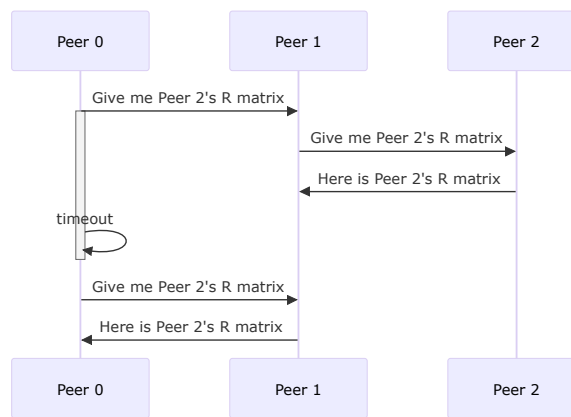


Figure 5.16: Peer requesting R matrix from its neighbour through a timeout

How to transmit data between peers to have all the information needed is already conceptualised. However, it assumes that the peers are aware of their total number. As per the algorithm, a peer only starts its iterative process once he is aware of the total number of rows obtained, which is done by summing the number of rows of each peer. If a peer had the incorrect assumption that the network had fewer peers than it had, it would start the iterative process with the wrong parameters, miscalculating its *R* matrix, leading the simulation to achieve incorrect results, if any. One possible solution to this problem would be to provide each peer with the total number of peers in the network *a priori*. However, the chosen approach was to introduce a discovery phase, which addressed this issue and attempted to mitigate the **Out-of-order Messages** problem by introducing an alternative method for initiating the iterative process.

The discovery phase was initially designed to have peers propagate `GlmSumRowsMessage` messages to announce their existence. A new identifier being received indicated the presence of an unknown peer, signalling that the receiving peer's awareness of the network's participants was still incomplete. A peer would only consider that it is aware of the network's participants upon triggering the stabilisation timeout. This timeout occurs after the number of identifiers the peer receives does not change in a specific timeframe. As a consequence of the timeout, the peer would assume it is aware of the total number of participants, would also have knowledge of the number

of rows associated with each (provided in the `GlmSumRowsMessage` messages), and therefore could then start sending `GlmConcatMessage` messages to initiate the iterative process.

This approach was ultimately not adopted due to the reliance on the specific duration of the timeout. If a peer's messages take too long to reach another peer and the timeout is called before a peer is aware of another's existence, the whole iterative process is ruined, as it would need to be restarted. Even restarting using this approach would not be the best option because message drops would risk a peer not receiving the necessary message that signals that it needs to restart. Since no mechanism verifies if two peers are in the iterative process with the same network awareness, the calculated R matrices may be calculated using the wrong data, leading to incorrect results.

With this in mind, restarting the iterative process was chosen as a feature of the algorithm. With each message sent, a hash of the list of identifiers belonging to the network would be added. The hash allowed peers to determine if they shared the same network awareness quickly. If they did, messages would be processed normally. If they did not, they would merge their list of identifiers, restarting the iterative process (updating their hash).

Using a hash that only processes messages from peers with the same network awareness ensures that the computations are performed with the correct coefficients, as the received $R_{remotes}$ matrices were also generated by peers with an identical view of the network.

5.2.5.1 Implementation

The implementation of this proposed solution can be divided into three main changes to the original algorithm, these are: the use of requests for information, the introduction of timeouts, and the discovery phase.

Discovery Phase The discovery phase is not limited to the beginning of the simulation. Instead, it can occur when a peer receives a message from another peer who has a different view of the network.

Since it can occur at any point during the simulation, peers attempt to synchronize their list of known peers only when they encounter a message from a peer with a different view of the network. Its implementation required changes to the algorithm's original structures and methods. In particular, the peer's state must be reset to the original values once there are changes in the network since the data calculated (not the number of rows) would result from outdated data and, therefore, be incorrect.

This behaviour can be observed in the modified `StartTimer` struct implementation (whose new definition can be seen in Listing B.57 and compared to the original in Listing B.35) which serves as the peers' entry point to the algorithm. While the original `StartTimer` timer would broadcast `GlmSumRowsMessage` messages while storing the identifiers of the peer's neighbours, this new version starts by storing the identifiers of the neighbours and preparing the peer's state before broadcasting the number of rows to its neighbours. This means that, although it has not yet received information about whether its neighbours are the only peers in the network, it sends its number of rows regardless. Before explaining how the peers will eventually receive the remaining

identifiers on the network (in case the network is not fully connected), the methods called in the `StarTimer`'s activation method will be discussed.

The `get_node_ids` method (whose definition can be consulted in Figure B.58), when called, obtains the identifiers of the peer's neighbours and adds them to its state. The identifiers involved in the simulation are stored using *Rust*'s standard library `BTreeSet` data structure⁴ so that the identifiers are ordered, readily comparable, and the hash calculation outputs the same result independently of the order at which the identifiers were added. During the execution of this method, the peer's state is prepared for a discovery phase. Being prepared for a discovery phase means incorrect and outdated data is discarded (data calculated when the number of peers differed). This discard is done every time the list of known peers is updated through the invocation of the `discovery_reset` method (whose definition can be found in Listing B.59), implemented in the `peer` struct.

The `broadcast_sum_rows` method (whose new definition can be seen in Listing B.60 and the original in Listing B.36) had its responsibilities reduced. Being the first method called when the peers started, it would be responsible for assigning the list of known peers to the peer's state. However, since the `get_node_ids` method already takes care of that, the method broadcasts the target peer's number of rows.

After broadcasting the peer's number of rows, the messages will only be processed if the sender and receiver peers have the same list of known identifiers. Identical lists lead to identical hashes, as Figure 5.17 illustrates. When one of the peers from the figure receives a message from another, since their hashes are equal, the message is processed.

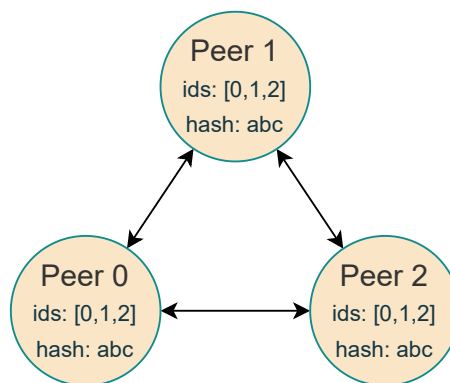


Figure 5.17: Fully connected topology with known identifiers and hash

However, Figure 5.18 shows a scenario where two peers are not connected and, therefore, do not have the same list of known identifiers and, consequently, the same hash. When one of the peers from the figure receives a message from another, since their hashes are different, the peers involved must obtain the missing identifiers.

⁴<https://doc.rust-lang.org/std/collections/struct.BTreeSet.html>

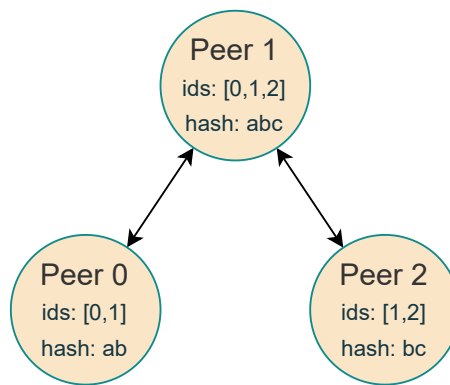


Figure 5.18: Non fully connected topology with known identifiers and hash

To compare the hashes of the peers communicating, the defined message structs include an extra field named `hash` of type `u64`. The struct definitions which contain the `hash` field can be consulted in Listing B.61 for the `PGLmSumRowsMessage` message, Listing B.62 for the `PGLmConcatMessage` message, Listing B.63 for the `ReqSumRowsMessage` message, and Listing B.64 for the `ReqConcatMessage` message. Section 5.2.5.1 will discuss the last two message structs.

The `hash` field is checked when a message of the previous message structs mentioned is received. They are checked using the method `operation_msg_hash_guard` (Listing B.65) which is called in the peer's message reception callback named `on_message_receive` (Listing B.66). This method verifies whether the received message and the receiver peer have the same hash. If they do, the message is processed. However, if they do not have the same hash, the message is not processed, and instead, a method name `send_discovery_msg` is invoked.

The `send_discovery_msg` method (Listing B.67) is responsible for sending the known list of identifiers to a target peer. A new message struct was created for this operation alone since it must contain the list of known identifiers and does not require the hash to be sent along. This message struct is called `DiscoveryMessage` (Listing B.68), containing the sender's identifier and the sender's list of known identifiers.

Once a peer receives a `DiscoveryMessage` message, it processes it using a method named `receive_discovery_msg` (Listing B.69) which verifies if the discovery message includes identifiers unknown to the receiver peer. In the case where the receiver peer adds a new identifier to its list of known identifiers, it calls the already explained `discovery_reset` method (recalculating the hash and discarding outdated data) and broadcasts `DiscoveryMessage` messages so that the new list of identifiers is shared with its neighbours, who may not be aware of the newly added identifiers. Finally, after broadcasting, a method named `check_missing_sum_rows` is called, which will be explained in Section 5.2.5.1.

Information Pooling By having a list of the known peers in the system, a peer can determine which messages it requires for its progression within the algorithm. The previously mentioned

`check_missing_sum_rows` method is an example of a method whose purpose is determining which messages the peer is missing.

After updating the list of known identifiers, the peer will miss the number of rows of the newly added identifiers' peers. Therefore, it calls the `check_missing_sum_rows` method. This method (whose definition can be consulted on Listing B.70) first verifies the identifiers whose number of rows is unknown to the peer and uses that list to create a message of type `ReqSumRowsMessage` (Listing B.63). This message struct contains the list of identifiers mentioned and the hash of the sending peer. After being created, it is broadcasted to the sending peer's neighbours.

Upon receiving a message of type `ReqSumRowsMessage` (handled by the `receive_sum_rows_req_msg` method whose definition can be consulted on Listing B.71), the receiving peer verifies if it contains each of the requested identifiers' number of rows and if it does, sends back the number to the requester using a `PGLmSumRowsMessage` message. Note that if the receiving peer does not have the requested information, the request is not propagated. Instead, the peer relies on the timeout mechanism, which will be explained in Section 5.2.5.1.

In the original algorithm, the iterative process would start after receiving each known peer's number of rows. This proposed solution is no different. When a `PGLmSumRowsMessage` is received, the `receive_sum_rows_msg` (Listing B.72) verifies if all the number of rows are known, and if they are, instead of broadcasting its R_{local} matrix like the original algorithm, it requests the other peers' R matrices (for iteration 0 as it is just starting the iterative process). The method responsible for requesting `PGLmConcatMessage` messages (Listing B.61), is the `check_missing_concat_r` method (Listing B.73). A request for an R matrix is created using a message of type `ReqConcatMessage` (Listing B.64), which contains the list of identifiers who calculated the wanted R matrices, the number of the iteration, and the hash of the sending peer.

Upon receiving a message of type `ReqConcatMessage` (using the `receive_concat_r_msg` method whose definition can be consulted on Listing B.74), the receiving peer verifies if it contains each of the requested identifiers' R matrices and on the requested iteration and if it does, sends them back to the requester using a `PGLmConcatMessage` message. The request is not propagated if the receiving peer does not have the requested matrices. The peer relies on the timeout mechanism to propagate the R_{remote} matrices.

The iterative process progresses when all the R matrices on the current iteration are collected. This occurrence is processed by the modified `handle_iter` method (Listing B.75) which differs from the original (Listing B.44) in that it does not broadcast its new iteration's R_{local} matrix but instead requests the iteration's R_{remote} matrices through the `check_missing_concat_r` method.

Although not directly mentioned yet, the peer struct was renamed and its state struct was modified to adhere to the functional requirements brought by the new approach. Their definitions can be consulted on Listing B.76 for the `PGLmPeer` struct and on Listing B.77 for the `PGLmState` struct.

the calculations to use the wrong R matrix. For example, the algorithm failed to obtain the correct results because the `discovery_reset` method was not resetting the R_{local} matrix to its initial value.

The experiments will not be detailed here, as they have already been explained in Section 5.2.3. Due to the existence of a timer which only stops re-adding itself to the simulation once every peer has finished, a deadline of 500.0 was added to each of the experiments. This way, the experiments can finish if the algorithm halts, not allowing the peers to finish.

The configuration file for the experiments can be consulted in Listing B.80, and the results of running the experiments with the proposed approach are presented in Table 5.1, which compares the results obtained by both implementations.

Table 5.1: Comparison of Distributed Generalised Linear Model Algorithm Experiments Results

Experiment	Original Algorithm		Proposed Solution	
	Success	Repetitions	Success	Repetitions
default_run	✓	100/100	✓	100/100
run_100_peers	✓	1/1	✓	1/1
constant_latency	✓	1/1	✓	1/1
custom_latency	✓	1/1	✓	1/1
ring_topology	×	0/1	✓	1/1
custom_ring_topology	×	0/1	✓	1/1
star_topology	×	0/10	✓	10/10
peer_killed	×	0/1	×	0/1
drop_rate_0_2	×	0/1	✓	1/1
duplicate_rate_0_2	✓	1/1	✓	1/1
jitter	×	57/100	✓	100/100

Extra Experiment An extra scenario was created and tested because of the proposed implementation's success in handling various network conditions. This scenario is the addition of a peer during the simulation. To implement this occurrence without having to change how the implementation instantiates peers (it would need to instantiate a peer later with the parameter given in the configuration), when an extra field `revive_peer` exists in the configuration, the first peer is killed before starting. Then, a new timer called `ReviveTimer` (Listing B.81) is added into the simulation, being triggered at timestamp 1.0. Upon being triggered, the timer calls the `revive` method on the killed peer and invokes the same methods invoked in the `StartTimer`. As a result, the once-killed peer starts communicating with the network's peers, forcing their iterative process to stop and the discovery phase to begin. Since the discovery phase was designed to occur at any point during the simulation, this experiment validates that specific behaviour. The configuration for the experiment mentioned can be consulted on Listing 5.39.

```
1 - name: peer_revived
2   scenario: proposed_dglm
3   seed: 2586804559409604052
4   arrival_time: distance
5   topology:
6     n_peers: 7
7     name: full
8   extra_args:
9     revive_peer: true
10  deadline: 500
```

Listing 5.39: The configuration of an extra experiment for the proposed solution

The experiment results show that adding peers to the network while iterating can be done. On the other hand, as shown by the `peer_killed` experiment, the proposed solution fails if a peer is removed from the network after the other peers are aware of its existence. Therefore, the proposed solution can only deal with network volatility if said volatility adds or temporarily disables peers. A permanently removed peer impossibilates the algorithm to achieve the desired results.

Analysis Naturally, focusing solely on whether the outcome is correct, the experiments yield better results with the proposed approach. The designed simulator's logging capabilities can obtain metrics such as messages sent, time to finish, and number of rounds to converge which were not extracted in these experiments as the focus was to attempt to solve the **Reliance on Full Connectivity** characteristic. However, after the proposed solution is shown to obtain favourable results, an iterative procedure that considers these metrics can be started to improve the algorithm.

Chapter 6

Conclusion

Evaluating distributed algorithms is inherently complicated, given the complexity of the algorithms and the environments in which they operate. Simulation is crucial in developing and assessing such algorithms, offering a faster and cost-effective way of drawing meaningful conclusions.

A simulator that supports the implementation of various algorithms, enables the reproduction of experiments with repeatable randomness, and provides built-in logging capabilities can significantly accelerate the research process. Most importantly, it removes the burden on researchers to build their simulations from scratch, allowing them to focus on developing and analysing their algorithms.

This dissertation reviews several existing simulators, analysing their strengths and limitations in the context of evaluating distributed statistical learning algorithms. Based on this analysis, the decision to adopt one of the simulators was weighted against the option to design a new simulator that combined the advantages shown by the simulators. Ultimately, the choice was made to design a new discrete-event simulator that was purpose-built to support the evaluation of the target algorithms. Its capabilities are demonstrated through experiments involving the distributed generalised linear model [31].

6.1 Main Contributions

This research studies discrete-event simulation as a tool for evaluating distributed algorithms, focusing on the algorithms' theoretical behaviour rather than physical constraints. The research explores how discrete-event simulators can effectively assess distributed statistical learning algorithms' correctness, efficiency, and scalability. This dissertation provides a practical evaluation framework that supports the development and study of the target algorithms.

1. Studied the specification of discrete-event simulation systems, including different types such as single logical process, parallel, and distributed simulations.

2. Reviewed the Sinalgo [13], QUANTAS [23], and SimGrid [6] simulators, describing their characteristics, strengths, and limitations, as well as providing a comparative analysis.
3. Designed and built¹ a discrete-event simulator purpose-built for evaluating distributed statistical learning algorithms.
4. Detailed how the distributed generalised linear model can be evaluated using the designed simulator, point out which flaws were found in its algorithm implementation.
5. Proposed a solution for the flaws found in the distributed generalised linear model algorithm.

6.2 Research Answers

With the research presented in this dissertation now complete, the research questions enunciated in Chapter 1 can now be answered.

6.2.1 RQ1. How can the state-of-the-art discrete-event simulation methodologies be used to evaluate distributed algorithms efficiently?

State-of-the-art discrete-event simulation methodologies facilitate efficient distributed algorithm evaluation through event-driven execution and modelling of only relevant aspects of the simulated environment. The relevant aspects of the simulated environment are characteristics such as which peers can communicate with whom, how long messages should take to reach their destination, and whether a peer is responsive. There is no need to add complexity to the simulation environment by adding system details. Therefore, by being able to represent the simulation environment analytically and combining it with the event-driven execution, multiple scenarios can quickly be set up and executed, allowing the introduction of network failures, topology changes and adverse conditions, which would be challenging to replicate physically. Gathering information about the scenarios allows them to be studied and evaluated.

6.2.2 RQ2. How can distributed statistical learning be specified and modelled for discrete-event simulation?

Distributed statistical learning can be specified for discrete-event simulation by separating data requirements from logical components. Data requirements define what information each peer must store, while logical components contain the processing logic triggered by simulation events.

6.2.3 RQ3. How is a discrete-event simulator for the purpose of evaluating distributed statistical learning algorithms designed?

A discrete-event simulator for the purpose of evaluating distributed statistical learning algorithms must be designed with flexibility, extensibility, configurability and observability in mind.

¹<https://github.com/AvilaAndre/tdes>

By providing flexibility and extensibility, the range of algorithms that can be implemented increases due to their implementation not having to follow strict guidelines. By being configurable, the simulator allows for more scenarios to be simulated and studied. However, the study of the scenarios is provided by the simulator's observability, which provides the tools needed to analyse the behaviours and metrics obtained by each scenario without needing external tooling.

6.2.4 RQ4. How can we characterise and evaluate a distributed algorithm?

A distributed algorithm can be characterised and evaluated by analysing its correctness, complexity and fault tolerance both theoretically, and empirically.

- **Correctness** refers to the algorithm's ability to obtain accurate and expected results.
- **Complexity** involves metrics such as converge time and the number of messages exchanged.
- **Fault tolerance** is the algorithm's capacity to maintain correctness despite adversities and failures.

These three characteristics are the primary focus of the experiments conducted in this work. Discrete-event simulation is important in the empirical evaluation as it bridges the theoretical foundations to observable behaviour.

6.3 Future Work

This dissertation could be extended through additional modifications and further experimental evaluation.

Simulator Enhancements

Several features were not implemented but could significantly benefit the researchers. Regarding the visualisation of metrics, a customisable global object could be introduced to the simulation state which the researcher could use to log global information. For instance, the object could be used to track the number of messages sent during a simulation by incrementing a counter each time a message was sent.

Additionally, for metric visualisation, a new logger level could automatically capture simulator events (message transmission, event processing, peer failures, among others) without requiring source code modifications, allowing a better analysis of simulator behaviour.

Regarding configurability, when using repetitions, if a seed is provided, the simulator could use it to generate the seeds for the simulations (so that every time the repetitions are executed, they receive the same seeds) instead of running every repetition with the same seed. This is fundamental to allow the researcher to reproduce the same results.

More broadly, the designed simulator's performance could be compared to that of the simulators studied, providing insights regarding which is more performant for complex simulations.

Proposed Solution

Finally, regarding the proposed solution, by design it sends significantly more messages than the original algorithm. However, this overhead was not verified in this work. A detailed analysis of the number of messages sent could identify specific optimisation opportunities, such as the mentioned table of operations or a timeout timer that considers whether the peer is receiving messages, reducing communication costs. Other performance metrics such as convergence time, fault tolerance, and scalability could be evaluated and compared against the original algorithm to provide a more comprehensive assessment of the trade-offs involved.

References

- [1] Paulo Sérgio Almeida et al. “Fault-Tolerant Aggregation: Flow-Updating Meets Mass-Distribution”. In: *Principles of Distributed Systems*. Springer Berlin Heidelberg, 2011, pp. 513–527. ISBN: 9783642258732. DOI: [10.1007/978-3-642-25873-2_35](https://doi.org/10.1007/978-3-642-25873-2_35). URL: http://dx.doi.org/10.1007/978-3-642-25873-2_35.
- [2] Daniel Garcia Bernal. “Decentralizing Large-Scale Natural Language Processing with Federated Learning”. MA thesis. KTH School of Electrical Engineering and Computer Science, 2020.
- [3] Ben Romdhanne Bilel, Nikaein Navid, and Mohamed Said Mosli Bouksiaa. “Hybrid CPU-GPU Distributed Framework for Large Scale Mobile Networks Simulation”. In: *2012 IEEE/ACM 16th International Symposium on Distributed Simulation and Real Time Applications*. 2012, pp. 44–53. DOI: [10.1109/DS-RT.2012.15](https://doi.org/10.1109/DS-RT.2012.15).
- [4] Randal Bryant. “Simulation of packet communications architecture computer systems. MIT-LCS-TR-188”. In: (Jan. 1977).
- [5] Laurent Capocchi. *DEVSImPy*. <https://github.com/capocchi/DEVSImPy>. 2019.
- [6] Henri Casanova et al. “Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms”. In: *Journal of Parallel and Distributed Computing* 74.10 (June 2014), pp. 2899–2917. URL: <http://hal.inria.fr/hal-01017319>.
- [7] K.M. Chandy and J. Misra. “Distributed Simulation: A Case Study in Design and Verification of Distributed Programs”. In: *IEEE Transactions on Software Engineering* SE-5.5 (1979), pp. 440–452. DOI: [10.1109/TSE.1979.230182](https://doi.org/10.1109/TSE.1979.230182).
- [8] A.C.H. Chow and B.P. Zeigler. “Parallel DEVS: a parallel, hierarchical, modular modeling formalism”. In: *Proceedings of Winter Simulation Conference*. 1994, pp. 716–722. DOI: [10.1109/WSC.1994.717419](https://doi.org/10.1109/WSC.1994.717419).
- [9] Judith S. Dahmann, Richard M. Fujimoto, and Richard M. Weatherly. “The Department of Defense High Level Architecture”. In: *Proceedings of the 29th Conference on Winter Simulation*. WSC ’97. Atlanta, Georgia, USA: IEEE Computer Society, 1997, pp. 142–149. ISBN: 078034278X. DOI: [10.1145/268437.268465](https://doi.org/10.1145/268437.268465). URL: <https://doi.org/10.1145/268437.268465>.
- [10] Sarah El Kazdadi. *faer-rs*. <https://github.com/sarah-quinones/faer-rs>. URL: <https://github.com/sarah-quinones/faer-rs>.
- [11] R.M. Fujimoto. “Parallel and distributed simulation systems”. In: *Proceeding of the 2001 Winter Simulation Conference (Cat. No.01CH37304)*. Vol. 1. 2001, 147–157 vol.1. DOI: [10.1109/WSC.2001.977259](https://doi.org/10.1109/WSC.2001.977259).
- [12] R.M. Fujimoto, J.-J. Tsai, and G.C. Gopalakrishnan. “Design and evaluation of the rollback chip: special purpose hardware for Time Warp”. In: *IEEE Transactions on Computers* 41.1 (1992), pp. 68–82. DOI: [10.1109/12.123382](https://doi.org/10.1109/12.123382).

- [13] Distributed Computing Group. *Sinalgo*. <https://github.com/sinalgo/sinalgo>. Accessed: 2024-12-13.
- [14] David R. Jefferson. “Virtual time”. In: *ACM Trans. Program. Lang. Syst.* 7.3 (July 1985), pp. 404–425. ISSN: 0164-0925. DOI: [10.1145/3916.3988](https://doi.org/10.1145/3916.3988). URL: <https://doi.org/10.1145/3916.3988>.
- [15] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. USA: No Starch Press, 2018. ISBN: 1593278284.
- [16] Jason Liu and David Nicol. “Lookahead revisited in wireless network simulations”. In: Jan. 2002, pp. 79–88. DOI: [10.1109/PADS.2002.1004203](https://doi.org/10.1109/PADS.2002.1004203).
- [17] Elizabeth Whitaker Lynch and George F. Riley. “Hardware Supported Time Synchronization in Multi-core Architectures”. In: *2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*. 2009, pp. 88–94. DOI: [10.1109/PADS.2009.19](https://doi.org/10.1109/PADS.2009.19).
- [18] Romolo Marotta, Alessandro Pellegrini, and Philipp Andelfinger. “Follow the Leader: Alternating CPU/GPU Computations in PDES”. In: *Proceedings of the 38th ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. SIGSIM-PADS ’24. Atlanta, GA, USA: Association for Computing Machinery, 2024, pp. 47–51. ISBN: 9798400703638. DOI: [10.1145/3615979.3656056](https://doi.org/10.1145/3615979.3656056). URL: <https://doi.org/10.1145/3615979.3656056>.
- [19] F. Mattern. “Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation”. In: *Journal of Parallel and Distributed Computing* 18.4 (1993), pp. 423–434. ISSN: 0743-7315. DOI: <https://doi.org/10.1006/jpdc.1993.1075>. URL: <https://www.sciencedirect.com/science/article/pii/S0743731583710750>.
- [20] René Mayrhofer et al. “DEVS Simulation of Spiking Neural Networks”. In: 2003. URL: <https://api.semanticscholar.org/CorpusID:7602616>.
- [21] Brendan McMahan and Daniel Ramage. *Federated learning: Collaborative machine learning without centralized training data*. <https://research.google/blog/federated-learning-collaborative-machine-learning-without-centralized-training-data/>. Accessed: 2024-10-05. Apr. 2017.
- [22] Jayadev Misra. “Distributed discrete-event simulation”. en. In: *ACM Computing Surveys* 18.1 (Mar. 1986), pp. 39–65. ISSN: 0360-0300, 1557-7341. DOI: [10.1145/6462.6485](https://doi.org/10.1145/6462.6485). URL: <https://dl.acm.org/doi/10.1145/6462.6485> (visited on 10/28/2024).
- [23] Joseph Oglio et al. “QUANTAS: Quantitative User-friendly Adaptable Networked Things Abstract Simulator”. In: *arXiv preprint arXiv:2205.04930* (2022).
- [24] Kalyan S. Perumalla. “Parallel and Distributed Simulation: Traditional Techniques and Recent Advances”. In: *Proceedings of the 2006 Winter Simulation Conference*. 2006, pp. 84–95. DOI: [10.1109/WSC.2006.323041](https://doi.org/10.1109/WSC.2006.323041).
- [25] *Quantas: Quantitative User-Friendly Adaptable Networked Things Abstract Simulator*. <https://github.com/QuantasSupport/Quantas>.
- [26] George F. Riley. “The Georgia Tech Network Simulator”. In: *Proceedings of the ACM SIGCOMM Workshop on Models, Methods and Tools for Reproducible Network Research*. MoMeTools ’03. Karlsruhe, Germany: Association for Computing Machinery, 2003, pp. 5–12. ISBN: 1581137486. DOI: [10.1145/944773.944775](https://doi.org/10.1145/944773.944775). URL: <https://doi.org/10.1145/944773.944775>.

- [27] Brian Paul Swenson. “Techniques to improve the performance of large-scale discrete-event simulation.” PhD Thesis. Georgia Institute of Technology, Atlanta, GA, USA, 2015. URL: <https://www.academia.edu/download/96036379/SWENSON-DISSERTATION-2015.pdf> (visited on 10/28/2024).
- [28] *The Cargo Book*. <https://doc.rust-lang.org/cargo/>. Accessed: 2025-06-10.
- [29] *The Rust Programming Language*. <https://www.rust-lang.org/>. Accessed: 2025-06-10.
- [30] Daniel Tinoco. *distributed_glm*. https://github.com/dbtnc/distributed_glm. Accessed: 2025-05-29. 2025.
- [31] Daniel Tinoco, Raquel Menezes, and Carlos Baquero. *Distributed Generalized Linear Models: A Privacy-Preserving Approach*. 2025. arXiv: 2503.15287 [stat.CO]. URL: <https://arxiv.org/abs/2503.15287>.
- [32] Yifan Wang et al. “DSP, a Plug-and-Play Process-Based Distributed Algorithm Simulation Platform”. en. In: *Scientific Programming 2022* (Feb. 2022). Ed. by Basilio B. Fraguera, pp. 1–13. ISSN: 1875-919X, 1058-9244. DOI: 10.1155/2022/3886729. URL: <https://www.hindawi.com/journals/sp/2022/3886729/> (visited on 10/30/2024).
- [33] B.P. Zeigler. *Theory of Modelling and Simulation*. A Wiley-Interscience publication. Wiley, 1976. ISBN: 9780471981527. URL: <https://books.google.pt/books?id=M-ZQAAAAMAAJ>.
- [34] Bernard P. Zeigler and Alexander Muzy. “From Discrete Event Simulation to Discrete Event Specified Systems (DEVS)”. In: *IFAC-PapersOnLine*. 20th IFAC World Congress 50.1 (July 2017), pp. 3039–3044. ISSN: 2405-8963. DOI: 10.1016/j.ifacol.2017.08.672. URL: <https://www.sciencedirect.com/science/article/pii/S2405896317310601> (visited on 11/02/2024).
- [35] Bernard P. Zeigler, Alexandre Muzy, and Ernesto Kofman. “Chapter 6 - Basic Formalisms: DEVS, DESS, DTSS”. In: *Theory of Modeling and Simulation (Third Edition)*. Ed. by Bernard P. Zeigler, Alexandre Muzy, and Ernesto Kofman. Third Edition. Academic Press, 2019, pp. 153–165. ISBN: 978-0-12-813370-5. DOI: <https://doi.org/10.1016/B978-0-12-813370-5.00014-6>. URL: <https://www.sciencedirect.com/science/article/pii/B9780128133705000146>.
- [36] Junlan Zhou et al. “MAYA: Integrating hybrid network modeling to the physical world”. In: *ACM Trans. Model. Comput. Simul.* 14.2 (Apr. 2004), pp. 149–169. ISSN: 1049-3301. DOI: 10.1145/985793.985796. URL: <https://doi.org/10.1145/985793.985796>.

Appendix A

Designing a Discrete Event Simulator

A.1 Simulator Structure

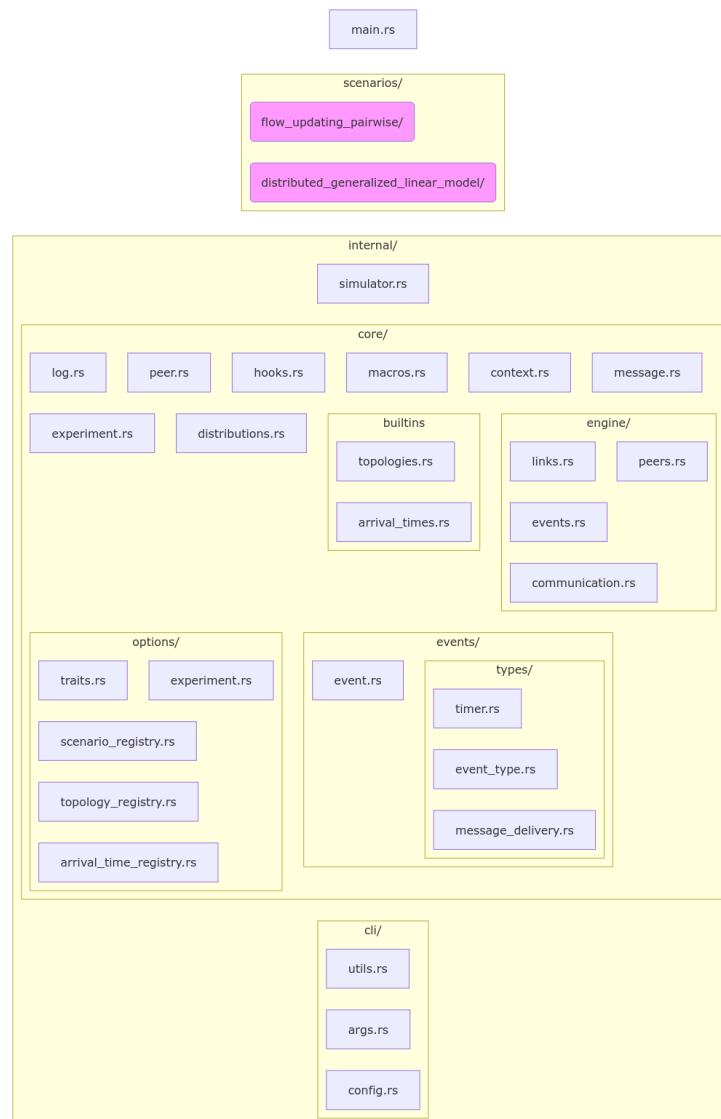


Figure A.1: The created simulator's structure

A.2 Code Definitions

```

1 pub struct Context {
2     event_id: u64,
3     event_q: BinaryHeap<Reverse<EventType>>,
4     pub clock: OrderedFloat<f64>,
5     pub peers: Vec<Box<dyn CustomPeer>>,
6     // Rust's HashMap is non-deterministic.
7     pub links: Vec<IndexMap<usize, LinkInfo>>,
8     pub rng: ChaCha8Rng,
9     pub seed: u64,
10    pub message_delay_cb: MessageDelayCallback,
11    pub logger: Logger,
12    drop_rate: f64,
13    duplicate_rate: f64,
14    jitter: Jitter,
15 }

```

Listing A.1: The Context struct

```

1 pub type CustomOnFinishHook = Box<dyn Fn(&mut Context)>;
2 pub type CustomFinishConditionHook = Box<dyn Fn(&mut Context) -> bool>;
3
4 pub struct SimulationHooks {
5     /// It is called when the simulation finishes.
6     pub on_simulation_finish: CustomOnFinishHook,
7     /// If it returns true, the simulation loop finishes.
8     pub finish_condition: CustomFinishConditionHook,
9 }
10
11 impl Default for SimulationHooks {
12     fn default() -> Self {
13         Self {
14             on_simulation_finish: Box::new(|_ctx| {}),
15             finish_condition: Box::new(|_ctx| false),
16         }
17     }
18 }

```

Listing A.2: The SimulationHooks struct

```

1 // context, sender_id, receiver_id, message
2 type OnMessageReceiveCallback = fn(&mut Context, usize, usize, &dyn Message) -> ();
3
4 #[derive(Clone)]
5 pub struct PeerInfo {
6     id: usize,
7     instantiated: bool,
8     alive: bool,

```

```

9     pub position: (f64, f64, f64),
10     pub on_message_receive: OnMessageReceiveCallback,
11 }

```

Listing A.3: The PeerInfo struct

```

1 #[derive(Debug, Serialize, Deserialize, Clone, Copy)]
2 pub enum LinkKind {
3     Bandwidth(f64),
4     Latency(f64),
5     Full { bandwidth: f64, latency: f64 },
6 }
7
8 pub type LinkInfo = Option<LinkKind>;

```

Listing A.4: The LinkInfo type and the LinkKind enum

```

1 pub trait MessageClone {
2     fn clone_box(&self) -> Box<dyn Message>;
3 }
4
5 impl<T> MessageClone for T
6 where
7     T: 'static + Message + Clone,
8 {
9     fn clone_box(&self) -> Box<dyn Message> {
10         Box::new(self.clone())
11     }
12 }
13
14 pub trait Message: Debug + Downcast + MessageClone {
15     fn size_bits(&self) -> u64 {
16         self.size_bytes() * 8 // Default: convert bytes to bits
17     }
18
19     fn size_bytes(&self) -> u64;
20 }
21 impl_downcast!(Message);

```

Listing A.5: The Message trait

```

1 #[derive(Debug)]
2 pub struct MessageDeliveryEvent {
3     id: u64,
4     timestamp: OrderedFloat<f64>,
5     sender: usize,
6     receiver: usize,
7     message: Box<dyn Message>,

```

```
8 }
```

Listing A.6: The MessageDeliveryEvent struct

```
1 pub trait Timer: Debug + Downcast {
2     fn fire(&self, ctx: &mut Context);
3 }
4 impl_downcast!(Timer);
```

Listing A.7: The Timer trait

```
1 #[derive(Debug)]
2 pub struct TimerEvent {
3     id: u64,
4     timestamp: OrderedFloat<f64>,
5     timer: Box<dyn Timer>,
6 }
```

Listing A.8: The TimerEvent struct

```
1 #[derive(Default)]
2 pub struct Simulator {
3     pub scenario_registry: ScenarioRegistry,
4     pub topology_registry: TopologyRegistry,
5     pub arrival_time_registry: ArrivalTimeRegistry,
6 }
```

Listing A.9: The Simulator struct

```
1 // Type alias for scenario functions
2 type ScenarioFn = fn(&mut Context, &Simulator, ExperimentOptions);
3
4 pub struct ScenarioRegistry {
5     scenarios: IndexMap<String, (ScenarioFn, &'static str)>,
6 }
```

Listing A.10: The ScenarioRegistry struct

```
1 pub trait Scenario {
2     fn name() -> &'static str
3     where
4         Self: Sized;
```

```

5
6     fn description() -> &'static str
7     where
8         Self: Sized;
9
10    fn start(ctx: &mut Context, simulator: &Simulator, opts: ExperimentOptions);
11 }

```

Listing A.11: The Scenario trait

```

1 // from, to, option(bandwidth or latency)
2 pub type ConnectionInfo = (usize, usize, LinkInfo);
3
4 // Type alias for topology functions
5 type TopologyFn = fn(&mut Context, usize, Option<Vec<ConnectionInfo>>);
6
7 pub struct TopologyRegistry {
8     topologies: IndexMap<String, TopologyFn>,
9 }

```

Listing A.12: The TopologyRegistry struct

```

1 pub trait Topology {
2     fn name() -> &'static str
3     where
4         Self: Sized;
5
6     fn connect(
7         ctx: &mut Context,
8         n_peers: usize,
9         custom_list: Option<Vec<ConnectionInfo>>
10    );
11 }

```

Listing A.13: The Topology trait

```

1 pub type MessageDelayCallback = fn(
2     &mut Context,
3     usize,
4     usize
5 ) -> Option<OrderedFloat<f64>>;
6
7 pub struct ArrivalTimeRegistry {
8     callbacks: IndexMap<String, MessageDelayCallback>,
9 }

```

Listing A.14: The ArrivalTimeRegistry struct

```

1 pub trait ArrivalTimeCallback {
2     fn name() -> &'static str
3     where
4         Self: Sized;
5
6     fn callback(ctx: &mut Context, from: usize, to: usize) -> Option<OrderedFloat<f64>>;
7 }

```

Listing A.15: The ArrivalTimeCallback trait

```

1 pub struct Logger {
2     level: LoggerLevel,
3     log_writer: Option<BufWriter<File>>,
4     log_file_location: String,
5     metrics_writer: Option<BufWriter<File>>,
6     metrics_file_location: String,
7     flush_threshold: usize,
8     log_unflushed_count: usize,
9     quiet: bool,
10 }

```

Listing A.16: The Logger struct

```

1 #[derive(Debug, PartialEq, PartialOrd, Clone, Copy, ValueEnum, Serialize, Deserialize)]
2 pub enum LoggerLevel {
3     Trace = 0,
4     Debug = 1,
5     Info = 2,
6     Warn = 3,
7     Error = 4,
8     Internal = 5,
9 }

```

Listing A.17: The LoggerLevel enum

```

1 #[derive(Debug, Default, Serialize, Deserialize, Clone, Copy)]
2 pub enum DistributionWrapper {
3     #[default]
4     Nothing,
5     Constant(f64), // c
6     Exponential(f64), // lambda
7     Gaussian(f64, f64), // mean, std_dev
8     Uniform(f64, f64), // min, max
9     Weibull(f64, f64), // shape, scale
10    LogNormal(f64, f64), // mu, sigma
11 }

```

Listing A.18: The DistributionWrapper enum

```

1  #[derive(Parser, Debug, Clone)]
2  #[command(
3      version,
4      about = "Discrete Event Simulator",
5      long_about = None,
6      group(
7          ArgGroup::new("run_mode")
8              .required(true)
9              .args(["config", "scenario", "list_scenarios", "list_topologies", "
              list_arrival_times"])
10     )
11 )]
12 pub struct Args {
13     /// Location of the configuration file to use
14     #[arg(short, long)]
15     pub config: Option<String>,
16
17     /// Log verbosity level
18     #[arg(long, value_enum, default_value = "info")]
19     pub logger_level: Option<LogLevel>,
20
21     /// The simulator does not output the logs to the console
22     #[arg(short, long)]
23     pub quiet: bool,
24
25     /// Which scenario should be run
26     #[arg(short, long)]
27     pub scenario: Option<String>,
28
29     /// Which scenarios can be run
30     #[arg(long)]
31     pub list_scenarios: bool,
32
33     /// Names a experiment - can only be used if 'scenario' is set
34     #[arg(long, requires = "scenario")]
35     pub name: Option<String>,
36
37     /// The scenario seed - can only be used if 'scenario' is set
38     #[arg(long, requires = "scenario")]
39     pub seed: Option<String>,
40
41     /// The amount of peers to instantiate - can only be used if 'scenario' is set
42     #[arg(long, requires = "scenario", default_value = "5")]
43     pub n_peers: Option<usize>,
44
45     /// Sets the message drop rate (a float in [0.0, 1.0]) - can only be used if '
        scenario' is set
46     #[arg(long, requires = "scenario")]
47     pub drop_rate: Option<f64>,
48

```

```

49     /// Sets the message duplicate rate (a float in [0.0, 1.0]) - can only be used if '
        scenario' is set
50     #[arg(long, requires = "scenario")]
51     pub duplicate_rate: Option<f64>,
52
53     /// The topology to use, must be registered in the simulator - can only be used if '
        scenario' is set
54     #[arg(long, requires = "scenario")]
55     pub topology: Option<String>,
56
57     /// If set to true, uses the jitter configuration Weibull(1.064, 2.872) * 0.001 -
        can only be used if 'scenario' is set
58     #[arg(long, requires = "scenario")]
59     pub use_jitter: bool,
60
61     /// Sets the amount of times an experiment should be run - can only be used if '
        scenario' is set
62     #[arg(long, requires = "scenario", default_value = "1")]
63     pub repetitions: u64,
64
65     /// Which topologies can be selected
66     #[arg(long)]
67     pub list_topologies: bool,
68
69     /// The arrival time callback to use, must be registered in the simulator - can only
        be used if 'scenario' is set
70     #[arg(long, requires = "scenario")]
71     pub arrival_time: Option<String>,
72
73     /// Which arrival time callbacks can be selected
74     #[arg(long)]
75     pub list_arrival_times: bool,
76
77     /// Optional value which tells the simulator when to stop (it may stop earlier if
        there are no events to process)
78     #[arg(long)]
79     pub deadline: Option<f64>,
80
81     /// The amount of logs needed to flush to file
82     #[arg(long, default_value = "200")]
83     pub flush_threshold: usize,
84
85     /// Where the configuration and logs should be stored (prints to console if not
        specified)
86     #[arg(short, long, requires = "scenario")]
87     pub dir: Option<String>,
88
89     /// Overwrites the provided a configuration file
90     #[arg(short, long, requires = "config")]
91     pub write_config: bool,
92 }

```

Listing A.19: The Args struct

```
1 #[derive(Debug, Serialize, Deserialize)]
2 pub struct Experiment {
3     pub name: String,
4     pub scenario: String,
5     pub seed: Option<u64>,
6     pub arrival_time: Option<String>,
7     pub topology: TopologyInfo,
8     pub drop_rate: Option<f64>,
9     pub duplicate_rate: Option<f64>,
10    pub jitter: Option<Jitter>,
11    pub deadline: Option<f64>,
12    pub extra_args: Option<Value>,
13    pub repetitions: Option<u64>,
14 }
```

Listing A.20: The Experiment struct

```
1 #[derive(Debug)]
2 pub struct ExperimentOptions {
3     pub topology: TopologyInfo,
4     pub arrival_time: Option<String>,
5     pub deadline: Option<f64>,
6     pub extra_args: Option<Value>,
7 }
```

Listing A.21: The ExperimentOptions struct

Appendix B

Experimental Evaluation

B.1 Simulator Validation

B.1.1 Message Passing Scenario

B.1.1.1 Implementation: Designed Simulator

```
1 pub struct SimpleMessageScenario {}
2
3 impl Scenario for SimpleMessageScenario {
4     fn name() -> &'static str
5     where
6         Self: Sized,
7     {
8         "simple_message"
9     }
10
11     fn description() -> &'static str
12     where
13         Self: Sized,
14     {
15         "An implementation used to evaluate the simulator."
16     }
17
18     fn start(ctx: &mut Context, simulator: &Simulator, opts: ExperimentOptions) {
19         for i in 0..opts.topology.n_peers {
20             let (pos_x, pos_y) = match opts.topology.positions.as_ref().and_then(|v| v.
21                 get(i)) {
22                 Some(&(px, py, _)) => (px, py),
23                 None => (0.0, 0.0),
24             };
25             engine::add_peer(ctx, SimplePeer::new(pos_x, pos_y));
26         }
27
28         simulator
29             .topology_registry
30             .connect_peers(ctx, opts.topology);
```

```

31     ctx.message_delay_cb = simulator
32         .arrival_time_registry
33         .get_callback(opts.arrival_time);
34
35     let message_size = opts
36         .extra_args
37         .as_ref()
38         .and_then(|custom| custom.get("msg_size"))
39         .and_then(|v| v.as_u64())
40         .unwrap_or(0);
41
42     // start event
43     engine::add_timer(ctx, OrderedFloat(0.0), StartTimer { message_size });
44
45     engine::run(ctx, &SimulationHooks::default(), opts.deadline);
46 }
47 }

```

Listing B.1: The SimpleMessageScenario struct

```

1 pub struct SimplePeer {
2     pub peer_info: PeerInfo,
3 }
4
5 define_custom_peer!(SimplePeer);
6
7 impl SimplePeer {
8     pub fn new(x: f64, y: f64) -> Self {
9         Self {
10             peer_info: PeerInfo::new(x, y, 0.0)
11                 .with_on_message_receive(simple_peer_on_message_receive),
12         }
13     }
14 }

```

Listing B.2: The SimplePeer struct and its implementation

```

1 pub fn simple_peer_on_message_receive(
2     ctx: &mut Context,
3     _sender_id: usize,
4     receiver_id: usize,
5     msg: &dyn Message,
6 ) {
7     let _peer: &mut SimplePeer =
8         get_peer_of_type!(ctx, receiver_id, SimplePeer).expect("peer should exist");
9
10    if let Some(_example_msg) = msg.downcast_ref::<EmptyMessage>() {
11        log::info(ctx, format!("Peer {receiver_id} received a message"));
12    }
13 }

```

Listing B.3: SimplePeer's callback when receiving a message

```

1  #[derive(Debug, Clone)]
2  pub struct StartTimer {
3      pub message_size: u64,
4  }
5
6  impl Timer for StartTimer {
7      fn fire(&self, ctx: &mut Context) {
8          if let Some(neighbors) = engine::get_neighbors_alive(ctx, 0) {
9              for neigh_id in neighbors {
10                 log::info(ctx, format!("Peer 0 sent message to {neigh_id}"));
11                 engine::send_message_to(
12                     ctx,
13                     0,
14                     neigh_id,
15                     EmptyMessage {
16                         size: self.message_size,
17                     },
18                 );
19             }
20         }
21     }
22 }

```

Listing B.4: The StartTimer struct

```

1  #[derive(Debug, Clone)]
2  pub struct EmptyMessage {
3      pub size: u64,
4  }
5
6  impl Message for EmptyMessage {
7      fn size_bytes(&self) -> u64 {
8          self.size
9      }
10 }

```

Listing B.5: The EmptyMessage struct and its implementation

B.1.1.2 Implementation: SimGrid

```

1  if __name__ == "__main__":
2      e = Engine(sys.argv)
3

```

```

4     scenario = sys.argv[1]
5
6     e.register_actor("sender", sender)
7     e.register_actor("receiver", receiver)
8
9     e.load_platform(f"./platforms/{scenario}.xml")
10    e.load_deployment(f"./actors/{scenario}.xml")
11
12    e.run()
13
14    this_actor.info("Simulation finished")

```

Listing B.6: SimGrid's implementation entry point

```

1 def sender(targets: str = "", msg_size: str = "0"):
2     pending_messages = ActivitySet()
3     this_actor.info("Sender started.")
4
5     for mailbox_name in targets.split(","):
6         msg = "Message content"
7         this_actor.info(f"Sending message to Mailbox({mailbox_name}): {msg}")
8         if comm := Mailbox.by_name(mailbox_name).put_async(msg, int(msg_size)):
9             pending_messages.push(comm)
10
11    pending_messages.wait_all()

```

Listing B.7: The sender actor method

```

1 def receiver(wait_for: int = 1):
2     host_name = this_actor.get_host().name
3     mailbox = Mailbox.by_name(host_name)
4
5     this_actor.info(f"Receiver started with Mailbox({host_name})")
6
7     for _ in range(int(wait_for)):
8         this_actor.info(f"Received: {mailbox.get()}")

```

Listing B.8: The receiver actor method

B.1.2 Flow Updating Scenario

B.1.2.1 Implementation: Designed Simulator

```

1 pub struct FlowUpdatingPairwise {}
2
3 impl Scenario for FlowUpdatingPairwise {

```

```

4   fn name() -> &'static str
5   where
6       Self: Sized,
7   {
8       "flow_updating_pairwise"
9   }
10
11  fn description() -> &'static str
12  where
13      Self: Sized,
14  {
15      "An implementation of the flow updating pairwise algorithm."
16  }
17
18  fn start(ctx: &mut Context, simulator: &Simulator, opts: ExperimentOptions) {
19      let n_peers = opts.topology.n_peers;
20
21      let peer_values: Vec<i32> = opts
22          .extra_args
23          .as_ref()
24          .and_then(|custom| custom.get("values"))
25          .and_then(|v| v.as_sequence())
26          .map(|seq| {
27              seq.iter()
28                  .filter_map(|val| val.as_f64().map(|f| f as i32))
29                  .collect()
30          })
31          .unwrap_or_default();
32
33      for i in 0..n_peers {
34          let (rx, ry) = (
35              ctx.rng.random_range(-100000.0..=100000.0),
36              ctx.rng.random_range(-100000.0..=100000.0),
37          );
38
39          let val: i32 = *peer_values.get(i).unwrap_or(&ctx.rng.random_range(0..=80));
40
41          engine::add_peer(ctx, FlowUpdatingPairwisePeer::new(rx, ry, val));
42      }
43
44      simulator
45          .topology_registry
46          .connect_peers(ctx, opts.topology);
47      ctx.message_delay_cb = simulator
48          .arrival_time_registry
49          .get_callback(opts.arrival_time);
50
51      let mut hooks = SimulationHooks::default();
52      hooks.set_on_simulation_finish_hook(Box::new(hooks::on_simulation_finish_hook));
53
54      let tick_interval = 0.001;
55      let metrics_interval = 0.001;
56
57      // init
58      for peer_id in 0..ctx.peers.len() {
59          engine::add_timer(ctx, OrderedFloat(0.0), StartTimer { peer_id });
60      }

```

```

61
62     engine::add_timer(
63         ctx,
64         OrderedFloat(tick_interval),
65         TickTimer {
66             interval: tick_interval,
67         },
68     );
69
70     engine::add_timer(
71         ctx,
72         OrderedFloat(0.0),
73         MetricsTimer {
74             interval: metrics_interval,
75         },
76     );
77
78     engine::run(ctx, &hooks, opts.deadline);
79 }
80 }

```

Listing B.9: The FlowUpdatingPairwise struct

```

1 pub struct FlowUpdatingPairwisePeer {
2     pub peer_info: PeerInfo,
3     pub value: i32,
4     pub flows: HashMap<usize, f64>,
5     pub estimates: HashMap<usize, f64>,
6     pub ticks_since_last_avg: HashMap<usize, u32>,
7     pub last_avg: f64,
8 }
9
10 define_custom_peer!(FlowUpdatingPairwisePeer);
11
12 impl FlowUpdatingPairwisePeer {
13     pub fn new(x: f64, y: f64, value: i32) -> Self {
14         Self {
15             peer_info: PeerInfo::new(x, y, 0.0)
16                 .with_on_message_receive(callbacks::example_on_message_receive),
17             value,
18             flows: HashMap::new(),
19             estimates: HashMap::new(),
20             ticks_since_last_avg: HashMap::new(),
21             last_avg: f64::from(value),
22         }
23     }
24 }

```

Listing B.10: The FlowUpdatingPairwisePeer struct and its implementation

```

1 pub fn on_message_receive(

```

```

2   ctx: &mut Context,
3   sender_id: usize,
4   receiver_id: usize,
5   msg: &dyn Message,
6 ) {
7   log::trace(
8     ctx,
9     format!("Peer_{receiver_id} received message from Peer_{sender_id}"),
10  );
11
12  let peer: &mut FlowUpdatingPairwisePeer =
13    get_peer_of_type!(ctx, receiver_id, FlowUpdatingPairwisePeer).expect("peer
14      should exist");
15
16  if let Some(example_msg) = msg.downcast_ref::<FlowUpdatingPairwiseMessage>() {
17    peer.estimates
18      .insert(example_msg.sender, example_msg.estimate);
19    peer.flows.insert(example_msg.sender, -example_msg.flow);
20
21    algorithms::avg_and_send(ctx, receiver_id, example_msg.sender);
22  }
23 }

```

Listing B.11: The FlowUpdatingPairwisePeer's on message receive callback

```

1  #[derive(Debug, Clone)]
2  pub struct FlowUpdatingPairwiseMessage {
3      pub sender: usize,
4      pub flow: f64,
5      pub estimate: f64,
6  }
7  impl Message for FlowUpdatingPairwiseMessage {
8      fn size_bytes(&self) -> u64 {
9          1 + 8 + 8
10     }
11 }

```

Listing B.12: The FlowUpdatingPairwiseMessage struct

```

1  pub fn avg_and_send(ctx: &mut Context, peer_id: usize, neigh_id: usize) {
2      log::trace(
3          ctx,
4          format!("Peer_{peer_id} avg_and_send to Peer_{neigh_id}"),
5      );
6
7      if let Some(neighbors) = engine::get_neighbors_alive(ctx, peer_id) {
8          let peer: &mut FlowUpdatingPairwisePeer =
9              get_peer_of_type!(ctx, peer_id, FlowUpdatingPairwisePeer).expect("peer
10                  should exist");
11
12          let flows_sum: f64 = neighbors

```

```

12         .into_iter()
13         .map(|idx| *peer.flows.get(&idx).unwrap_or(&0.0))
14         .sum();
15     let estimate = f64::from(peer.value) - flows_sum;
16     let avg = (peer.estimates.get(&neigh_id).copied().unwrap_or(0.0) + estimate) /
17         2.0;
18     peer.last_avg = avg;
19     peer.flows.insert(
20         neigh_id,
21         peer.flows.get(&neigh_id).copied().unwrap_or(0.0) + avg
22         - peer.estimates.get(&neigh_id).copied().unwrap_or(0.0),
23     );
24     peer.estimates.insert(neigh_id, avg);
25
26     peer.ticks_since_last_avg.insert(neigh_id, 0);
27
28     let payload = FlowUpdatingPairwiseMessage {
29         sender: peer_id,
30         flow: peer.flows.get(&neigh_id).copied().unwrap_or(0.0),
31         estimate: avg,
32     };
33
34     log::trace(
35         ctx,
36         format!("Peer_{peer_id} flows_sum {flows_sum}"),
37     );
38     log::trace(
39         ctx,
40         format!("Peer_{peer_id} estimate {estimate}"),
41     );
42     log::trace(
43         ctx,
44         format!("Peer_{peer_id} new last_avg {avg}"),
45     );
46
47     engine::send_message_to(ctx, peer_id, neigh_id, payload);
48 }
49 }

```

Listing B.13: The avg_and_send function

```

1 pub fn on_simulation_finish_hook(ctx: &mut Context) {
2     let avgs: Vec<OrderedFloat<f64>> = ctx
3         .peers
4         .iter()
5         .filter_map(|p| {
6             p.downcast_ref::<FlowUpdatingPairwisePeer>()
7                 .map(|peer| OrderedFloat(peer.last_avg))
8         })
9         .collect();
10
11     let real_total: Vec<f64> = ctx
12         .peers

```

```

13     .iter()
14     .filter_map(|p| {
15         p.downcast_ref::<FlowUpdatingPairwisePeer>()
16         .map(|peer| f64::from(peer.value))
17     })
18     .collect();
19
20     let total: OrderedFloat<f64> = avgs.iter().sum();
21
22     log::info(ctx, format!("initial values {real_total:?}"));
23     log::info(
24         ctx,
25         format!("The resulting averages are the following {avgs:?}"),
26     );
27     log::info(
28         ctx,
29         format!(
30             "should_be: {:?}",
31             real_total.iter().sum::<f64>() / real_total.len() as f64,
32         ),
33     );
34     log::info(ctx, format!("avg: {:?}", total / avgs.len() as f64,));
35     log::info(ctx, format!("max: {:?}", avgs.iter().max().unwrap(),));
36     log::info(ctx, format!("min: {:?}", avgs.iter().min().unwrap(),));
37 }

```

Listing B.14: The FlowUpdatingPairwise scenario's hook called on finish

```

1  #[derive(Debug, Clone)]
2  pub struct StartTimer {
3      pub peer_id: usize,
4  }
5
6  impl Timer for StartTimer {
7      fn fire(&self, ctx: &mut Context) {
8          if let Some(neighbors) = engine::get_neighbors_alive(ctx, self.peer_id) {
9              for neigh_id in neighbors {
10                 log::info(
11                     ctx,
12                     format!("Peer_{} sending to Peer_{neigh_id}", self.peer_id),
13                 );
14
15                 let peer = get_peer_of_type!(ctx, self.peer_id, FlowUpdatingPairwisePeer)
16                     .expect("peer should exist");
17
18                 let payload = FlowUpdatingPairwiseMessage {
19                     sender: self.peer_id,
20                     flow: 0.0,
21                     estimate: f64::from(peer.value),
22                 };
23
24                 engine::send_message_to(ctx, self.peer_id, neigh_id, payload);
25             }
26         }
27     }
28 }

```

```

26     }
27 }

```

Listing B.15: The StartTimer struct and its implementation

```

1  #[derive(Debug, Clone)]
2  pub struct TickTimer {
3      pub interval: f64,
4  }
5
6  impl Timer for TickTimer {
7      fn fire(&self, ctx: &mut Context) {
8          log::trace(ctx, "TickTimer fired");
9          engine::add_timer(ctx, ctx.clock + self.interval, self.clone());
10
11          // call tick for every peer
12          for peer_id in 0..ctx.peers.len() {
13              algorithms::tick(ctx, peer_id);
14          }
15      }
16 }

```

Listing B.16: The TickTimer struct and its implementation

```

1  const TICKS: u32 = 50;
2
3  pub fn tick(ctx: &mut Context, peer_id: usize) {
4      let peer: &mut FlowUpdatingPairwisePeer =
5          get_peer_of_type!(ctx, peer_id, FlowUpdatingPairwisePeer).expect("peer should
6              exist");
7
8      if !peer.is_alive() {
9          return;
10     }
11
12     if let Some(neighbors) = engine::get_neighbors(ctx, peer_id) {
13         for neigh_id in neighbors {
14             let peer: &mut FlowUpdatingPairwisePeer =
15                 get_peer_of_type!(ctx, peer_id, FlowUpdatingPairwisePeer)
16                     .expect("peer should exist");
17
18             let neigh_ticks = peer
19                 .ticks_since_last_avg
20                 .get(&neigh_id)
21                 .copied()
22                 .unwrap_or(0);
23
24             if neigh_ticks > TICKS {
25                 avg_and_send(ctx, peer_id, neigh_id);
26             } else {
27                 peer.ticks_since_last_avg.insert(neigh_id, neigh_ticks + 1);
28             }
29         }
30     }
31 }

```

```

27         }
28     }
29 }
30 }

```

Listing B.17: The tick function called by the TickTimer

```

1  #[derive(Debug, Clone)]
2  pub struct MetricsTimer {
3      pub interval: f64,
4  }
5
6  impl Timer for MetricsTimer {
7      fn fire(&self, ctx: &mut Context) {
8          engine::add_timer(ctx, ctx.clock + self.interval, self.clone());
9
10         let mut avgs_metric = json!({});
11
12         // call tick for every peer
13         for peer_id in 0..ctx.peers.len() {
14             if let Ok(peer) = get_peer_of_type!(ctx, peer_id, FlowUpdatingPairwisePeer)
15             {
16                 avgs_metric[peer_id.to_string()] = peer.last_avg.into();
17             }
18
19             log::metrics(ctx, "last_averages", &avgs_metric);
20         }
21     }

```

Listing B.18: The MetricsTimer struct and its implementation

B.1.2.2 Implementation: SimGrid

```

1  global_values = defaultdict(dict)
2  log_file = None
3
4  if __name__ == "__main__":
5      e = Engine(sys.argv)
6
7      scenario = sys.argv[1]
8      should_stop = float(sys.argv[2])
9      metrics_interval = 0.001
10
11      e.register_actor("peer", Peer)
12
13      e.load_platform(f"./platforms/{scenario}.xml")
14      e.load_deployment(f"./actors/{scenario}.xml")
15
16      e.netzone_root.add_host("observer", 25e6)

```

```

17
18     log_file = open(f"{scenario}.jsonl", "w")
19
20     Actor.create(
21         "watcher", Host.by_name("observer"), watcher, should_stop, metrics_interval
22     )
23
24     e.run()
25
26     this_actor.info("Simulation finished")

```

Listing B.19: SimGrid's Flow Updating pairwise version implementation entry point

```

1 class Peer:
2     TICK_INTERVAL = 0.001
3     TICK_TIMEOUT = 50
4
5     def __init__(self, value: str, neighbors_ids_str: str = ""):
6         self.name = this_actor.get_host().name
7         self.value = float(value)
8         neighbors_ids = []
9         if len(neighbors_ids_str):
10             neighbors_ids = neighbors_ids_str.split(",")
11
12         self.flows = defaultdict(float)
13         self.estimateds = defaultdict(float)
14         self.ticks_since_last_avg = defaultdict(float)
15         self._last_avg = 0.0
16         self.neighbors = dict()
17         for name in neighbors_ids:
18             self.neighbors[name] = Mailbox.by_name(f"{name}_from_{self.name}")
19
20         # setup mailbox
21         self.mailbox = Mailbox.by_name(self.name)
22
23         self.pending_comms = ActivitySet()
24
25     @property
26     def value(self):
27         return self._value
28
29     @value.setter
30     def value(self, val):
31         global_values["last_avg"][self.name] = val
32         global_values["value"][self.name] = val
33         self._value = val
34
35     @property
36     def last_avg(self):
37         return self._last_avg
38
39     @last_avg.setter
40     def last_avg(self, val):
41         global_values["last_avg"][self.name] = val

```

```

42     this_actor.info(f"{self.name} new last_avg {val}")
43     self._last_avg = val
44
45     # this is called right away
46     def __call__(self):
47         this_actor.info(
48             f"Peer {self.name} with value {self.value} started {self.neighbors}."
49         )
50
51         Actor.create(f"{self.name}_tick_handler", this_actor.get_host(), self.tick_loop)
52         self.init()
53         self.message_loop()
54
55     def message_loop(self):
56         boxes = [Mailbox.by_name(f"{self.name}_from_{i}") for i in self.neighbors]
57         comms = [box.get_async() for box in boxes]
58         aset = ActivitySet()
59         for comm in comms:
60             aset.push(comm)
61
62         while True:
63             try:
64                 done_comm = aset.wait_any()
65                 msg = done_comm.get_payload()
66
67                 if isinstance(msg, FlowUpdatingMsg):
68                     self.on_receive(msg)
69
70                 idx = comms.index(done_comm)
71                 new_comm = boxes[idx].get_async()
72                 comms[idx] = new_comm
73                 aset.push(new_comm)
74
75             except Exception as e:
76                 if "Actor killed" not in str(e):
77                     this_actor.error(f"{self.name}: Error in message loop: {e}")
78
79     def tick_loop(self):
80         while True:
81             this_actor.sleep_for(self.TICK_INTERVAL)
82             self.tick()
83
84     def init(self):
85         for neigh in self.neighbors.keys():
86             payload = FlowUpdatingMsg(self.name, 0.0, self.value)
87
88             this_actor.info(f"{self.name} sending to {neigh}")
89
90             if comm := self.neighbors[neigh].put_async(payload, payload.size()):
91                 self.pending_comms.push(comm)
92
93     def tick(self):
94         for neigh in self.neighbors.keys():
95             if self.ticks_since_last_avg[neigh] > Peer.TICK_TIMEOUT:
96                 self.avg_and_send(neigh)
97             else:
98                 self.ticks_since_last_avg[neigh] += 1

```

```

99
100     def on_receive(self, msg: FlowUpdatingMsg):
101         this_actor.info(f"{self.name} received message from {msg.sender}")
102
103         if msg.sender not in self.neighbors.keys():
104             self.neighbors[msg.sender] = Mailbox.by_name(msg.sender)
105             this_actor.error(f"{msg.sender} is not {self.name}'s neighbor")
106
107         self.estimate[msg.sender] = msg.estimate
108         self.flows[msg.sender] = -msg.flow
109         self.avg_and_send(msg.sender)
110
111     def avg_and_send(self, neigh: str):
112         this_actor.info(f"{self.name} avg_and_send to {neigh}")
113
114         flows_sum = sum(map(lambda x: self.flows[x], self.neighbors.keys()))
115         estimate = self.value - flows_sum
116         avg = (self.estimate[neigh] + estimate) / 2.0
117
118         this_actor.info(f"{self.name} flows_sum {flows_sum}")
119         this_actor.info(f"{self.name} estimate {estimate}")
120         self.last_avg = avg
121         self.flows[neigh] = self.flows[neigh] + avg - self.estimate[neigh]
122         self.estimate[neigh] = avg
123
124         self.ticks_since_last_avg[neigh] = 0
125
126         payload = FlowUpdatingMsg(self.name, self.flows[neigh], avg)
127         # FIXME: pending comms just keeps growing because detaching a comm gives
128         # Segmentation Fault
129         if comm := self.neighbors[neigh].put_async(payload, payload.size()):
130             self.pending_comms.push(comm)

```

Listing B.20: The Peer class

```

1 def watcher(run_until: float, time_interval: float = 10.0):
2     while Engine.clock < run_until:
3         this_actor.sleep_for(min(time_interval, run_until - Engine.clock))
4         print_global_values()
5
6     this_actor.info("Killing every actor but myself.")
7     Actor.kill_all()
8
9 def print_global_values():
10     if log_file is not None:
11         json_line = json.dumps(
12             {
13                 "title": "last_averages",
14                 "timestamp": Engine.clock,
15                 # Remove Peer_ prefix
16                 "metrics": {
17                     key.split("_")[-1]: value
18                     for key, value in global_values["last_avg"].items()
19                 },

```

```
20         }
21     )
22     log_file.write(json_line + "\n")
23     log_file.flush()
```

Listing B.21: The watcher method

```
1 @dataclass
2 class FlowUpdatingMsg:
3     sender: str
4     flow: float
5     estimate: float
6
7     def size(self) -> int:
8         return 1 + 8 + 8
```

Listing B.22: The FlowUpdatingPairwiseMessage dataclass

B.1.2.3 Results

```
1 @dataclass
2 class Entry:
3     title: str
4     timestamp: float
5     metrics: object
6
7
8 def read_jsonl(filename: str):
9     title_filter = "last_averages"
10    entries = []
11
12    with open(filename) as f:
13        for line in f:
14            entry = json.loads(line)
15            if entry.get("title") == title_filter:
16                entries.append(entry)
17
18    entries: [Entry] = list(map(lambda e: Entry(**e), entries))
19
20    for i in range(len(entries)):
21        entries[i].timestamp = round(entries[i].timestamp, 4)
22
23    entries.sort(key=lambda x: x.timestamp)
24
25    return entries
```

Listing B.23: Read metrics logs

```

1 def draw_graph(
2     entries: list = [], filename: str = "", values: list[int] = [], n_peers: int = 1
3 ):
4     plt.close()
5
6     keys_of_interest = [str(i) for i in range(0, n_peers)]
7     metrics_by_key = {k: {"x": [], "y": []} for k in keys_of_interest}
8
9     for entry in entries:
10         for key in keys_of_interest:
11             if key in entry.metrics:
12                 metrics_by_key[key]["x"].append(entry.timestamp)
13                 metrics_by_key[key]["y"].append(entry.metrics[key])
14
15     for key in keys_of_interest:
16         data = metrics_by_key[key]
17         if data["x"]:
18             plt.plot(data["x"], data["y"], label=f"Peer {key}")
19
20     actual_value = sum(values) / n_peers
21
22     plt.axhline(y=actual_value, color="black", linestyle="--", label="Actual Value")
23
24     plt.xlabel("Timestamp")
25     plt.ylabel("Value")
26     plt.title("Average Value Over Time by Peer")
27     plt.legend()
28     plt.grid(True)
29     plt.tight_layout()
30
31     plt.savefig(filename, bbox_inches="tight")

```

Listing B.24: Draws a graph from the average values by peer

```

1 def draw_global_diff_graph(
2     entries: list = [], filename: str = "", values: list[int] = [], n_peers: int = 1
3 ):
4     plt.close()
5
6     keys_of_interest = [str(i) for i in range(0, 7)]
7     metrics_by_key = {"x": [], "y": []}
8
9     for entry in entries:
10         tot = 0
11         n = 0
12         for key in keys_of_interest:
13             if key in entry.metrics:
14                 tot += entry.metrics[key]
15                 n += 1
16
17         metrics_by_key["x"].append(entry.timestamp)
18         metrics_by_key["y"].append(tot / n)
19

```

```

20     actual_value = sum(values) / n_peers
21
22     if metrics_by_key["x"]:
23         plt.plot(
24             metrics_by_key["x"],
25             list(map(lambda x: abs(x - actual_value), metrics_by_key["y"])),
26             label="All Peers",
27         )
28
29     plt.axhline(y=0.0, color="black", linestyle="--", label="Actual Value")
30
31     plt.xlabel("Timestamp")
32     plt.ylabel("Value")
33     plt.title("Global Difference Between Average Value And Actual Value Over Time")
34     plt.legend()
35     plt.grid(True)
36     plt.tight_layout()
37
38     plt.savefig(filename, bbox_inches="tight")

```

Listing B.25: Draws a graph from the difference between the total and the actual average value

```

1 - name: flow_updating_0_2
2   scenario: flow_updating_pairwise
3   topology:
4     n_peers: 7
5     name: twowaycustom
6     connections:
7       - [0, 1, !Full {bandwidth: 80000, latency: 0.02} ]
8       - [0, 2, !Full {bandwidth: 80000, latency: 0.03604} ]
9       - [0, 3, !Full {bandwidth: 80000, latency: 0.04768} ]
10      - [0, 4, !Full {bandwidth: 80000, latency: 0.04768} ]
11      - [0, 5, !Full {bandwidth: 80000, latency: 0.03604} ]
12      - [0, 6, !Full {bandwidth: 80000, latency: 0.02} ]
13      - [1, 2, !Full {bandwidth: 80000, latency: 0.02} ]
14      - [1, 3, !Full {bandwidth: 80000, latency: 0.03604} ]
15      - [1, 4, !Full {bandwidth: 80000, latency: 0.04768} ]
16      - [1, 5, !Full {bandwidth: 80000, latency: 0.04768} ]
17      - [1, 6, !Full {bandwidth: 80000, latency: 0.03604} ]
18      - [2, 3, !Full {bandwidth: 80000, latency: 0.02} ]
19      - [2, 4, !Full {bandwidth: 80000, latency: 0.03604} ]
20      - [2, 5, !Full {bandwidth: 80000, latency: 0.04768} ]
21      - [2, 6, !Full {bandwidth: 80000, latency: 0.04768} ]
22      - [3, 4, !Full {bandwidth: 80000, latency: 0.02} ]
23      - [3, 5, !Full {bandwidth: 80000, latency: 0.03604} ]
24      - [3, 6, !Full {bandwidth: 80000, latency: 0.04768} ]
25      - [4, 5, !Full {bandwidth: 80000, latency: 0.02} ]
26      - [4, 6, !Full {bandwidth: 80000, latency: 0.03604} ]
27      - [5, 6, !Full {bandwidth: 80000, latency: 0.02} ]
28   deadline: 1.75
29   drop_rate: 0.2
30   extra_args:
31     values: [55, 72, 49, 30, 32, 49, 65]

```

Listing B.26: The designed simulator's faulty experiment configuration

B.2 Statistical Learning Algorithm Evaluation

B.2.1 Implementation

```

1 pub struct DistributedGeneralizedLinearModel;
2
3 impl Scenario for DistributedGeneralizedLinearModel {
4     fn name() -> &'static str {
5         "dglm"
6     }
7
8     fn description() -> &'static str {
9         "A distributed implementation of the generalized linear model."
10    }
11
12    fn start(ctx: &mut Context, simulator: &Simulator, opts: ExperimentOptions) {
13        let n_peers: usize = opts.topology.n_peers;
14
15        let data: ModelData = match model_data("glm") {
16            Ok(d) => d,
17            Err(e) => panic!("Failed to load model_data: {e}"),
18        };
19        let beta = match model_beta("glm") {
20            Ok(b) => b,
21            Err(e) => panic!("Failed to load model_beta: {e}"),
22        };
23        let beta_mat = Mat::from_fn(
24            beta.len(),
25            beta.first().map_or(0, std::vec::Vec::len),
26            |i, j| *beta.get(i).unwrap().get(j).unwrap(),
27        );
28
29        let y_len = data.y.nrows();
30        let ncols = data.x.ncols();
31
32        assert!(data.x.nrows() == y_len, "x.nrows() != y.nrows()");
33        assert!(n_peers * (ncols + 1) < y_len, "split > ncols");
34
35        let x_chunks = chunk_nx(data.x, n_peers);
36        let y_chunks = chunk_nx(data.y, n_peers);
37
38        for (i, (x, y)) in x_chunks.into_iter().zip(y_chunks.into_iter()).enumerate() {
39            let (pos_x, pos_y) = match opts.topology.positions.as_ref().and_then(|v| v.
40                get(i)) {
41                Some(&(px, py, _)) => (px, py),
42                None => (
43                    ctx.rng.random_range(-100.0..=100.0) * 1000.0,
44                    ctx.rng.random_range(-100.0..=100.0) * 1000.0,

```

```

44         ),
45     };
46
47     engine::add_peer(ctx, GlmPeer::new(pos_x, pos_y, x, y));
48 }
49
50 simulator
51     .topology_registry
52     .connect_peers(ctx, opts.topology);
53 ctx.message_delay_cb = simulator
54     .arrival_time_registry
55     .get_callback(opts.arrival_time);
56
57 // init
58 for peer_id in 0..ctx.peers.len() {
59     engine::add_timer(ctx, OrderedFloat(0.0), StartTimer { peer_id });
60 }
61
62 if let Some(custom) = opts.extra_args {
63     if let Some(Value::Bool(true)) = custom.get("kill_peer") {
64         engine::add_timer(ctx, OrderedFloat(0.1), KillTimer::new(0));
65     }
66 }
67
68 let mut hooks = SimulationHooks::default();
69 hooks.set_on_simulation_finish_hook(hooks::on_simulation_finish_hook(beta_mat));
70
71 engine::run(ctx, &hooks, opts.deadline);
72 }
73 }

```

Listing B.27: The DistributedGeneralizedLinearModel struct

```

1 pub fn model_data(m: &str) -> Result<ModelData, Box<dyn Error>> {
2     let path = format!("./resources/generalized_linear_model/{m}_mm.csv");
3     let path = Path::new(&path).canonicalize()?;
4
5     let mut x_data: Vec<Vec<f64>> = Vec::new();
6     let mut y_data: Vec<Vec<f64>> = Vec::new();
7
8     let mut reader = ReaderBuilder::new().has_headers(true).from_path(path)?;
9
10    for result in reader.records() {
11        let record = result?;
12        let float_row: Vec<f64> = record
13            .iter()
14            .map(str::parse::<f64>)
15            .collect::<Result<Vec<f64>, _>>()?;
16
17        if !float_row.is_empty() {
18            let (y, x) = float_row.split_last().unwrap();
19
20            x_data.push(x.to_vec());
21            y_data.push(vec![*y]);

```

```

22     }
23 }
24
25 Ok(ModelData {
26     x: Mat::from_fn(x_data.len(), x_data[0].len(), |i, j| x_data[i][j]),
27     y: Mat::from_fn(y_data.len(), y_data[0].len(), |i, j| y_data[i][j]),
28 })
29 }

```

Listing B.28: The function which parses the input data

```

1 pub fn model_beta(m: &str) -> Result<Vec<Vec<f64>>, Box<dyn Error>> {
2     let path = format!("./resources/generalized_linear_model/{m}_beta.csv");
3     let path = Path::new(&path).canonicalize()?;
4
5     let mut result: Vec<Vec<f64>> = Vec::new();
6
7     let mut reader = ReaderBuilder::new().has_headers(true).from_path(path)?;
8
9     for record_result in reader.records() {
10         let record = record_result?;
11         if let Some(first_field) = record.get(0) {
12             let value: f64 = first_field.parse()?;
13             result.push(vec![value]);
14         }
15     }
16
17     Ok(result)
18 }

```

Listing B.29: The function which parses the input coefficients

```

1 pub fn chunk_nx(mat: Mat<f64>, n: usize) -> Vec<Mat<f64>> {
2     if n == 1 {
3         return vec![mat];
4     }
5
6     let nsplits = mat.nrows() / n;
7     let mut chunks = Vec::new();
8     let mut start_row = 0;
9
10    // Create n-1 chunks of equal size
11    for _ in 0..(n - 1) {
12        let chunk = mat.subrows(start_row, nsplits);
13        chunks.push(chunk.to_owned());
14        start_row += nsplits;
15    }
16
17    // Add the remaining rows as the last chunk
18    let remaining_rows = mat.nrows() - start_row;
19    let last_chunk = mat.subrows(start_row, remaining_rows);

```

```

20     chunks.push(last_chunk.to_owned());
21
22     chunks
23 }

```

Listing B.30: The function which divides the input data

```

1 pub struct GlmState {
2     pub model: GeneralizedLinearModel,
3     pub data: ModelData,
4     pub r_n_rows: HashMap<usize, usize>, // how many rows remotes have
5     pub r_remotes: HashMap<usize, HashMap<usize, Mat<f64>>>,
6     pub total_nrow: usize,
7     pub nodes: Vec<usize>,
8     pub finished: bool,
9 }
10
11 pub struct GlmPeer {
12     pub peer_info: PeerInfo,
13     pub state: GlmState,
14 }
15
16 define_custom_peer!(GlmPeer);
17
18 impl GlmPeer {
19     pub fn new(pos_x: f64, pos_y: f64, x: Mat<f64>, y: Mat<f64>) -> Self {
20         let (r, c) = x.shape();
21         let beta: Mat<f64> = Mat::zeros(c, 1);
22
23         let family: FamilyEnum = FamilyEnum::Binomial;
24
25         let r_local =
26             generalized_linear_model::distributed_single_iter_n(family, &x, &y, beta.
27                 clone());
28
29         let model = GeneralizedLinearModel {
30             r_local,
31             coefficients: beta.clone(),
32             family,
33             iter: 0,
34         };
35
36         Self {
37             peer_info: PeerInfo::new(pos_x, pos_y, 0.0)
38                 .with_on_message_receive(callbacks::on_message_receive),
39             state: GlmState {
40                 model,
41                 data: ModelData { x, y },
42                 r_n_rows: HashMap::new(),
43                 r_remotes: HashMap::new(),
44                 total_nrow: r,
45                 nodes: Vec::new(),
46                 finished: false,

```

```

47     }
48 }
49 }

```

Listing B.31: The GlmPeer struct and its implementation

```

1 pub struct GeneralizedLinearModel {
2     pub r_local: Mat<f64>,
3     pub coefficients: Mat<f64>,
4     pub family: FamilyEnum,
5     pub iter: usize,
6 }

```

Listing B.32: The GeneralizedLinearModel struct

```

1 pub fn on_message_receive(
2     ctx: &mut Context,
3     _sender_id: usize,
4     receiver_id: usize,
5     msg: &dyn Message,
6 ) {
7     if let Some(sum_rows_msg) = msg.downcast_ref::<GlmSumRowsMessage>() {
8         receive_sum_rows_msg(ctx, receiver_id, *sum_rows_msg);
9     } else if let Some(concat_msg) = msg.downcast_ref::<GlmConcatMessage>() {
10        receive_concat_r_msg(ctx, receiver_id, concat_msg.clone());
11    } else {
12        log::warn(ctx, format!("Wrong message type received {msg:?}"));
13    }
14 }

```

Listing B.33: GlmPeer's callback when receiving a message

```

1 #[derive(Debug, Clone)]
2 pub struct KillTimer {
3     pub target: usize,
4 }
5
6 impl KillTimer {
7     pub fn new(target: usize) -> Self {
8         Self { target }
9     }
10 }
11
12 impl Timer for KillTimer {
13     fn fire(&self, ctx: &mut Context) {
14         if let Some(target) = ctx.peers.get_mut(self.target) {
15             target.kill();
16             log::info(ctx, format!("Peer {} killed.", self.target));
17         }
18     }
19 }

```

```

17     }
18 }
19 }

```

Listing B.34: The KillTimer struct

```

1 #[derive(Debug, Clone)]
2 pub struct StartTimer {
3     pub peer_id: usize,
4 }
5
6 impl Timer for StartTimer {
7     fn fire(&self, ctx: &mut Context) {
8         algorithms::broadcast_sum_rows(ctx, self.peer_id);
9     }
10 }

```

Listing B.35: The StartTimer struct

```

1 pub fn broadcast_sum_rows(ctx: &mut Context, peer_id: usize) {
2     let mut nodes_filtered: Vec<usize> = Vec::new();
3     if let Some(neighbors) = engine::get_neighbors_alive(ctx, peer_id) {
4         for neigh_id in neighbors {
5             if ctx.peers.get(neigh_id).is_some_and(|p| p.is::<GlmPeer>()) {
6                 send_sum_rows(ctx, peer_id, neigh_id);
7                 nodes_filtered.push(neigh_id);
8             }
9         }
10    }
11
12    let peer: &mut GlmPeer = get_peer_of_type!(ctx, peer_id, GlmPeer).expect("peer
    should exist");
13
14    peer.state.nodes = nodes_filtered;
15 }

```

Listing B.36: A method which broadcasts a peer's number of rows

```

1 fn send_sum_rows(ctx: &mut Context, peer_id: usize, target_id: usize) {
2     let peer: &mut GlmPeer = get_peer_of_type!(ctx, peer_id, GlmPeer).expect("peer
    should exist");
3
4     let msg = GlmSumRowsMessage {
5         origin: peer_id,
6         nrow: peer.state.total_nrow,
7     };
8
9     let trace = match engine::send_message_to(ctx, peer_id, target_id, msg) {

```

```

10         Some(lat) => format!("Sent GlmSumRowsMessage from {peer_id} to {target_id} in {
11             lat}"),
12         None => format!("Failed to send GlmSumRowsMessage from {peer_id} to {target_id}"
13             ),
14     };
15     log::trace(ctx, trace);
16 }

```

Listing B.37: The method that sends a peer's number of rows

```

1  #[derive(Debug, Clone, Copy)]
2  pub struct GlmSumRowsMessage {
3      pub origin: usize,
4      pub nrows: usize,
5  }
6  impl Message for GlmSumRowsMessage {
7      fn size_bytes(&self) -> u64 {
8          1 + 1
9      }
10 }

```

Listing B.38: The GlmSumRowsMessage message struct

```

1  pub fn receive_sum_rows_msg(ctx: &mut Context, peer_id: usize, msg: GlmSumRowsMessage) {
2      log::trace(
3          ctx,
4          format!(
5              "peer {peer_id} received a GlmSumRowsMessage message from peer {sender} with
6              nrows: {nrows}",
7              sender = msg.origin,
8              nrows = msg.nrows
9          ),
10     );
11     let peer: &mut GlmPeer = get_peer_of_type!(ctx, peer_id, GlmPeer).expect("peer
12         should exist");
13     if let std::collections::hash_map::Entry::Vacant(e) = peer.state.r_n_rows.entry(msg.
14         origin) {
15         // INFO: replaced r_remotes with r_n_rows
16         e.insert(msg.nrows);
17     }
18     if peer.state.nodes.len() == peer.state.r_n_rows.keys().len() {
19         peer.state.total_nrow += peer.state.r_n_rows.values().sum::<usize>();
20         peer.state.r_remotes = HashMap::new();
21         broadcast_nodes(ctx, peer_id);
22     }
23     } else {
24         log::warn(
25             ctx,
26             format!(

```

```

26         "peer {peer_id} received GlmSumRowsMessage but r_remotes already
27         contained {}",
28         msg.origin
29     ),
30 };
31 }

```

Listing B.39: The function called when a GlmSumRowsMessage message is received

```

1 fn broadcast_nodes(ctx: &mut Context, peer_id: usize) {
2     let peer: &mut GlmPeer = get_peer_of_type!(ctx, peer_id, GlmPeer).expect("peer
3         should exist");
4
5     let neigh_ids = peer.state.nodes.clone();
6     for neigh_id in neigh_ids {
7         send_concat_r(ctx, peer_id, neigh_id);
8     }
9 }

```

Listing B.40: A method which broadcasts a peer's number of rows

```

1 fn send_concat_r(ctx: &mut Context, peer_id: usize, target_id: usize) {
2     let peer: &mut GlmPeer = get_peer_of_type!(ctx, peer_id, GlmPeer).expect("peer
3         should exist");
4
5     let msg = GlmConcatMessage {
6         origin: peer_id,
7         r_remote: peer.state.model.r_local.clone(),
8         iter: peer.state.model.iter,
9     };
10
11     let iter = peer.state.model.iter;
12
13     let trace = match engine::send_message_to(ctx, peer_id, target_id, msg) {
14         Some(lat) => format!(
15             "Sent GlmConcatMessage from {peer_id} to {target_id} (iteration {iter}) in {
16             lat}",
17         ),
18         None => format!("Failed to send GlmConcatMessage from {peer_id} to {target_id}")
19     };
20     log::trace(ctx, trace);
21 }

```

Listing B.41: The method that sends an upper triangular matrix to another peers

```

1 #[derive(Debug, Clone)]

```

```

2 pub struct GlmConcatMessage {
3     pub origin: usize,
4     pub r_remote: Mat<f64>,
5     pub iter: usize,
6 }
7 impl Message for GlmConcatMessage {
8     fn size_bytes(&self) -> u64 {
9         let (r, c) = self.r_remote.shape();
10
11         (1 + r * c * 8 + 1) as u64
12     }
13 }

```

Listing B.42: The GlmConcatMessage message struct

```

1 pub fn receive_concat_r_msg(ctx: &mut Context, peer_id: usize, msg: GlmConcatMessage) {
2     log::trace(
3         ctx,
4         format!(
5             "peer {peer_id} received a GlmConcatMessage from {sender} on iteration {iter}
6             ),
7         sender = msg.origin,
8         iter = msg.iter
9     );
10
11     let peer: &mut GlmPeer = get_peer_of_type!(ctx, peer_id, GlmPeer).expect("peer
12         should exist");
13
14     // if does not have key msg.iter then insert HashMap::new()
15     peer.state.r_remotes.entry(msg.iter).or_default();
16
17     handle_iter(ctx, peer_id, msg.origin, msg.r_remote, msg.iter);
18 }

```

Listing B.43: The function called when a GlmConcatMessage message is received

```

1 fn handle_iter(ctx: &mut Context, peer_id: usize, sender: usize, r_remote: Mat<f64>,
2     iter: usize) {
3     let peer: &mut GlmPeer = get_peer_of_type!(ctx, peer_id, GlmPeer).expect("peer
4         should exist");
5
6     if peer.state.r_remotes.contains_key(&iter)
7         && !peer
8             .state
9             .r_remotes
10             .get(&iter)
11             .unwrap()
12             .contains_key(&sender)
13     {
14         peer.state

```

```

13         .r_remotes
14         .get_mut(&iter)
15         .and_then(|r| r.insert(sender, r_remote));
16
17     let current_iter = peer.state.model.iter;
18     let msgs_to_receive = peer.state.nodes.len();
19     let msgs_received_in_iter = peer.state.r_remotes.get(&iter).unwrap().keys().len
20         ();
21
22     if iter == current_iter && msgs_to_receive == msgs_received_in_iter {
23         let mut all_r_remotes = peer
24             .state
25             .r_remotes
26             .get(&iter)
27             .unwrap()
28             .values()
29             .cloned()
30             .collect::

```

```

65         broadcast_nodes(ctx, peer_id);
66     } else {
67         log::info(ctx, format!("peer {peer_id} finished on iteration {iter}"));
68     }
69 }
70 } else if msgs_to_receive == msgs_received_in_iter {
71     log::warn(
72         ctx,
73         format!(
74             "peer {peer_id} handle_iter, received all messages from iteration {
75                 iter} but current iteration is {current_iter}",
76             ),
77     );
78 } else {
79     log::trace(
80         ctx,
81         format!(
82             "peer {peer_id} handle_iter, received ({msgs_received_in_iter}/{
83                 msgs_to_receive}) messages on iteration {iter} (current
84                 iteration is {current_iter})",
85             ),
86     );
87 }
88 } else {
89     log::warn(
90         ctx,
91         format!(
92             "peer {peer_id} handle_iter failed because r_remotes[{iter}] already
93             contains sender {sender}"
94         ),
95     );
96 }
97 }
98 }

```

Listing B.44: The function which handles the algorithm's iterations

```

1  #[derive(Copy, Clone)]
2  pub enum CatDim {
3      Vertical = 0,
4      Horizontal = 1,
5  }
6  pub fn mat_cat(a: &Mat<f64>, b: &Mat<f64>, dim: CatDim) -> Mat<f64> {
7      match dim {
8          CatDim::Vertical => {
9              let ncols = a.ncols();
10             let nrows_total = a.nrows() + b.nrows();
11
12             assert_eq!(
13                 a.ncols(),
14                 b.ncols(),
15                 "Column count must match for vertical concatenation"
16             );
17
18             let mut out = Mat::::zeros(nrows_total, ncols);

```

```

19         out.submatrix_mut(0, 0, a.nrows(), ncols).copy_from(a);
20         out.submatrix_mut(a.nrows(), 0, b.nrows(), ncols)
21             .copy_from(b);
22         out
23     }
24     CatDim::Horizontal => {
25         let nrows = a.nrows();
26         let ncols_total = a.ncols() + b.ncols();
27
28         assert_eq!(
29             a.nrows(),
30             b.nrows(),
31             "Row count must match for horizontal concatenation"
32         );
33
34         let mut out = Mat::<f64>::zeros(nrows, ncols_total);
35         out.submatrix_mut(0, 0, nrows, a.ncols()).copy_from(a);
36         out.submatrix_mut(0, a.ncols(), nrows, b.ncols())
37             .copy_from(b);
38         out
39     }
40 }
41 }
42
43 pub fn mat_cat_vec(mats: &[Mat<f64>], dim: CatDim) -> Mat<f64> {
44     assert!(
45         !mats.is_empty(),
46         "mat_cat_vec must receive at least one matrix"
47     );
48
49     let mut iterator = mats.iter();
50     let mut result = iterator.next().unwrap().clone();
51
52     for mat in iterator {
53         result = mat_cat(&result, mat, dim);
54     }
55
56     result
57 }

```

Listing B.45: The function which concatenates matrices

```

1 pub fn distributed_single_solve_n(
2     r_local_with_all_r_remotes: &Mat<f64>,
3     beta: &Mat<f64>,
4     family: FamilyEnum,
5     total_nrow: usize,
6     maxit: usize,
7     tol: f64,
8     iter: usize,
9 ) -> (Mat<f64>, Mat<f64>, bool) {
10     let beta_old = beta.clone();
11
12     let (r_local, beta) = ols_n(r_local_with_all_r_remotes);

```

```

13
14     let vcov = vcov(&r_local, family, total_nrow);
15     let delta = mat_div_elemwise(
16         &(beta_old - beta.clone()),
17         &mat_sqrt_elemwise(&mat_diag(&vcov)),
18     );
19     let diff = delta.max().unwrap().abs();
20     let stop = stop(maxit, tol, iter, diff);
21
22     (r_local, beta, stop)
23 }

```

Listing B.46: Distributed GLM solver function for convergence checking and coefficients updates

```

1 pub fn distributed_single_iter_n(
2     family: FamilyEnum,
3     x: &Mat<f64>,
4     y: &Mat<f64>,
5     beta: Mat<f64>,
6 ) -> Mat<f64> {
7     let eta = x.clone() * beta;
8
9     let mu = match family {
10         FamilyEnum::Binomial => binomial::linkinv(&eta),
11         FamilyEnum::Gaussian => gaussian::linkinv(&eta),
12     };
13     let dmu = match family {
14         FamilyEnum::Binomial => binomial::mu_eta(&eta),
15         FamilyEnum::Gaussian => gaussian::mu_eta(&eta),
16     };
17
18     let variance = match family {
19         FamilyEnum::Binomial => binomial::variance(&mu),
20         FamilyEnum::Gaussian => gaussian::variance(&mu),
21     };
22     let z = eta + mat_div_elemwise(&(y - mu.clone()), &dmu.clone());
23     let w = mat_div_elemwise(&mat_sqrt_elemwise(&dmu.clone()), &variance);
24
25     let sqrt_w = &mat_sqrt_elemwise(&w.clone());
26     let x_tilde = &mul_elementwise(sqrt_w, x);
27     let z_tilde = &mul_elementwise(sqrt_w, &z);
28
29     // r_local
30     mat_cat(x_tilde, z_tilde, CatDim::Horizontal)
31         .qr()
32         .R()
33         .to_owned()
34 }

```

Listing B.47: Single iteration function for distributed GLM

```

1 pub fn on_simulation_finish_hook(central: Mat<f64>) -> CustomOnFinishHook {
2     Box::new(move |ctx| {
3         let coefficients: Vec<Mat<f64>> = ctx
4             .peers
5             .iter()
6             .filter_map(|p| {
7                 p.downcast_ref::<GlmPeer>()
8                 .map(|peer| peer.state.model.coefficients.clone())
9             })
10            .collect();
11
12        check(ctx, &central, &coefficients);
13    })
14 }
15
16 fn check(ctx: &mut Context, central: &Mat<f64>, coefficients: &[Mat<f64>]) {
17     let res = coefficients
18         .iter()
19         .all(|coef| mat_allclose_default(coef, central));
20
21     let data = json!({
22         "coefficients": format!("{coefficients:?}"),
23         "central": format!("{central:?}")
24     });
25     log::metrics(ctx, "central", &data);
26
27     log::info(
28         ctx,
29         format!("Are the coefficients from every peer equal to central's? {res}"),
30     );
31 }

```

Listing B.48: The on finish hook and its helper function

B.2.1.1 Helper Functions

```

1 pub fn linkinv(eta: &Mat<f64>) -> Mat<f64> {
2     Mat::from_fn(eta.nrows(), eta.ncols(), |i, j| {
3         let x = eta.get(i, j);
4         1.0 / (1.0 + (-x).exp())
5     })
6 }
7
8 pub fn mu_eta(eta: &Mat<f64>) -> Mat<f64> {
9     Mat::from_fn(eta.nrows(), eta.ncols(), |i, j| {
10         let x = eta.get(i, j);
11         let exp = x.exp();
12         exp / ((1.0 + exp).powi(2))
13     })
14 }
15
16 pub fn variance(mu: &Mat<f64>) -> Mat<f64> {
17     Mat::from_fn(mu.nrows(), mu.ncols(), |i, j| {

```

```

18         let x = mu.get(i, j);
19         x * (1.0 - x)
20     })
21 }

```

Listing B.49: The Binomial family's module functions

```

1 pub fn mat_div_elemwise(a: &Mat<f64>, b: &Mat<f64>) -> Mat<f64> {
2     assert_eq!(a.nrows(), b.nrows(), "Mismatched number of rows");
3     assert_eq!(a.ncols(), b.ncols(), "Mismatched number of columns");
4
5     Mat::from_fn(a.nrows(), a.ncols(), |i, j| a.get(i, j) / b.get(i, j))
6 }

```

Listing B.50: Matrix division element-wise

```

1 pub fn mat_sqrt_elemwise(a: &Mat<f64>) -> Mat<f64> {
2     Mat::from_fn(a.nrows(), a.ncols(), |i, j| a.get(i, j).sqrt())
3 }

```

Listing B.51: Matrix square root element-wise

```

1 pub fn mat_sqr_elemwise(a: &Mat<f64>) -> Mat<f64> {
2     Mat::from_fn(a.nrows(), a.ncols(), |i, j| a.get(i, j) * a.get(i, j))
3 }

```

Listing B.52: Matrix square element-wise

```

1 pub fn mat_diag(a: &Mat<f64>) -> Mat<f64> {
2     let sz = a.nrows().min(a.ncols());
3     Mat::from_fn(sz, 1, |i, _| *a.get(i, i))
4 }

```

Listing B.53: Retrieves a column of the matrix's diagonal

```

1 pub fn mul_elementwise(a: &Mat<f64>, b: &Mat<f64>) -> Mat<f64> {
2     let (nrows, ncols) = (a.nrows(), a.ncols());
3     let (brows, bcols) = (b.nrows(), b.ncols());
4
5     match (nrows, ncols, brows, bcols) {
6         // Direct element-wise multiplication
7         (ar, ac, br, bc) if ar == br && ac == bc => {

```

```

8         Mat::from_fn(ar, ac, |i, j| a.get(i, j) * b.get(i, j))
9     }
10    // Column broadcast on b
11    (ar, ac, br, 1) if ar == br => Mat::from_fn(ar, ac, |i, j| a.get(i, j) * b.get(i
    , 0)),
12    // Column broadcast on a
13    (ar, 1, br, bc) if ar == br => Mat::from_fn(br, bc, |i, j| a.get(i, 0) * b.get(i
    , j)),
14    // Row broadcast on b
15    (ar, ac, 1, bc) if ac == bc => Mat::from_fn(ar, ac, |i, j| a.get(i, j) * b.get
    (0, j)),
16    // Row broadcast on a
17    (1, ac, br, bc) if ac == bc => Mat::from_fn(br, bc, |i, j| a.get(0, j) * b.get(i
    , j)),
18    // Scalar broadcast on b
19    (ar, ac, 1, 1) => {
20        let val = b.get(0, 0);
21        Mat::from_fn(ar, ac, |i, j| a.get(i, j) * val)
22    }
23    // Scalar broadcast on a
24    (1, 1, br, bc) => {
25        let val = a.get(0, 0);
26        Mat::from_fn(br, bc, |i, j| a.get(i, j) * val)
27    }
28    // Incompatible shapes
29    _ => panic!("Cannot multiply matrices element-wise due to incompatible shapes"),
30 }
31 }

```

Listing B.54: Matrix multiplication element-wise and broadcast

```

1  const DEFAULT_RTOL: f64 = 1e-05;
2  const DEFAULT_ATOL: f64 = 1e-08;
3
4  pub fn mat_allclose(a: &Mat<f64>, b: &Mat<f64>, rtol: f64, atol: f64) -> bool {
5      if a.nrows() != b.nrows() || a.ncols() != b.ncols() {
6          return false;
7      }
8
9      for i in 0..a.nrows() {
10         for j in 0..a.ncols() {
11             let diff = (a.get(i, j) - b.get(i, j)).abs();
12             let tolerance = atol + rtol * b.get(i, j).abs();
13             if diff > tolerance {
14                 return false;
15             }
16         }
17     }
18     true
19 }
20
21 pub fn mat_allclose_default(a: &Mat<f64>, b: &Mat<f64>) -> bool {
22     mat_allclose(a, b, DEFAULT_RTOL, DEFAULT_ATOL)
23 }

```

Listing B.55: Checks if two matrices are approximately equal element-wise

B.2.2 Experiments

```

1 seed: &default_seed 2586804559409604052
2 arrival_time: &default_arrival_time distance
3 topology: &default_topology
4   n_peers: 7
5   name: full
6
7 experiments:
8 - name: default_run
9   scenario: dglm
10  arrival_time: *default_arrival_time
11  topology: *default_topology
12  repetitions: 100
13
14 - name: run_100_peers
15   scenario: dglm
16   seed: *default_seed
17   arrival_time: *default_arrival_time
18   topology:
19     n_peers: 100
20     name: full
21
22 - name: constant_latency
23   scenario: dglm
24   seed: *default_seed
25   arrival_time: constant
26   topology: *default_topology
27
28 - name: custom_latency
29   scenario: dglm
30   seed: *default_seed
31   topology:
32     n_peers: 7
33     name: twowaycustom
34     connections:
35       - [0, 1, !Full {bandwidth: 80000, latency: 0.02} ]
36       - [0, 2, !Full {bandwidth: 80000, latency: 0.03604} ]
37       - [0, 3, !Full {bandwidth: 80000, latency: 0.04768} ]
38       - [0, 4, !Full {bandwidth: 80000, latency: 0.04768} ]
39       - [0, 5, !Full {bandwidth: 80000, latency: 0.03604} ]
40       - [0, 6, !Full {bandwidth: 80000, latency: 0.02} ]
41       - [1, 2, !Full {bandwidth: 80000, latency: 0.02} ]
42       - [1, 3, !Full {bandwidth: 80000, latency: 0.03604} ]
43       - [1, 4, !Full {bandwidth: 80000, latency: 0.04768} ]
44       - [1, 5, !Full {bandwidth: 80000, latency: 0.04768} ]
45       - [1, 6, !Full {bandwidth: 80000, latency: 0.03604} ]
46       - [2, 3, !Full {bandwidth: 80000, latency: 0.02} ]
47       - [2, 4, !Full {bandwidth: 80000, latency: 0.03604} ]

```

```

48     - [2, 5, !Full {bandwidth: 80000, latency: 0.04768} ]
49     - [2, 6, !Full {bandwidth: 80000, latency: 0.04768} ]
50     - [3, 4, !Full {bandwidth: 80000, latency: 0.02} ]
51     - [3, 5, !Full {bandwidth: 80000, latency: 0.03604} ]
52     - [3, 6, !Full {bandwidth: 80000, latency: 0.04768} ]
53     - [4, 5, !Full {bandwidth: 80000, latency: 0.02} ]
54     - [4, 6, !Full {bandwidth: 80000, latency: 0.03604} ]
55     - [5, 6, !Full {bandwidth: 80000, latency: 0.02} ]
56
57 - name: ring_topology
58   scenario: dgml
59   seed: *default_seed
60   arrival_time: *default_arrival_time
61   topology:
62     n_peers: 7
63     name: ring
64
65 - name: custom_ring_topology
66   scenario: dgml
67   topology:
68     n_peers: 7
69     name: twowaycustom
70     connections:
71       - [0, 1, !Full {bandwidth: 80000, latency: 0.02} ]
72       - [1, 2, !Full {bandwidth: 80000, latency: 0.02} ]
73       - [2, 3, !Full {bandwidth: 80000, latency: 0.02} ]
74       - [3, 4, !Full {bandwidth: 80000, latency: 0.02} ]
75       - [4, 5, !Full {bandwidth: 80000, latency: 0.02} ]
76       - [5, 6, !Full {bandwidth: 80000, latency: 0.02} ]
77       - [6, 0, !Full {bandwidth: 80000, latency: 0.02} ]
78
79 - name: star_topology
80   scenario: dgml
81   arrival_time: *default_arrival_time
82   topology:
83     n_peers: 7
84     name: star
85   repetitions: 10
86
87 - name: peer_killed
88   scenario: dgml
89   seed: *default_seed
90   arrival_time: *default_arrival_time
91   topology: *default_topology
92   extra_args:
93     kill_peer: true
94
95 - name: drop_rate_0_2
96   scenario: dgml
97   seed: *default_seed
98   arrival_time: *default_arrival_time
99   topology: *default_topology
100  drop_rate: 0.2
101
102 - name: duplicate_rate_0_2
103   scenario: dgml
104   seed: *default_seed

```

```

105 arrival_time: *default_arrival_time
106 topology: *default_topology
107 duplicate_rate: 0.2
108
109 - name: jitter
110   scenario: dglm
111   arrival_time: *default_arrival_time
112   topology: *default_topology
113   jitter:
114     distribution: !Weibull [1.064, 2.872]
115     multiplier: 0.001
116     repetitions: 100

```

Listing B.56: The evaluation configuration file

B.2.3 Proposed Solution

```

1  #[derive(Debug, Clone)]
2  pub struct StartTimer {
3      pub peer_id: usize,
4  }
5
6  impl Timer for StartTimer {
7      fn fire(&self, ctx: &mut Context) {
8          let peer: &mut PGlmPeer =
9              get_peer_of_type!(ctx, self.peer_id, PGlmPeer).expect("peer should exist");
10         if !peer.is_alive() {
11             return;
12         }
13         algorithms::get_node_ids(ctx, self.peer_id);
14         algorithms::broadcast_sum_rows(ctx, self.peer_id);
15     }
16 }

```

Listing B.57: The StartTimer struct

```

1  pub fn get_node_ids(ctx: &mut Context, peer_id: usize) {
2      log::trace(ctx, format!("peer {peer_id} get node ids"));
3
4      let mut peer_ids_filtered: Vec<usize> = engine::get_neighbors_alive(ctx, peer_id)
5          .map(|neighbors| {
6              neighbors
7                  .into_iter()
8                  .filter(|&neigh_id| ctx.peers.get(neigh_id).is_some_and(|p| p.is::<
9                      PGlmPeer>()))
10                 .collect()
11          })
12      .unwrap_or_default();

```

```

13     let peer: &mut PGlmPeer = get_peer_of_type!(ctx, peer_id, PGlmPeer).expect("peer
        should exist");
14
15     peer.state.neighbors = peer_ids_filtered.clone();
16
17     peer_ids_filtered.push(peer.get_id());
18     peer.state.nodes = BTreeSet::from_iter(peer_ids_filtered);
19     peer.discovery_reset();
20     log::trace(ctx, format!("peer {peer_id} called discovery_reset"));
21 }

```

Listing B.58: The method which sets up the peer's discovery phase

```

1 impl PGlmPeer {
2
3     ...
4
5     pub fn update_hash(&mut self) {
6         let mut s = hash_map::DefaultHasher::new();
7         self.state.nodes.hash(&mut s);
8         self.state.hash = s.finish()
9     }
10
11     pub fn discovery_reset(&mut self) {
12         self.state.r_n_rows = HashMap::new();
13         self.state.r_matrices = HashMap::new();
14         self.state.total_nrow = 0;
15         self.state.finished = false;
16         self.state.model = self.state.initial_model.clone();
17
18         let peer_id = self.get_id();
19         self.state
20             .r_matrices
21             .entry(0)
22             .or_default()
23             .insert(peer_id, self.state.model.r_local.clone());
24
25         self.update_hash();
26     }
27 }

```

Listing B.59: The methods which delete the peer's state outdated data and calculates the hash

```

1 pub fn broadcast_sum_rows(ctx: &mut Context, peer_id: usize) {
2     let peer: &mut PGlmPeer = get_peer_of_type!(ctx, peer_id, PGlmPeer).expect("peer
        should exist");
3
4     for neigh_id in peer.state.neighbors.clone() {
5         send_sum_rows(ctx, peer_id, neigh_id);
6     }
7 }

```

Listing B.60: A method which broadcasts a peer's number of rows

```

1  #[derive(Debug, Clone)]
2  pub struct PGlmSumRowsMessage {
3      pub origin: usize,
4      pub nrows: usize,
5      pub hash: u64,
6  }
7  impl Message for PGlmSumRowsMessage {
8      fn size_bytes(&self) -> u64 {
9          1 + 1 + 8
10     }
11 }

```

Listing B.61: The PGlmSumRowsMessage struct

```

1  #[derive(Debug, Clone)]
2  pub struct PGlmConcatMessage {
3      pub origin: usize,
4      pub r_remote: Mat<f64>,
5      pub iter: usize,
6      pub hash: u64,
7  }
8  impl Message for PGlmConcatMessage {
9      fn size_bytes(&self) -> u64 {
10         let (r, c) = self.r_remote.shape();
11
12         (1 + r * c * 8 + 1 + 8) as u64
13     }
14 }

```

Listing B.62: The PGlmConcatMessage struct

```

1  #[derive(Debug, Clone)]
2  pub struct ReqSumRowsMessage {
3      pub needs: Vec<usize>,
4      pub hash: u64,
5  }
6  impl Message for ReqSumRowsMessage {
7      fn size_bytes(&self) -> u64 {
8          self.needs.len() as u64 + 8
9      }
10 }

```

Listing B.63: The ReqSumRowsMessage struct

```

1  #[derive(Debug, Clone)]
2  pub struct ReqConcatMessage {
3      pub needs: Vec<usize>,
4      pub iter: usize,
5      pub hash: u64,
6  }
7  impl Message for ReqConcatMessage {
8      fn size_bytes(&self) -> u64 {
9          self.needs.len() as u64 + 1 + 8
10     }
11 }

```

Listing B.64: The ReqConcatMessage struct

```

1  /// Checks if both peers have the same knowledge of the network
2  /// Returns whether the message should proceed or not
3  pub fn operation_msg_hash_guard(
4      ctx: &mut Context,
5      sender_id: usize,
6      receiver_id: usize,
7      sender_hash: u64,
8  ) -> bool {
9      let Ok(peer) = get_peer_of_type!(ctx, receiver_id, PGlmPeer) else {
10         return false;
11     };
12
13     let receiver_hash = peer.state.hash;
14
15     if receiver_hash == sender_hash {
16         true
17     } else {
18         log::warn(
19             ctx,
20             format!(
21                 "receiver peer {receiver_id} has a different hash than sender peer {
22                     sender_id}, {receiver_hash} != {sender_hash}"
23             ),
24         );
25
26         send_discovery_msg(ctx, sender_id, receiver_id);
27         false
28     }
29 }

```

Listing B.65: A method which checks if peers have the same hash

```

1  pub fn on_message_receive(
2      ctx: &mut Context,
3      sender_id: usize,
4      receiver_id: usize,

```

```

5     msg: &dyn Message,
6 ) {
7     if let Some(sum_rows_msg) = msg.downcast_ref::<PGlmSumRowsMessage>() {
8         if operation_msg_hash_guard(ctx, sender_id, receiver_id, sum_rows_msg.hash) {
9             receive_sum_rows_msg(ctx, receiver_id, sum_rows_msg.clone());
10        }
11    } else if let Some(concat_msg) = msg.downcast_ref::<PGlmConcatMessage>() {
12        if operation_msg_hash_guard(ctx, sender_id, receiver_id, concat_msg.hash) {
13            receive_concat_r_msg(ctx, receiver_id, concat_msg.clone());
14        }
15    } else if let Some(discovery_req_msg) = msg.downcast_ref::<ReqSumRowsMessage>() {
16        if operation_msg_hash_guard(ctx, sender_id, receiver_id, discovery_req_msg.hash)
17        {
18            receive_sum_rows_req_msg(ctx, receiver_id, sender_id, discovery_req_msg.
19                clone());
20        }
21    } else if let Some(concat_req_msg) = msg.downcast_ref::<ReqConcatMessage>() {
22        if operation_msg_hash_guard(ctx, sender_id, receiver_id, concat_req_msg.hash) {
23            receive_concat_r_req_msg(ctx, receiver_id, sender_id, concat_req_msg.clone()
24                );
25        }
26    } else if let Some(discovery_msg) = msg.downcast_ref::<DiscoveryMessage>() {
27        receive_discovery_msg(ctx, receiver_id, discovery_msg.clone());
28    } else {
29        log::warn(
30            ctx,
31            format!("Wrong message type received {msg:?} from {sender_id}"),
32        );
33    }
34 }

```

Listing B.66: The PGlmPeer's on message receive callback

```

1 pub fn send_discovery_msg(ctx: &mut Context, sender_id: usize, receiver_id: usize) {
2     let sender_peer: &mut PGlmPeer =
3         get_peer_of_type!(ctx, sender_id, PGlmPeer).expect("peer should exist");
4
5     let ids: Vec<usize> = sender_peer.state.nodes.iter().copied().collect();
6
7     let msg = DiscoveryMessage {
8         origin: sender_id,
9         nodes: ids.clone(),
10    };
11
12    log::trace(
13        ctx,
14        format!("peer {sender_id} Sent DiscoveryMessage to peer {receiver_id} with ids {
15            ids:?}"),
16    );
17    engine::send_message_to(ctx, sender_id, receiver_id, msg);
18 }

```

Listing B.67: A method sends a message with the known list of identifiers

```

1  #[derive(Debug, Clone)]
2  pub struct DiscoveryMessage {
3      pub origin: usize,
4      pub nodes: Vec<usize>,
5  }
6  impl Message for DiscoveryMessage {
7      fn size_bytes(&self) -> u64 {
8          let n_nodes = self.nodes.len();
9
10         (1 + n_nodes) as u64
11     }
12 }

```

Listing B.68: The DiscoveryMessage struct

```

1  pub fn receive_discovery_msg(ctx: &mut Context, peer_id: usize, msg: DiscoveryMessage) {
2      log::trace(
3          ctx,
4          format!(
5              "peer {peer_id} received a DiscoveryMessage from {sender} with ids {ids:?}",
6              sender = msg.origin,
7              ids = msg.nodes
8          ),
9      );
10
11     let peer: &mut PGlmPeer = get_peer_of_type!(ctx, peer_id, PGlmPeer).expect("peer
        should exist");
12
13     let old_peers_size = peer.state.nodes.len();
14     peer.state.nodes.extend(msg.nodes);
15
16     // Check if discovery updated
17     if old_peers_size != peer.state.nodes.len() {
18         peer.discovery_reset();
19         log::trace(ctx, format!("peer {peer_id} called discovery_reset"));
20
21         // before broadcasting, check if a neighbor was added
22         let peer_ids_filtered: Vec<usize> = engine::get_neighbors_alive(ctx, peer_id)
23             .map(|neighbors| {
24                 neighbors
25                     .into_iter()
26                     .filter(|&neigh_id| ctx.peers.get(neigh_id).is_some_and(|p| p.is::<
27                         PGlmPeer>()))
28                     .collect()
29             })
30             .unwrap_or_default();
31
32         let peer: &mut PGlmPeer =
33             get_peer_of_type!(ctx, peer_id, PGlmPeer).expect("peer should exist");
34
35         peer.state.neighbors = peer_ids_filtered.clone();
36
37         broadcast_ids(ctx, peer_id);

```

```

37
38     check_missing_sum_rows(ctx, peer_id);
39 }
40 }

```

Listing B.69: The function called when a GlmSumRowsMessage message is received

```

1 pub fn check_missing_sum_rows(ctx: &mut Context, peer_id: usize) -> bool {
2     let peer: &mut PGlmPeer = get_peer_of_type!(ctx, peer_id, PGlmPeer).expect("peer
    should exist");
3
4     let mut sum_rows_to_req: Vec<usize> = vec![];
5     for neigh_id in peer.state.nodes.iter() {
6         if !peer.state.r_n_rows.contains_key(neigh_id) && *neigh_id != peer_id {
7             sum_rows_to_req.push(*neigh_id);
8         }
9     }
10
11     let hash = peer.state.hash;
12     let nodes = peer.state.nodes.clone();
13     let neighbors = peer.state.neighbors.clone();
14
15     log::debug(
16         ctx,
17         format!(
18             "tick check check_missing_sum_rows peer {peer_id} misses: {sum_rows_to_req
                :?}, {nodes:?}, neighbors: {neighbors:?}"
19         ),
20     );
21
22     if !sum_rows_to_req.is_empty() {
23         log::trace(
24             ctx,
25             format!("peer {peer_id} has missing rows {sum_rows_to_req:?}"),
26         );
27         let msg = ReqSumRowsMessage {
28             needs: sum_rows_to_req.clone(),
29             hash,
30         };
31         broadcast_sum_rows_req(ctx, peer_id, msg);
32         true
33     } else {
34         false
35     }
36 }

```

Listing B.70: A method which checks for and requests missing number of rows

```

1 pub fn receive_sum_rows_req_msg(
2     ctx: &mut Context,
3     peer_id: usize,

```

```

4     requester_id: usize,
5     msg: ReqSumRowsMessage,
6 ) {
7     let peer: &mut PGlmPeer = get_peer_of_type!(ctx, peer_id, PGlmPeer).expect("peer
        should exist");
8
9     let mut has: Vec<usize> = peer.state.r_n_rows.keys().copied().collect();
10    has.push(peer_id);
11
12    log::trace(
13        ctx,
14        format!(
15            "peer {peer_id} received a request for rows from {rows:?}, it has: {has:?}",
16            rows = msg.needs
17        ),
18    );
19
20    for p_id in msg.needs {
21        let peer: &mut PGlmPeer =
22            get_peer_of_type!(ctx, peer_id, PGlmPeer).expect("peer should exist");
23
24        let mut n_rows: usize = 0;
25        let hash = peer.state.hash;
26
27        let mut has_rows = false;
28        if let Some(n) = peer.state.r_n_rows.get(&p_id) {
29            n_rows = *n;
30            has_rows = true;
31        } else if p_id == peer_id {
32            n_rows = peer.state.local_nrow;
33            has_rows = true;
34        }
35
36        if has_rows {
37            engine::send_message_to(
38                ctx,
39                peer_id,
40                requester_id,
41                PGlmSumRowsMessage {
42                    origin: p_id,
43                    n_rows: n_rows,
44                    hash,
45                },
46            );
47        }
48    }
49 }

```

Listing B.71: The function called when a ReqSumRowsMessage message is received

```

1 pub fn receive_sum_rows_msg(ctx: &mut Context, peer_id: usize, msg: PGlmSumRowsMessage)
2 {
3     log::trace(
4         ctx,

```

```

4      format!(
5          "peer {peer_id} received a PGlmSumRowsMessage message from peer {sender}
           with nrows: {nrows}",
6          sender = msg.origin,
7          nrows = msg.nrows
8      ),
9  );
10
11  let peer: &mut PGlmPeer = get_peer_of_type!(ctx, peer_id, PGlmPeer).expect("peer
           should exist");
12
13  if let std::collections::hash_map::Entry::Vacant(e) = peer.state.r_n_rows.entry(msg.
           origin) {
14      e.insert(msg.nrows);
15
16      if peer.state.nodes.len() - 1 == peer.state.r_n_rows.keys().len() {
17          peer.state.total_nrow =
18              peer.state.local_nrow + peer.state.r_n_rows.values().sum::<usize>();
19          peer.state.r_matrices = HashMap::new();
20          // r_matrices should include initial r_local
21          peer.state
22              .r_matrices
23              .entry(0)
24              .or_default()
25              .insert(peer_id, peer.state.model.r_local.clone());
26
27          peer.state.model = peer.state.initial_model.clone();
28
29          check_missing_concat_r(ctx, peer_id);
30      }
31  } else {
32      log::warn(
33          ctx,
34          format!(
35              "peer {peer_id} received GlmSumRowsMessage but r_remotes already
               contained {}",
36              msg.origin
37          ),
38      );
39  }
40 }

```

Listing B.72: The function called when a PGlmSumRowsMessage message is received

```

1 pub fn check_missing_concat_r(ctx: &mut Context, peer_id: usize) -> bool {
2     let peer: &mut PGlmPeer = get_peer_of_type!(ctx, peer_id, PGlmPeer).expect("peer
           should exist");
3
4     let iter = peer.state.model.iter;
5
6     let mut concat_r_to_req: Vec<usize> = vec![];
7     for neigh_id in peer.state.nodes.iter() {
8         if !peer
9             .state

```

```

10         .r_matrices
11         .entry(iter)
12         .or_default()
13         .contains_key(neigh_id)
14         && *neigh_id != peer_id
15     {
16         concat_r_to_req.push(*neigh_id);
17     }
18 }
19
20 let hash = peer.state.hash;
21
22 if !concat_r_to_req.is_empty() {
23     log::trace(
24         ctx,
25         format!("peer {peer_id} has missing r_matrices[{iter}] {concat_r_to_req:?}")
26         ,
27     );
28     let msg = ReqConcatMessage {
29         needs: concat_r_to_req.clone(),
30         iter,
31         hash,
32     };
33     broadcast_concat_req(ctx, peer_id, msg);
34     true
35 } else {
36     false
37 }

```

Listing B.73: A method which checks for and requests missing R matrices

```

1 pub fn receive_concat_r_req_msg(
2     ctx: &mut Context,
3     peer_id: usize,
4     requester_id: usize,
5     msg: ReqConcatMessage,
6 ) {
7     let peer: &mut PGlmPeer = get_peer_of_type!(ctx, peer_id, PGlmPeer).expect("peer
8         should exist");
9
10    let peer_iter = peer.state.model.iter;
11    let msg_iter = msg.iter;
12
13    if msg_iter > peer_iter {
14        log::trace(
15            ctx,
16            format!(
17                "peer {peer_id} received a request for R local matrix in iter {msg_iter}
18                but is on iter {peer_iter}",
19            ),
20        );
21        return;
22    }
23 }

```

```

21
22     let has: Vec<usize> = peer
23         .state
24         .r_matrices
25         .entry(msg_iter)
26         .or_default()
27         .keys()
28         .copied()
29         .collect();
30
31     log::trace(
32         ctx,
33         format!(
34             "peer {peer_id} received a request for R local matrix on iter {msg_iter}
              from {requester_id} for {rs:?}, it has: {has:?}" ,
35             rs = msg.needs
36         ),
37     );
38
39     for p_id in msg.needs {
40         let peer: &mut PGlmPeer =
41             get_peer_of_type!(ctx, peer_id, PGlmPeer).expect("peer should exist");
42
43         let mut r_to_send: Mat<f64> = Mat::new();
44         let hash = peer.state.hash;
45
46         let mut has_r = false;
47         if let Some(r) = peer
48             .state
49             .r_matrices
50             .entry(msg.iter)
51             .or_default()
52             .get(&p_id)
53         {
54             r_to_send = r.clone();
55             has_r = true;
56         } else if p_id == peer_id {
57             panic!(
58                 "peer_id {peer_id} r_local must exist in r_matrices r_matrice msg_iter {
                  msg_iter} r_matrices: {:?} r_matrices[{msg_iter}]: {:?}",
59                 peer.state.r_matrices.keys().copied().collect::<Vec<_>>(),
60                 peer.state
61                     .r_matrices
62                     .entry(msg_iter)
63                     .or_default()
64                     .keys()
65                     .collect::<Vec<_>>(),
66             );
67         }
68
69         if has_r {
70             engine::send_message_to(
71                 ctx,
72                 peer_id,
73                 requester_id,
74                 PGlmConcatMessage {
75                     origin: p_id,

```

```

76         r_remote: r_to_send,
77         iter: msg.iter,
78         hash,
79     },
80 );
81 }
82 }
83 }

```

Listing B.74: The function called when a ReqConcatMessage message is received

```

1 fn handle_iter(ctx: &mut Context, peer_id: usize, sender: usize, r_remote: Mat<f64>,
2   iter: usize) {
3     let peer: &mut PGlmPeer = get_peer_of_type!(ctx, peer_id, PGlmPeer).expect("peer
4       should exist");
5     if peer.state.r_matrices.contains_key(&iter)
6       && !peer
7         .state
8         .r_matrices
9         .get(&iter)
10        .unwrap()
11        .contains_key(&sender)
12    {
13        peer.state
14            .r_matrices
15            .get_mut(&iter)
16            .and_then(|r| r.insert(sender, r_remote));
17
18        let current_iter = peer.state.model.iter;
19        let msgs_to_receive = peer.state.nodes.len();
20        let msgs_received_in_iter = peer.state.r_matrices.get(&iter).unwrap().keys().len();
21
22        if iter == current_iter && msgs_to_receive == msgs_received_in_iter {
23            let all_r_remotes = peer
24                .state
25                .r_matrices
26                .get(&iter)
27                .unwrap()
28                .values()
29                .cloned()
30                .collect::<Vec<Mat<f64>>>();
31
32            let r_local_with_all_r_remotes = mat_cat_vec(&all_r_remotes, CatDim::
33                Vertical);
34
35            let (r_local, beta, stop) = generalized_linear_model::
36                distributed_single_solve_n(
37                    &r_local_with_all_r_remotes,
38                    &peer.state.model.coefficients,
39                    peer.state.model.family,
40                    peer.state.total_nrow,
41                    generalized_linear_model::DEFAULT_MAXIT,

```

```

39         generalized_linear_model::DEFAULT_TOL,
40         peer.state.model.iter,
41     );
42
43     peer.state.model.r_local = r_local.clone();
44     peer.state.model.coefficients = beta.clone();
45     peer.state.model.iter += 1;
46     peer.state.finished = stop;
47
48     let new_current_iter = peer.state.model.iter;
49     // add r_local to r_matrices
50     peer.state
51         .r_matrices
52         .entry(new_current_iter)
53         .or_default()
54         .insert(peer_id, r_local);
55
56     if !stop {
57         peer.state.model.r_local = generalized_linear_model::
58             distributed_single_iter_n(
59                 peer.state.model.family,
60                 &peer.state.data.x,
61                 &peer.state.data.y,
62                 beta,
63             );
64         // add r_local to r_matrices
65         peer.state
66             .r_matrices
67             .entry(new_current_iter)
68             .or_default()
69             .insert(peer_id, peer.state.model.r_local.clone());
70
71         log::info(
72             ctx,
73             format!(
74                 "peer {peer_id} broadcasting as it starts iteration {
75                     new_current_iter}"
76             ),
77         );
78         check_missing_concat_r(ctx, peer_id);
79     } else {
80         log::info(ctx, format!("peer {peer_id} finished on iteration {iter}"));
81     }
82 } else if msgs_to_receive == msgs_received_in_iter {
83     log::warn(
84         ctx,
85         format!(
86             "peer {peer_id} handle_iter, received all messages from iteration {
87                 iter} but current iteration is {current_iter}",
88             ),
89     );
90 } else {
91     log::trace(
92         ctx,
93         format!(

```

```

93         "peer {peer_id} handle_iter, received ({msgs_received_in_iter}/{
           msgs_to_receive}) messages on iteration {iter} (current
           iteration is {current_iter})",
94     ),
95 );
96 }
97 } else {
98     log::warn(
99         ctx,
100         format!(
101             "peer {peer_id} handle_iter failed because r_remotes[{iter}] already
              contains sender {sender}"
102         ),
103     );
104 }
105 }

```

Listing B.75: The function which handles the algorithm's iterations

```

1 pub struct PGlmPeer {
2     pub peer_info: PeerInfo,
3     pub state: PGlmState,
4 }
5
6 define_custom_peer!(PGlmPeer);
7
8 impl PGlmPeer {
9     pub fn new(pos_x: f64, pos_y: f64, x: Mat<f64>, y: Mat<f64>) -> Self {
10         let (r, c) = x.shape();
11         let beta: Mat<f64> = Mat::zeros(c, 1);
12
13         let family: FamilyEnum = FamilyEnum::Binomial;
14
15         let r_local =
16             generalized_linear_model::distributed_single_iter_n(family, &x, &y, beta.
                clone());
17
18         let initial_model = GeneralizedLinearModel {
19             r_local,
20             coefficients: beta.clone(),
21             family,
22             iter: 0,
23         };
24
25         Self {
26             peer_info: PeerInfo::new(pos_x, pos_y, 0.0)
                .with_on_message_receive(callbacks::on_message_receive),
27             state: PGlmState {
28                 model: initial_model.clone(),
29                 initial_model,
30                 data: ModelData { x, y },
31                 r_n_rows: HashMap::new(),
32                 r_matrices: HashMap::new(),
33                 local_nrow: r,
34             }

```

```

35         total_nrow: r,
36         nodes: BTreeSet::new(),
37         neighbors: Vec::new(),
38         finished: false,
39         hash: 0,
40     },
41 }
42 }
43
44 ...
45
46 }

```

Listing B.76: The PGlmPeer struct and its implementation

```

1 pub struct PGlmState {
2     pub initial_model: GeneralizedLinearModel,
3     pub model: GeneralizedLinearModel,
4     pub data: ModelData,
5     pub r_n_rows: HashMap<usize, usize>, // how many rows remotes have
6     pub r_matrices: HashMap<usize, HashMap<usize, Mat<f64>>>,
7     pub local_nrow: usize,
8     pub total_nrow: usize,
9     pub nodes: BTreeSet<usize>,
10    pub neighbors: Vec<usize>,
11    pub finished: bool,
12    pub hash: u64,
13 }

```

Listing B.77: The PGlmState struct

```

1 #[derive(Debug, Clone)]
2 pub struct TimeoutTimer {
3     pub interval: f64,
4 }
5
6 impl Timer for TimeoutTimer {
7     fn fire(&self, ctx: &mut Context) {
8         for peer_id in 0..ctx.peers.len() {
9             if ctx.peers[peer_id].is::<PGlmPeer>() {
10                 let peer: &mut PGlmPeer =
11                     get_peer_of_type!(ctx, peer_id, PGlmPeer).expect("peer should exist")
12                     ;
13                 if !peer.is_alive() {
14                     continue;
15                 }
16                 algorithms::timeout(ctx, peer_id);
17             }
18         }
19         let all_peers_finished: bool = ctx

```

```

20         .peers
21         .iter()
22         .filter_map(|p| p.downcast_ref::<PGlmPeer>())
23         .all(|p| p.state.finished);
24
25         if !all_peers_finished {
26             engine::add_timer(ctx, ctx.clock + self.interval, self.clone());
27         }
28     }
29 }

```

Listing B.78: The TimeoutTimer struct

```

1 pub fn timeout(ctx: &mut Context, peer_id: usize) {
2     if check_missing_sum_rows(ctx, peer_id) {
3         return;
4     }
5     check_missing_concat_r(ctx, peer_id);
6 }

```

Listing B.79: A method which makes the peers request missing information

B.2.3.1 Experiments

```

1 seed: &default_seed 2586804559409604052
2 arrival_time: &default_arrival_time distance
3 topology: &default_topology
4     n_peers: 7
5     name: full
6 deadline: &default_deadline 500
7
8 experiments:
9 - name: default_run
10     scenario: proposed_dglm
11     arrival_time: *default_arrival_time
12     topology: *default_topology
13     repetitions: 100
14     deadline: *default_deadline
15
16 - name: run_100_peers
17     scenario: proposed_dglm
18     seed: *default_seed
19     arrival_time: *default_arrival_time
20     topology:
21         n_peers: 100
22         name: full
23     deadline: *default_deadline
24
25 - name: constant_latency
26     scenario: proposed_dglm

```

```

27 seed: *default_seed
28 arrival_time: constant
29 topology: *default_topology
30 deadline: *default_deadline
31
32 - name: custom_latency
33   scenario: proposed_dglm
34   seed: *default_seed
35   topology:
36     n_peers: 7
37     name: twowaycustom
38     connections:
39       - [0, 1, !Full {bandwidth: 80000, latency: 0.02} ]
40       - [0, 2, !Full {bandwidth: 80000, latency: 0.03604} ]
41       - [0, 3, !Full {bandwidth: 80000, latency: 0.04768} ]
42       - [0, 4, !Full {bandwidth: 80000, latency: 0.04768} ]
43       - [0, 5, !Full {bandwidth: 80000, latency: 0.03604} ]
44       - [0, 6, !Full {bandwidth: 80000, latency: 0.02} ]
45       - [1, 2, !Full {bandwidth: 80000, latency: 0.02} ]
46       - [1, 3, !Full {bandwidth: 80000, latency: 0.03604} ]
47       - [1, 4, !Full {bandwidth: 80000, latency: 0.04768} ]
48       - [1, 5, !Full {bandwidth: 80000, latency: 0.04768} ]
49       - [1, 6, !Full {bandwidth: 80000, latency: 0.03604} ]
50       - [2, 3, !Full {bandwidth: 80000, latency: 0.02} ]
51       - [2, 4, !Full {bandwidth: 80000, latency: 0.03604} ]
52       - [2, 5, !Full {bandwidth: 80000, latency: 0.04768} ]
53       - [2, 6, !Full {bandwidth: 80000, latency: 0.04768} ]
54       - [3, 4, !Full {bandwidth: 80000, latency: 0.02} ]
55       - [3, 5, !Full {bandwidth: 80000, latency: 0.03604} ]
56       - [3, 6, !Full {bandwidth: 80000, latency: 0.04768} ]
57       - [4, 5, !Full {bandwidth: 80000, latency: 0.02} ]
58       - [4, 6, !Full {bandwidth: 80000, latency: 0.03604} ]
59       - [5, 6, !Full {bandwidth: 80000, latency: 0.02} ]
60   deadline: *default_deadline
61
62 - name: ring_topology
63   scenario: proposed_dglm
64   seed: *default_seed
65   arrival_time: *default_arrival_time
66   topology:
67     n_peers: 7
68     name: ring
69   deadline: *default_deadline
70
71 - name: custom_ring_topology
72   scenario: proposed_dglm
73   topology:
74     n_peers: 7
75     name: twowaycustom
76     connections:
77       - [0, 1, !Full {bandwidth: 80000, latency: 0.02} ]
78       - [1, 2, !Full {bandwidth: 80000, latency: 0.02} ]
79       - [2, 3, !Full {bandwidth: 80000, latency: 0.02} ]
80       - [3, 4, !Full {bandwidth: 80000, latency: 0.02} ]
81       - [4, 5, !Full {bandwidth: 80000, latency: 0.02} ]
82       - [5, 6, !Full {bandwidth: 80000, latency: 0.02} ]
83       - [6, 0, !Full {bandwidth: 80000, latency: 0.02} ]

```

```

84     deadline: *default_deadline
85
86 - name: star_topology
87     scenario: proposed_dglm
88     arrival_time: *default_arrival_time
89     topology:
90         n_peers: 7
91         name: star
92     repetitions: 10
93     deadline: *default_deadline
94
95 - name: peer_killed
96     scenario: proposed_dglm
97     seed: *default_seed
98     arrival_time: *default_arrival_time
99     topology: *default_topology
100    extra_args:
101        kill_peer: true
102    deadline: *default_deadline
103
104 - name: drop_rate_0_2
105     scenario: proposed_dglm
106     seed: *default_seed
107     arrival_time: *default_arrival_time
108     topology: *default_topology
109     drop_rate: 0.2
110     deadline: *default_deadline
111
112 - name: duplicate_rate_0_2
113     scenario: proposed_dglm
114     seed: *default_seed
115     arrival_time: *default_arrival_time
116     topology: *default_topology
117     duplicate_rate: 0.2
118     deadline: *default_deadline
119
120 - name: jitter
121     scenario: proposed_dglm
122     arrival_time: *default_arrival_time
123     topology: *default_topology
124     jitter:
125         distribution: !Weibull [1.064, 2.872]
126         multiplier: 0.001
127     repetitions: 100
128     deadline: *default_deadline

```

Listing B.80: The proposed solution's configuration file

```

1  #[derive(Debug, Clone)]
2  pub struct ReviveTimer {
3      pub target: usize,
4  }
5
6  impl Timer for ReviveTimer {

```

```
7     fn fire(&self, ctx: &mut Context) {  
8         if let Some(target) = ctx.peers.get_mut(self.target) {  
9             target.revive();  
10            log::info(ctx, format!("Peer {} revived.", self.target));  
11            algorithms::get_node_ids(ctx, self.target);  
12            algorithms::broadcast_sum_rows(ctx, self.target);  
13        }  
14    }  
15 }
```

Listing B.81: The ReviveTimer struct