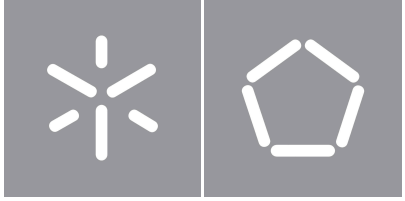
University of Minho
School of Engineering

Luís Filipe Cruz Sobral

Simulation of epidemic protocols

november 2023



University of Minho
School of Engineering

Luís Filipe Cruz Sobral

Simulation of epidemic protocols

Masters Dissertation
Integrated Master's in Informatics Engineering

Dissertation supervised by
Ricardo Manuel Pereira Vilaça

november 2023

Copyright and Terms of Use for Third Party Work

License granted to users of this work:

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Ricardo Manuel Pereira Vilaça, for his unwavering support and mentorship during the entire research process. Ricardo's expertise, wisdom, and dedication played a key role in shaping the direction and success of this study. His insightful feedback and constructive criticism have been instrumental in refining the content and methodology of this article.

Furthermore, I wish to thank INESC TEC for providing the necessary resources, facilities, and funding to make this research possible. Without their support, this project would not have been achievable.

Finally, I am deeply grateful to my friends and family for their encouragement and understanding during the often-demanding phases of this research. Their unwavering support has been a constant source of motivation and strength. Their belief in my abilities and willingness to stand by my side through the challenges and triumphs are priceless.

In particular, I want to thank my parents for their continuous support and dedication, and for providing everything I needed to be successful. Your individual efforts have made a profound impact on my work.

This article stands as a testament to the collective effort and commitment of these individuals and entities. Thank you for making this research journey a rewarding and instructive experience.

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020.

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, Braga, november 2023

Luís Filipe Cruz Sobral

Abstract

We live in a digital era, in a world connected by technology. The incredible capabilities of our mobile phones and computers let us communicate and get data from all over the globe, in the instance of a millisecond. However, technological progress doesn't stop. We persist in looking for faster connections, innovative applications and platforms, more efficient, scalable, and resilient. Distributed systems are the fundamental basis driving this progress in several scientific and industry fields. Epidemic protocols are crucial to ensure efficient data dissemination on these systems, providing fault tolerance, scalability, and availability. Its relevance grows as networks become more dynamic and distributed, playing a main role in ensuring the reliability and efficient operation of these systems.

Progress is not possible without studies and experimental evaluation of proposed algorithms. Although, as they are projected to systems compromising millions of nodes and processes, these studies are almost impossible at this scale, so most rely on simulation. Discrete-event simulation is one of the major experimental methodologies in several scientific and engineering domains. The used simulator is often seen as a technical detail, and many researchers develop their custom tool. Simulation tools vary in complexity and application, catering to a wide range of industries and research domains. The choice of a specific tool depends on the nature of the simulation, the problem being addressed, and the preferences and expertise of the user.

In this dissertation, we present, analyze, and compare a set of selected simulation tools, to choose the one that better fits epidemic protocol simulations in **P2P** systems. After choosing the most adequate simulation tool, we defined a generic simulation framework for epidemic protocols, and implementations of two different peer sampling services and one dissemination protocol. Leveraging this framework, we perform a extensive evaluation of the different protocols.

Keywords Distributed and Parallel Computing, Discrete-Event Simulation, Epidemic protocols, Peer Sampling Service, Performance, Scalability

Resumo

Atualmente, vivemos na era digital, num mundo conectado pela tecnologia onde os nossos telemóveis e computadores pessoais possuem capacidades incríveis que nos permite, em milésimas de segundo, comunicar e obter informações vindas dos 4 cantos do mundo. No entanto, o avanço tecnológico não para, continuamos incessantemente à procura de conexões mais rápidas, aplicações e plataformas inovadoras, mais eficientes, mais escaláveis e mais resilientes. Os sistemas distribuidos são a base fundamental que impulsiona todo este avanço em diversas áreas da ciência e da indústria. Os protocolos epidémicos são essenciais para garantir a disseminação eficaz de informações nestes sistemas, fornecendo tolerância a falhas, escalabilidade e disponibilidade. A sua importância cresce à medida que as redes se tornam mais dinâmicas e distribuídas, desempenhando um papel crítico em garantir a confiabilidade e o funcionamento eficaz desses sistemas.

O avanço não é possível sem o estudo e avaliação experimental de novos algoritmos e protocolos. Porém, sendo estes projetados para sistemas distribuídos compostos por milhões de nós e processos, é quase impossível testá-los a esta escala, por isso a sua maioria depende da simulação. A simulação por eventos é uma das principais metodologias experimentais no domínio da ciência e da engenharia. Temos à nossa disposição várias ferramentas de simulação que variam na sua complexidade e areas de aplicação. Contudo nem sempre é fácil escolher a ferramenta mais adequada e muitos investigadores acabam por desenvolver o seu próprio simulador.

Nesta dissertação, apresentamos, analisamos e comparamos um conjunto de ferramentas de simulação selecionadas, de modo a escolher a ferramenta que melhor se adequa à simulação de protocolos epidémicos. Após escolher a ferramenta mais adequada, definimos uma framework de simulação genérica, e implementação de 2 serviços de amostragem de nós e um protocolo epidémico. Aproveitando esta framework, realizamos uma avaliação extensiva dos diferentes protocolos

Palavras-chave Computação Distribuída e Paralela, Simulação por eventos, Protocolos epidémicos, Serviço de amostragem de nós, Performance, Escalabilidade

Contents

List of Figures

List of Tables

Acronyms

DES Discrete-event simulation. 1, 3, 5, 6, 56

HPV HyParView. 47, 48, 50, 51, 52, 53, 54

LDH Last delivery hop. 17, 46, 47, 48, 51

MPI Message-Passing Interface. 8, 10, 35

P2P Peer-to-peer. iv, 2, 6, 18, 19, 20, 56

PDES Parallel Discrete-event simulation. 8, 35

RMR Relative message redundancy. 17, 47, 51

SPMD Single Program Multiple Data. 8, 10

TCP Transmission Control Protocol. 13, 14, 39, 40, 41, 43, 49, 52, 53, 54

Chapter 1

Introduction

Nowadays, parallel and distributed computing platforms have a huge importance in a wide range of contexts and applications. You can see it on telephone or computer networks, multiplayer online games, virtual reality, aircraft control systems, peer-to-peer applications, and many other platforms.

High-performance computing (HPC) has played a significant role in consuming and driving these platforms. Commodity clusters, built from off-the-shelf computers linked together through switches, have been employed in a wide range of scientific and engineering fields. Moreover, there are ongoing discussions about developing exascale systems with millions of processing cores [1]. Distributed applications and platforms have also seen an increase in popularity in the peer-to-peer and volunteer computing areas. At the same time, with the impressive capacities of our personal computers and the speed of personal Internet connections, the biggest challenges have emerged. Some of the most relevant ones include resource management, application scheduling, data management, decentralized algorithms, electrical power management, fault tolerance, scalability, and performance.

Studying parallel and distributed applications and platforms sometimes requires empirical evaluation of proposed algorithms. Gossip protocols, in particular, have been used in large-scale data centers [2] and blockchains and are known for their efficiency, scalability, and simplicity. Researchers from the Peer-to-Peer (P2P) community aim at constituting distributed systems comprising millions of processes collaborating to a common goal. Studies on real systems became nearly impossible at this scale, so most rely on simulation.

Discrete-event simulation (**DES**) [3] is one of the main experimental methodologies in several scientific and engineering domains. Parallel Discrete Event Simulation (PDES) [4] has been a very active research field for at least three decades to surpass speed and size limitations.

1.1 Problem

In the context of **Peer-to-peer** (**P2P**) protocols, most studies rely on simulation. Scalability is considered one of the major quality metrics, although most of the discrete event simulators remain single-threaded and none of the mainstream **P2P** simulators allow Parallel Simulation.

Many factors can influence the performance of a gossip protocol, including the network topology, the rate at which messages are transmitted, and the rules for deciding which messages to forward. A discrete event simulation can help explore the algorithm's behavior under different conditions and optimize its performance.

The used simulator is often seen as a technical detail, and many researchers develop their custom tool. Simulation tools vary in complexity and application, catering to a wide range of industries and research domains. The choice of a specific tool depends on the nature of the simulation, the problem being addressed, and the preferences and expertise of the user.

Gossip simulation models provide a practical way to gain insights into the behavior of massive distributed systems and to test various strategies and protocols.

1.2 Objectives

This thesis aims to design and build a flexible simulation model for gossip protocols capable of generating accurate results for different real scenarios. This model should take into account the scalability, performance, and fault tolerance required by this type of protocols. Some main objectives we pretend to achieve are:

- Evaluation of the main simulation tools. Choose 3 or 4 simulation tools and make a theoretical and empirical evaluation to select the one that better fits epidemic protocol simulation in peer-to-peer systems.
- Build a generic simulation model for gossip protocols using a suitable simulation platform. Select 2 peer sampling services and a dissemination protocol to implement.
- Test and compare the produced work with other existing solutions, in a suitable environment.
- Experimental evaluation and possible optimization of the proposed simulation model. Choose some main metrics to help explain the obtained results.

1.3 Results

As a result of this work, we obtained a benchmark to compare simulation tools that better suit epidemic protocols simulation, focused on scalability, memory usage efficiency, and CPU usage. This comparison considers a description of a gossip protocol implementation on each simulator in terms of the main difficulties, features, and code size and, then, an analysis of the results produced by several simulations under different settings.

Taking into account the mentioned benchmarking results, we designed a generic simulation model for epidemic protocols and implemented 2 peer sampling services and one dissemination protocol. This resulted in an empirical evaluation and analysis of these two combinations considering several configuration settings, and different membership and recovery strategies.

All implementations are available on https://github.com/luissobral4/EpidemicSimulationModels.

1.4 Structure

The following chapters compose this work: Chapter 1 - **Introduction**, Chapter 2 - **State of the Art**, Chapter 3 - **Simulators scalability evaluation**, Chapter 4 - **Simulation model**, and Chapter 5 - **Conclusions and Future Work**.

State of art, Chapter 2, introduces the **Discrete-event simulation** (**DES**) and associated paradigms. Describes existing simulation tools, their main features, advantages, and disadvantages. Explains gossip protocols and peer sampling services, highlighting their relevance and giving some examples. Finally, we present some Generic Gossip Frameworks.

Simulators scalability evaluation, Chapter 3, focuses on implementing the PlumTree protocol on 3 chosen simulation tools, considering some main questions related to data, communication, time, and the overlay. Additionally, we present and analyze the results obtained from several simulations of these implementations.

Simulation model, Chapter 4, considers the benchmarking results obtained to develop a more specific model using a suitable simulation tool. This model implements 2 selected peer sampling services to combine with the PlumTree gossip protocol and get comparative results. We provide an exhaustive explanation of both implementations and the metrics used in their evaluation. Ultimately, we ran several simulations and presented a detailed analysis, considering different configuration settings and distinct membership and recovery strategies. This evaluation was divided into 3 phases: graph properties, scalability, and

failure recovery.

Finally, Conclusions, Chapter 5, presents a conclusion about the developed work and provided results, also mentioning some considerations for future work.

Chapter 2

State of the Art

This chapter summarizes and presents the findings of our literature review for simulation models and gossip protocols. This work plays a significant role in this dissertation. To improve and surpass the current problems, it's required to choose the simulation tool that best suits gossip protocols in terms of scalability, performance, and other valuable evaluation metrics.

2.1 Discrete-Event Simulation

Discrete-event simulation (**DES**) [3] is a method used to model real-world systems as a discrete sequence of events in time. **DES** is used to model and analyze the behavior of systems that operate in a discrete event-driven mode. The system's progress is represented by a sequence of events that occur at specific times or orders. Each event corresponds to a change in the system state and may trigger other events to occur. Between consecutive events, there is no change in the system.

A Discrete-event simulation model is implemented using computer software, which allows the model to be run and the results to be analyzed and visualized. Many distinct tools and approaches for building and running discrete-event simulations are available, ranging from specialized simulation languages and software packages to general-purpose programming languages.

Discrete-event simulation is often used to study the behavior of complex systems that are difficult to analyze or experiment on real systems. It can provide insight into the performance of a system under different conditions and help optimize its design or operation.

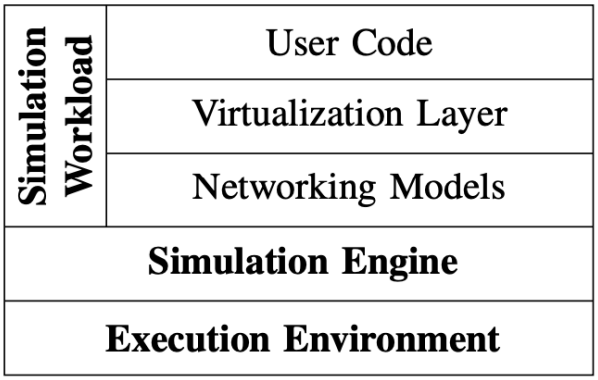


Figure 1: Layered View of **DES** [4]

Figure 1 is a layered view of classic Discrete-Event Simulation of Distributed Applications. As mentioned in [4], most of the time is spent in the layers built on top of the simulation engine. The time of the events is given by some hardware models of the network and computing resources. This model can be simplistic in **P2P** simulators (PeerSim [5], ProtoPeer [6], etc) or more complex in packet-level simulators (Bit-SiEm [7], NS2 [8]). On top of it, the virtualization layer is responsible for executing user code and converting its action into user requests that will be changed to events by the hardware models.

There are 3 major discrete-event simulation paradigms [3]:

(i) **Activity-Oriented Paradigm**

The activity-oriented paradigm is an approach to discrete-event simulation that focuses on the actions or activities that occur within the system being modeled. In this paradigm, the simulation model is constructed around the activities that take place in the system and the resources required to perform those activities.

The activity-oriented paradigm is often used to model complex systems with a high degree of concurrency, where multiple activities may be in progress at the same time and may interact with each other. It is particularly well-suited for modeling systems that involve processing items or objects, such as manufacturing systems or supply chains.

One key feature of the activity-oriented paradigm is the use of activity-based components, which represent the activities and resources in the system and define the rules for how they interact.

On activity-oriented paradigm, we break time into tiny increments. If, for example, the mean inter-arrival time is 20 seconds, we might break time into increments of 0.001. At each time point, our code looks around all the activities, checking for the occurrence of events or the completion of a

service. This will be very slow to execute once most time increments produce no state change to the system, and activity checks will waste processor time.

(ii) Event-Oriented Paradigm

The event-oriented paradigm is an approach to discrete-event simulation that focuses on the events that occur within the system being modeled. In this paradigm, the simulation model is constructed around the events that trigger changes in the state of the system and the relationships between those events.

This paradigm is often used to model systems driven by external events or inputs, such as communication networks or queueing systems. It is particularly well-suited for modeling systems that involve the flow of information or the processing of requests, like computer systems or service systems.

The event-Oriented Paradigm dramatically increases simulation speed compared to Activity-Oriented Paradigm. Instead of a small increment, we advance simulated time directly to the time of the next event. There is an event set with all pending events, and in each iteration, we update SimTime to the minimum among the scheduled event times and add a new event to the set. Compared to the process-oriented paradigm, this is easier to implement, faster to execute, and more flexible.

(iii) Process-Oriented Paradigm

The process-oriented paradigm is a simulation model based on the idea that simulation models consist of a collection of processes that interact with one another and with other elements of the simulated system. In this paradigm, the focus is on modeling the processes that make up the system, rather than on modeling the individual events that occur within the system. Processes in a process-oriented simulation model are usually represented as entities that move through the system, performing activities and interacting with other entities and resources as they go. Here, each simulation activity is modeled by a process. The process-oriented paradigm produces more modular code, and it's probably easier to write and easier for others to read. This paradigm is considered more elegant and is the more popular of the two main world views today (Process-Oriented Paradigm and Event-Oriented Paradigm).

2.1.1 Parallel Discrete-Event Simulation

Parallel Discrete-event simulation (**PDES**) [4] is a powerful technique for modeling and studying complex dynamic systems. It is useful in scenarios where traditional sequential simulations may be impractical due to the computational intensity or scale of the system being analyzed. **PDES** introduces parallel computing to process events concurrently on multiple processors or threads, potentially speeding up the simulation significantly. However, it requires a careful balance between the benefit gained from parallelism and synchronization overhead. **PDES** also introduces some new challenges, including load balancing, managing shared resources, and synchronization mechanisms to ensure that events occurring at the same simulated time are processed in the correct order.

Message-Passing Interface (**MPI**) [9] can be a powerful way to distribute the simulation across multiple processors. The simulation is divided into logical units that can be processed independently by different MPI processes. These units can be regions of a simulated space, different components of the system, or separate entities within the simulation. Each MPI process will be responsible for simulating a portion of the system or handling a subset of the discrete events. **MPI** can also be used to exchange information and synchronize between processes, ensure workload is distributed evenly across MPI processes, and run on parallel machines, in a multi-node architecture.

Another common approach for parallel programming is using **Single Program Multiple Data** (**SPMD**). In this paradigm, multiple instances of the same program run concurrently, each working on a different input or a different portion of the same data. **SPMD** divide the simulation space or entities into multiple partitions, each representing a portion of the simulation world that will be processed by a separate process or instance of the program. These partitions generally refer to **MPI** processes, because the majority of **MPI** programs is based on this paradigm.

2.2 Simulation tools

2.2.1 SimPy

SimPy [10] is a process-based discrete-event simulation library based on standard Python. It is used to model and simulate the operation of systems that consist of processes that interact with one another and with other elements of the system. All processes live in an environment and interact with each other and with the environment via Events. Instead of using threads, as is the case for most process-oriented simulation packages, SimPy uses Python's generator capability. This allows the programmer to specify that

a function can be prematurely exited and later re-entered at the point of the last exit, enabling coroutines. The exit/re-entry points are marked by Python's yield keyword. SimPy also provides various shared resource types to model limited capacity congestion points. Simulation can be performed in real-time (wall clock time) or by manually stepping through events.

SimPy is based on the process-oriented paradigm for discrete-event simulation, and it provides several features and tools that make it easy to build and run simulation models. Some of the key features of SimPy include:

- **Process-based modeling**: SimPy allows you to define and simulate processes that interact with one another and with other elements of the system.
- **Event scheduling**: SimPy provides a flexible event scheduling mechanism that allows you to specify when events should occur in the simulation.
- **Resource modeling**: SimPy provides tools for modeling and simulating the allocation and use of shared resources, such as stores, containers, and resources shared by multiple processes.
- **Data collection and analysis**: SimPy provides tools for collecting data from the simulation and analyzing it to understand the behavior of the system being simulated.

SimPy is widely used in a variety of fields, including manufacturing, logistics, and service systems. It is a powerful and flexible tool for building and analyzing discrete-event simulation models.

2.2.2 Simulus

Simulus [11] is an open-source discrete-event simulator in Python. It implements a process-oriented simulation world-view with several advanced features to ease modeling and simulation tasks with both events and processes. Simulus is designed to be easy to use and to provide fast and efficient simulation of systems that consist of processes that interact with one another and with other elements of the system. There is also support for parallel and distributed simulation via concurrent execution of multiple simulators. These simulators can be created and run simultaneously on different processors or cores on the same or different machines in a cluster.

To understand how simulus handles parallel and distributed simulation, we first introduce the concept of a synchronized group of simulators, where time advances synchronously among the simulators in the group. A synchronized group of simulators can run sequentially or in parallel by running on shared-memory multiprocessors, on distributed-memory machines in a cluster, or a combination of both.

Simulus supports distributed simulation, allowing the synchronized group of simulators to be instantiated and run on a parallel cluster. When running on parallel machines, Simulus instances need to communicate using the **Message-Passing Interface** (**MPI**) under the hood.

Single Program Multiple Data (**SPMD**) is the most common parallel programming paradigm. In **SPMD**, multiple machines execute the same program simultaneously, each operating on a different input or a different portion of the same data. Simulus offers an **SPMD** programming style for distributed simulation on parallel computers, which can also be combined with shared-memory multiprocessing. [12]

Simulus is a relatively new simulation library, but it has gained popularity among Python users for its simplicity and efficiency. It is a good choice for building and analyzing discrete-event simulation models in Python.

2.2.3 SimGrid

SimGrid [13] is a framework for developing simulators of distributed applications targetting distributed platforms that allows you to build and simulate the behavior of distributed systems, such as computer networks, data centers, and other types of distributed systems. It is realistic, flexible, accurate, scalable, and the validity of its analytical models was thoughtfully studied, ensuring their realism. Experimental results show that the SimGrid sequential version is more scalable than the state-of-the-art simulators.

SimGrid is particularly relevant when:

- **Compare an application to another**: This is the classical use case for scientists who use SimGrid to test how the solution they contribute compares to the existing solutions from the literature.
- **Design the best-simulated platform for a given application**: Tweaking the platform file is much easier than building a new real platform for testing purposes. SimGrid also allows for the co-design of the platform, and the application by modifying both.
- **Debug real applications**: With SimGrid, you are clairvoyant about your reproducible experiments: you can explore every part of the system, and your probe will not change the simulated state.

SimGrid is widely used for research and development in distributed systems. It is a powerful and flexible tool for building and analyzing simulation models of distributed systems.

2.2.4 Simian

Simian [12] is a process-oriented, conservative parallel discrete event simulator. It is designed to be easy to use and to provide fast and efficient simulation and is implemented using interpreted languages and

just-in-time compilation techniques. Simian should be able to achieve an event rate comparable to the performance of highly optimized parallel simulators implemented in C or C++. The user can run the simulation mode in sequential mode or parallel, using MPI. Simian allows mixing both event-based and process-oriented simulation techniques in the same model, and it's extremely competitive in simulation speed in terms of events/second. The aggregation of the following features makes Simian unique:

- **Simple**: Simian has a minimalistic design. For example, its Python implementation consists of only three source files with less than 550 lines of code
- **User-Friendly**: Simian is designed for domain experts with minimal programming requirements. The large set of libraries for Python and its script-like approach to input handling and output visualization, allow the user to quickly come to tangible results.
- **Pragmatically Scalable**: Simian has full support for running on computing clusters of any size using MPI and adopts a simple barrier-based synchronization protocol.
- **Portable**: On most platforms, Simian runs right out of the box. Simian minimizes its dependency on third-party libraries. The only dependencies are for supporting data communications (MPI) and user threads (greenlet for Python). Co-routines are a standard feature of Lua.

2.3 Gossip protocols

Gossip protocols (SCAMP [14], Directional Gossip [15], etc, PlumTree [16]) are a class of distributed algorithms that are used to disseminate information or messages throughout a network of nodes or processors. In a gossip protocol, each node communicates with a small number of other nodes at regular intervals, and the information is spread throughout the network through a process of "gossip". Gossip protocols are designed to be scalable, efficient, and robust, and they are often used in distributed systems to disseminate information about system states, perform distributed computation, and support other types of distributed communication and coordination. There are many distinct variants of gossip protocols, which differ in the specific communication patterns and strategies used to spread information throughout the network. Some key considerations in the design of gossip protocols include the degree of decentralization, the rate of information dissemination, the amount of communication overhead, and the robustness of the protocol in the face of node failures and network partitioning.

2.3.1 Peer Sampling Service

The peer sampling service [17] is applied within a collection of nodes that constitute the domain of gossip-based protocols utilizing this service. Multiple gossip protocols can use the same sampling service concurrently since they share the same target group. The service's objective is to provide a participating node in a gossip-based application with a subset of peers belonging to the same group, enabling the node to transmit gossip messages.

This service plays a crucial role in gossip protocols. It forms the backbone for efficient communication and information dissemination in distributed systems.

Gossip protocols are used for decentralized information dissemination, where nodes in the network exchange information with their neighbors.

The Peer Sampling Service's importance can be understood by analyzing the following key elements:

- **Dynamic Network Topology:** Frequently, nodes may join or leave the network. A Peer Sampling Service provides a mechanism to maintain an up-to-date view of the overlay, by continuously sampling and refreshing neighbor nodes. This ensures that the gossip protocol adapts to changes in the network topology efficiently and maintains its effectiveness.

- **Fault Tolerance:** Peer Sampling Service contributes to fault tolerance by maintaining multiple random connections to peers. If a node fails or becomes unreachable, other links can still ensure the flow of information, reducing the impact of individual node failures on the overall system. Some membership services can detect node failures and use strategies to recover and maintain overlay connectivity.

- **Scalability:** In large-scale distributed systems, maintaining a complete view of all network nodes can be impractical due to memory and communication overhead. Peer Sampling Service helps in sampling a subset of nodes. This decreases significantly the number of connections and messages required for efficient information exchange. This ensures that the gossip protocol remains scalable even as the network grows.

- **Randomization and Load Balancing:** Gossip protocols benefit from the randomness introduced by the Peer Sampling Service. Randomly selecting nodes to exchange information helps load balancing, preventing any specific node from being overwhelmed with incoming gossip requests.

- **Privacy and Security:** By maintaining a random subset of connections, the Peer Sampling Service improves the network's privacy and security. It reduces the probability of malicious nodes

gaining complete knowledge of the entire network structure, making it harder for them to launch targeted attacks or compromise the system's integrity.

Overlay graphs should support fast dissemination and a high fault tolerance level to node failures. For this, there are some desirable properties that node views must own:

- **Connectivity**: The overlay defined by node views must be connected to avoid isolated nodes.
- **Degree distribution**: To improve fault tolerance, both in-degree (number of nodes that have a node *n* in their view) and out-degree (number of nodes in the node view) should be uniformly distributed across all nodes.
- **Average Path Length**: This property is closely related to the overlay diameter. To improve the overlay efficiency, low average Path Length values are essential.
- **Clustering Coefficient**: Indicates a density of neighbor relations across the neighbors of a node. It's a value between 0 and 1.

HyParView

HyParView [18] is a peer sampling service protocol that ensures high reliability even in the presence of high rates of node failures. It proposes a new approach that relies on the use of two distinct partial views (*Active View* and *Passive View*), maintained by different strategies. Its design provides a robust and efficient mechanism for managing membership in large-scale distributed systems using gossip-based communication.

Some **HyParView** [18] key features include:

- **Gossip-based Membership Management:** The protocol uses a gossip-based approach to manage the membership information in the system. This approach relies on using a reliable transport protocol, such as **TCP**, to gossip between peers. This way, the gossip does not need to be configured to mask network omissions.
- **Small symmetric active views:** Each node maintains a small symmetric *active view* with the length of *fanout* + 1. Assuming that the links do not omit messages, this strategy allows for using smaller *fanouts* than protocols that use unreliable transport to support gossip exchanges. The broadcast is performed by flooding the graph defined by the *active views*. While this graph is generated randomly (using our membership service), gossip is **deterministic** as long as the graph remains unchanged.

- **Failure detection:** **TCP** is also used as a **failure detector**, and since all members of the *active view* are tested at each gossip step, node failures are detected quickly.

- **Backup nodes:** Each node maintains a *passive view* of backup nodes that can be promoted to the *active view* when one of the nodes fails by either disconnecting, crashing, or blocking.

- **Partial View Maintenance:** A membership protocol is in charge of maintaining the *passive view* and selecting members to promote to the *active view*. A **reactive strategy** is used to preserve the *active view* that remains unaltered in stable conditions, while *passive View* is updated using a **cyclic strategy**.

Brahms

Brahms [19] tackles the challenge of selecting random peers within an unstructured network. According to the authors, this approach is robust to network changes and malicious actions. All while ensuring that each peer maintains a compact view of the network. [20]

Unlike push-only gossip, where the whole view is updated with pushes only, Brahms is **push–pull gossip**. A constant part of each view is updated with pushes, while the other part is updated with pulls.

While push flooding, in push-pull gossip, only affects a portion of the view, it still requires a logarithmic amount of time, relative to the view size, to corrupt the entire *view*. This problem becomes more pronounced because the other part of the view is updated with pull operations, which can suffer from skewed pulls, where faulty nodes may respond only with incorrect IDs to pull requests.

Brahms requires extra care to protect against poisoning of the views with faulty IDs and uses the following strategies:

- **Limited pushes**: Restricting the transmission of push messages (handled by send_lim) reduces the proportion of faulty pushes. Since push messages are sent without request, an adversary with unlimited capacity could flood the system with push requests, causing correct IDs to be primarily spread through pull operations and reducing their representation exponentially.

- **Attack detection and blocking**: Brahms protects against targeted attack by blocking the update of *V* if more than the expected $\alpha\ell 1$ pushes are received in a round. This policy hinders the pace of advancement, and its anticipated effect is limited when there are no attacks (nodes frequently recalculate the *View* in most rounds).

- **Controlling the contribution of pushes versus pulls**: Brahms updates *V* with randomly chosen $\alpha\ell 1$ pushed ids and $\beta\ell 1$ pulled ids. Views are at greater risk from neighbor-initiated pulls than adversarial pushes because all pushes from valid nodes are accurate. A pull from a randomly selected correct node may contain faulty IDs. It is essential to strike a balance in the contribution of both pushes and pulls to the View: Pushes should be controlled to safeguard the nodes they target, while pulls should also be constrained to protect the rest of the network.

- **History samples**: The attack detection and blocking techniques slow targeted attacks instead of preventing them. When an adversary tries to boost its presence in a victim's view through targeted pushes, it compels the victim to request more data from faulty nodes. As a result, the targeted node's view deteriorates, decreasing its transmission of pushes to valid nodes and fewer receptions of valid pushes, creating opportunities for more erroneous push attempts. Brahms effectively counters such attacks by implementing a self-repairing mechanism, whereby a portion γ of V reflects the history.

- **Parameter settings**: Brahms's parameters allow a trade-off between performance and resilience against Byzantine attacks. γ must not be too large since the algorithm needs to deal with churn, and it must not be too small, or the feedback will be ineffective. The choice of $\beta\ell 2$ and $\beta\ell 2$ is crucial for guaranteeing that a targeted attack can be contained until the attacked node's sample stabilizes.

2.3.2 Dissemination Protocol - PlumTree

Gossip protocols exhibit high message complexity in steady-state to ensure reliability. On the other hand, tree-based approaches have a small message complexity in steady-state but are very fragile in the presence of failures. PlumTree combines both approaches to use a low-cost scheme to build and maintain broadcast trees embedded on a gossip-based overlay.

PlumTree [16] protocol operates as any pure gossip protocol. To broadcast a message, each node gossips with *f* nodes provided by a peer sampling service (*f* is the protocol fanout). However, each node uses a combination of eager push gossip, for a subset of *f* nodes, and lazy push gossip for the remaining nodes. Plumtree protocol attempts to create a broadcast tree structure by using hybrid gossip. Eager push links are selected in a way that their closure effectively builds a broadcast tree embedded in the random overlay network. For lazy push, links are used to ensure gossip reliability in case of failure and to quickly heal the broadcast tree. The set of peers doesn't change at each gossip round like other gossip protocols

instead the same peers are used until failures are detected.

Plumtree has 2 main functions:

- **Tree construction** [2]: This component is in charge of selecting which links of the random overlay network will be used to forward the message payload using an eager push strategy. It aims to create a tree construction mechanism as simple as possible, with minimal overhead in terms of control messages.
- **Tree repair** [2]: This component repairs the tree when failures occur. The process should ensure that, despite failures, all nodes remain covered by the spanning tree. Therefore, it should be able to detect and heal partitions of the tree. The overhead imposed by this operation should also be as low as possible.

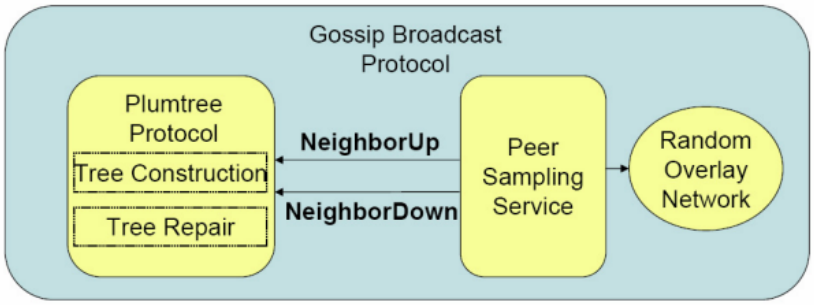


Figure 2: PlumTree architecture

Gossip and Tree Construction

The algorithm maintains two sets of peers for each node: eagerPushPeers, with which the node uses eager push gossip and lazyPushPeers, with which it uses lazy push gossip. In the beginning, eagerPushPeers contains *f* random peers obtained by the peer sampling service, and lazyPushPeers is empty. The protocol operates as pure push gossip protocol at the first rounds, and the fanout value f must be selected such that the overlay defined by the eagerPushPeers of all nodes is connected and covers all nodes. After this, nodes construct the spanning tree by moving neighbors from eagerPushPeers to lazyPushPeers, in such a way that, the overlay defined by the first set becomes a tree.

Fault Tolerance and Tree Repair

When failure is detected, at least one tree branch is affected. An eager push isn't enough to ensure message delivery, so lazy push messages exchanged through the remaining nodes of the gossip overlay

are used to recover missing messages and provide a quick mechanism to heal the multicast tree.

Essential Properties

- **Connectivity:** Overlay should be connected, despite failures that might occur. For this, all nodes should have, at least, another correct node in their partial views and they should be in the partial view of, at least, a correct node.
- **Scalable:** PlumTree protocol aims to support *f* large distributed applications. The peer sampling service 2 should be able to operate in systems with more than 10,000 nodes.
- **Reactive membership:** The stability of the spanning tree structure depends on partial views maintained by the peer sampling service. When a node is added or removed to the partial view of a given node, it could produce changes in the spanning tree, so the peer sampling service should employ a reactive strategy that maintains the same elements in partial views when operating in a steady state.

2.3.3 Metrics

To evaluate the key properties of gossip protocols and identify what could be improved in the solution being studied or compare it to other solutions, it is essential to identify some relevant metrics. [21]

- **Reliability**: Robustness is one of the key properties of these protocols and could be measured by the percentage of nodes that receive a given message.
- **Latency**: Latency is always an interesting metric for these systems and could be measured by the **Last delivery hop** (**LDH**), the number of rounds a gossip message took to be delivered to all nodes, multiplied for the gossip round time.
- **Redundancy**: As gossip protocols tend to generate redundant information, it is important to determine the overhead on the network by counting the number of redundant messages. **Relative message redundancy** (**RMR**) measures the average number of message copies (besides the first) that each node received and thus the overhead of the dissemination process, accounting for scenarios where not all the nodes received the information.
- **Average shortest path**: This metric quantifies the average number of rounds a message takes to be delivered.

- **Connectivity degree**: Refers to the number of connections a node has with other nodes in the overlay. We also calculate the node with the minimum and maximum degree for a degree distribution notion.

2.4 Generic Gossip Frameworks

2.4.1 GossipKit

GossipKit [22] is a comprehensive and unified framework designed to streamline the implementation of gossip-based protocols in distributed computing and peer-to-peer systems. Some key features and benefits:

- **Centralized Gossip Management**: GOSSIPKIT serves as a central hub for managing and coordinating gossip-based protocols, simplifying the development process and ensuring consistency in protocol execution.
- **Modular Components**: It provides a set of modular components that can be easily integrated into various applications. This modularity allows developers to select and combine components based on their specific requirements.
- **Scalability and Efficiency**: GOSSIPKIT is designed to handle large and dynamic networks efficiently, allowing for seamless scaling while minimizing the overhead of information dissemination.
- **Robustness and Fault Tolerance**: The framework enhances network robustness by facilitating the dissemination of information in a decentralized and resilient manner, even in the presence of node failures or network disruptions.

GOSSIPKIT simplifies the integration and management of gossip-based protocols in distributed systems, offering a flexible and unified framework that enhances scalability, robustness, and efficiency. It is a valuable resource for developers seeking to implement and fine-tune gossip protocols in their applications.

2.4.2 LUNES

LUNES [23] is a specialized simulation tool designed for modeling and analyzing **Peer-to-peer** (**P2P**) systems. This software serves as a valuable resource for researchers and practitioners in the field of distributed computing. Some LUNE's key features are:

- **Agent-Based Simulation**: LUNES employs an agent-based approach to simulate and model P2P systems. This means it represents individual agents or nodes within the network, allowing for granular analysis of their interactions and behaviors.
- **P2P System Modeling**: The software is tailored specifically for simulating P2P systems, making it a powerful tool for understanding the dynamics and performance of decentralized networks.
- **Network Customization**: LUNES offers flexibility in customizing the network topology and parameters. Users can design and simulate different types of P2P architectures, including structured and unstructured networks, to study their behavior under varying conditions.
- **Realistic Simulations**: The agent-based approach allows for realistic modeling of node behaviors, interactions, and communication protocols.
- **Data Collection and Analysis**: LUNES provides tools for data collection and analysis during simulations. Researchers can gather insights into network metrics, performance indicators, and emerging patterns, aiding in the evaluation of P2P system designs.

In summary, LUNES is an agent-based simulation tool designed for the comprehensive modeling and analysis of Peer-to-Peer systems. It offers a flexible and realistic platform for studying various aspects of P2P networks, aiding both researchers and educators in their exploration and understanding of decentralized computing environments.

2.5 Discussion

We consider there is a gap in the studies of epidemic protocol simulation. There are options available that can be adapted for such simulations, but none of them is specialized or tailored specifically for gossip protocols. We found it critical to develop a benchmark methodology for this type of protocol, taking into account some mainstream simulation tools. The chosen tools should let us develop efficient, adaptative, and scalable simulations compromising thousands of nodes, via a scripting language like Python so it could be easy and simple to implement.

We also believe that developing a specialized gossip protocol simulation model using a suitable simulation tool might be a valuable initiative for researchers and developers interested in this area. A generic simulation framework could offer predefined models, standard components for gossip communication, and built-in performance analysis tools, making it easier to design and analyze gossip-based systems.

Chapter 3

Simulators scalability evaluation

In this section, we will develop a benchmark methodology for simulation tools in the context of epidemic protocols in **P2P** systems. We will take into account each one of the 4 simulation tools mentioned above (*Simulus*, *Simian*, *SimPy* and *SimGrid*), considering their main features and limitations. All steps, decisions, difficulties, and choices we faced during this process will be detailed. We pretended to use Python simulation tools, so we decided to exclude SimGrid. Despite providing a Python development environment, it is implemented in C++, and the task of creating **P2P** platforms compromising thousands of Nodes (represented by *Actors*) dealing with resources such as Hosts, Links, and Disks revealed some complex problems.

After that, we will proceed to the execution of several tests to choose the simulation tool that better suits the epidemic protocols simulation. We need to compare them in similar environments and ensure that the different executions under the same overlay should produce several simulated events and exchanged messages as identically as possible. Once we verify this, we can start running simulations under different environments, considering that our final choice will be sustained by 3 key concerns: *scalability* of the tool under a large number of nodes, *memory efficiency usage* and ease of development.

The presented tests are the result of several executions on distinct machines, upon several settings (overlay size, seed for the overlay generation, distance to neighbors, message delivery delay, number of broadcast messages).

3.1 PlumTree implementation

Each simulation tool has its style, strategies, and features. We selected 3 process-oriented simulation tools in Python (*Simulus*, *Simian*, and *SimPy*) and implemented a Plumtree protocol on each one of them, which could be accessed on [24].

For these implementations, there are some main questions:

- **Data**: which data to store? which data to exchange? in which data structures? using which update strategies?
- **Communication**: when to exchange data? in which direction? using which communication feature? how to simulate sending delay?
- **Time**: How to measure time to execute simulation? How to create timers?
- **Overlay**: how to create similar overlays for the 3 simulation tools?

3.1.1 Data

To resolve some of the problems related to the data exchanged, we created the following data structures on all implementations:

- **eagerPushPeers**: list of peers that nodes will use for eager push gossip
- **lazyPushPeers**: list of peers that nodes will use for lazy push gossip
- **lazyQueues**: set of IHAVE messages
- **missing**: list of missing messages
- **receivedMsgs**: dictionary with received messages, for each message ID
- **timers**: dictionary with timers for each message ID

The class **Msg** defines the messages used to exchange data. These messages have 5 fields:

- **type**: there are 5 message types (PRUNE, IHAVE, GRAFT, GOSSIP, and BROADCAST)
- **payload**: carries the message data
- **ID**: Message ID
- **round**: message round
- **sender**: message sender

3.1.2 Messages

There are 5 message types exchanged between nodes (PRUNE, IHAVE, GRAFT, GOSSIP, and BROADCAST). BROADCAST messages are created by the system, and inform a node to start message propagation. The receiver adds it to its received messages and spreads it by eager and lazy push.

When a PRUNE message is received, the node removes the sender from its eagerPushPeers and adds it to lazyPushPeers.

Lazy push is implemented by sending IHAVE messages. A node that receives an IHAVE message adds it to its missing list and starts a timer for sending GRAFT messages asking for the lost message, if the message ID received isn't in the receivedMsgs list.

Upon receiving a GRAFT message, the node moves the sender to eagerPushPeers, the message ID is stored in the receivedMsgs list, and replies with a GOSSIP message with the message content.

When a GOSSIP message is received, if this message has not yet been received, the node adds it to its ReceivedMsgs list, clears all timers for that message, moves the sender to eagerPushPeers, and propagates the message through lazy and eager push, with the round incremented. Otherwise, if the message was already received, it moves the sender node to lazyPushPeers and replies with a PRUNE message.

3.1.3 Overlay

To create the overlay, we defined a matrix where we display nodes by generating coordinates x and y.

We want similar overlays on all implementations, so we need to set an input seed for random numbers generation and a distance to calculate neighbor peers. Each node neighbor's peers are those whose distances are less than the input distance given.

3.1.4 Simulus

Simulators in a synchronized group can send messages to named mailboxes that belong to other simulators. Considering multiprocessors and MPI versions of Simulus, we need to create a simulator for each Node with an associated mailbox to have a synchronized group of simulators that can run sequentially or in parallel.

Mailboxes were used to exchange information by direct-event scheduling. It can be considered a store with an infinite storage capacity, and allow nodes to schedule events on other mailboxes with a delay time, using the mailbox's send() method.

In Simulus, an event is simply a function to be invoked at a designated time. Simulus uses 2 main methods. *process()* method is used to create a process to execute a function on a given simulator at a given scheduled time. *sched()* method can be used for direct event scheduling and lets us schedule any function on a given simulator.

PlumTree implementation makes use of *process()* method to create a process to execute the *receive()* method, that will recursively look for received messages and implement each message strategy. *sched()* schedules a timer event upon receiving an IHAVE message. This event is stored on the timers list and can be canceled using *cancel()* method when receiving a GOSSIP message.

To monitor Simulus execution, Simulus has a *show_runtime_reportdispay()* method to display some relevant values (execution time, scheduled events, canceled events, created processes, finished processes, canceled processes, process context switches). We also used a Python timer to measure execution time, but this time was very similar to the execution time on the *runtime_reportdispay*.

This was the first implementation, and it took some time to understand discrete event simulation, so it took over two weeks to implement. Simulus features like mailboxes make communication between entities easier, and after we get used to Simulus simulation, it is revealed to be easy to model. Simulus plumTree has 276 code lines [24].

The code below presents some key details of the Simulus implementation:

```
# Node entity
    # Create mailbox
    self.mbox = sim.mailbox(name='mb%d'%idx, min_delay=lookahead)
    # send message
    self.sim.sync().send(self.sim, 'mb%d'% receiver, message,delay)
    # start receiving messages
    self.sim.process(self.receive)
    # schedule event Timer
    self.timers[msg.ID] = self.sim.sched(self.timer,until=self.sim.now+self
    .lookahead,mID=msg.ID)

nodes=[] # all nodes instantiated on this machine
sims = [] # all simulators instantiated on this machine

# create simulators and nodes
for s in range(BLOCK_LOW(rank, psize, args.nsims), BLOCK_LOW(rank+1, psize,
    args.nsims)):
    sim = simulus.simulator(name='sim%d'%s)
```

```
    sims.append(sim)
    for idx in range(BLOCK_LOW(s, args.nsims, args.total_nodes),
                     BLOCK_LOW(s+1, args.nsims, args.total_nodes)):
        nodes.append(node(sim, idx, args.total_nodes,args.lookahead))

syn = simulus.sync(sims)
syn.run(args.endtime)
```

3.1.5 SimPy

On SimPy, the behavior of active components (like vehicles, customers, or messages) is modeled with processes. All processes live in an environment and interact with each other and with the environment via events. Therefore, we started by creating an environment where all nodes will schedule events in this environment.

Processes are described by simple Python generators. A SimPy Process can be used like an event (technically, a process is an event). Some key concepts are *Environment.Process()* method, which we used to schedule events at the current simulation time, *Environment.timeout()* are used to simulate the passage of time. Events of this type are triggered after a certain amount of (simulated) time has passed and allow a process to sleep (or hold its state) for the given period. Another important concept is how to use *yield*. When a process yields an event, the process gets suspended and will be resumed when the event occurs. As SimPy runs sequentially, the process will pass the control flow back to the simulation once a yield statement is reached.

For process communication, we used a simple BroadcastPipe constructed from Store (Store is a SimPy resource used to model the production and consumption of concrete objects). There is a pipe per node, and each node can get messages from its own pipe and schedule event messages in other pipes, using the 'timeout' method to simulate delay time.

Nodes start by using *process()* method to create a process to execute function *receive()*. This function recursively looks for received messages (yield self.in_pipe.get(), creates a GET event, and suspends it until there is a message to receive) and implements each message strategy. *process()* is also used to create lazy and eager push events and to start a timer when receiving an IHAVE message. We use *yield env.timeout(delay)* to simulate delay times. Timer events are stored on a timer list, and when receiving a GOSSIP message they are interrupted using *Process.interrupt()* method.

SimPy doesn't have resources to monitor the execution, which means that to trace all events created (messages sent and received, canceled and executed events), the environment was associated with a trace

function that uses *Environment.step()* to trace all processed events. A Python timer is used to measure execution time.

This simulator was the most difficult one to implement. It took over 2 weeks, and concepts like yield, environments, timeouts to simulate passing time, and problems like how nodes will exchange messages and how to trace the simulation were a bit more complex to understand. SimPy Plumtree was implemented in 265 code lines [24], almost the same as Simulus Plumtree.

Here are some essential details of the SimPy implementation:

```
# Node entity
    # send a message
    yield env.timeout(self.lookahead)
    pipes[receiverNode].put(msg)
    # receive message
    msg = yield self.in_pipe.get()
    # start receiving messages process
    self.env.process(self.receive())

env = simpy.Environment()

# create an environment tracer
def monitor(data, t, prio, eid, event):
    if isinstance(event,simpy.resources.store.StoreGet):
        data[0] += 1  # msg received
    elif isinstance(event,simpy.resources.store.StorePut):
        data[1] += 1  # msg sent
    elif isinstance(event,simpy.events.Interruption):
        data[2] += 1  # timers cancelled
    elif isinstance(event,simpy.events.Timeout):
        data[3] += 1

trace(env, monitor)

# create broadcast pipes and nodes
pipes=[]
nodes=[]
for i in range(args.total_nodes):
    pipes.append(simpy.Store(env))
    nodes.append(Node(env,pipes[i],i,args.total_nodes,args.lookahead))
```

```
31
32 env.run(until=args.endtime)
```

3.1.6 Simian

Simian uses entities as objects containing event handling functions and can be distributed among the MPI ranks for parallel processing [25].

The essential method of the Entity class is *reqService*, which is used to schedule a future event at the entity to be processed by an event handler. If the event is destined for the same entity or one in the same logical process, Simian inserts the event in the local event queue. Otherwise, a timestamped message is sent to the appropriate logical process. The *attachService* method associates an event handler that processes events destined for this entity.

The entity class also has several methods that deal with simulation processes, but we decide to base our implementation on creating an *Entity* per node, and each node uses *reqService* method to schedule future events on the local event queue or in other nodes event queues.

Contrary to other implementations where we create a process to execute a *receive()* function that recursively looks for messages, this implementation is more simplistic. To transmit a message, a node schedules a *receive* event on the destination node, using the *reqServive* method. Timers implementation was as simple as using *reqService* to schedule a timer event on the same Entity and add it to the timers list. There isn't a method to interrupt events, so when a GOSSIP message is received, the message is removed from the timers list, and the timer function executes if the message ID is still in the timers list.

To start the messages broadcast, we use the *schedService* method of the Simian Engine to schedule a BROADCAST event on the node that will initiate the transmission.

Simian provides the execution time, number of simulated events, and events per second of each execution. Therefore, we can monitor executions without implementing extra monitoring functions.

This implementation was the simplest and easiest to implement and took less than a week. The biggest challenge was the MPI execution because the MPICH lib import path was not working, and we changed the Simian base code to import MPI from mpi4py. Simian PlumTree was implemented in 208 code lines [24], less than the other simulators, highlighting Simian simplicity.

Simian main details are depicted in the following code:

```
1 simianEngine = Simian(simName, startTime, endTime, minDelay, useMPI)
2
3 # Node entity
```

```
    # Schedule Receive event on the receiver node
    self.reqService(lookahead, "Receive", message, "Node", receiver)
    # Schedule Timer event
    self.reqService(lookahead, "Timer", msg.ID)

# Create nodes
for i in range(nodes):
    simianEngine.addEntity("Node", Node, i,i,nodes)

simianEngine.run()
```

3.2 Evaluation

3.2.1 Experimental Settings

All experiments were conducted in a network composed of 10.000, 62 500, 90 000, 160 000, and 250 000 nodes, and results show an aggregation from 3 runs of each experiment. This experiment compares the 3 chosen simulation tools regarding scalability and memory used. To monitor these simulations, we used a Python timer to measure the execution time, and **psrecord** [26] to record the memory usage.

The same peer sampling service [17] was used in all experiments. The distance matrix described above 3.1.3 defines the overlay, depending on the input **seed**, and the number of neighbors is given by the maximum **distance** input value.

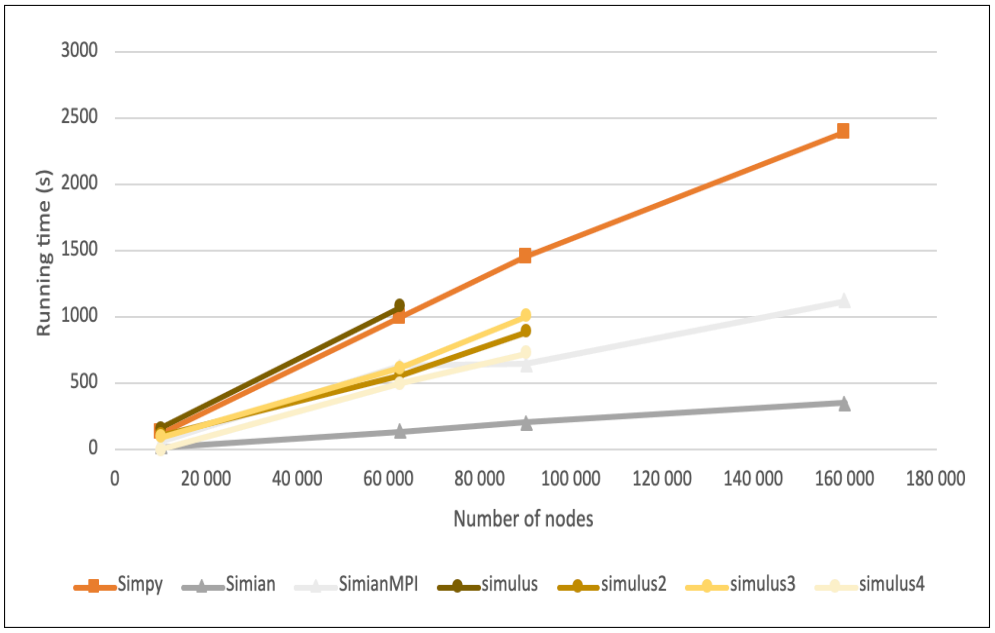
For the simulations, we did not use any piggybacking policy for the IHAVE messages. An IHAVE message is sent immediately in each lazy link. Almost all experiments show the results when sending one broadcast message per 10 seconds from multiple senders. In the end, we will test the best simulation tool scalability by starting 200 broadcast messages on a 1 msg/s rate. All simulations were executed for 200 simulation cycles.

3.2.2 Results

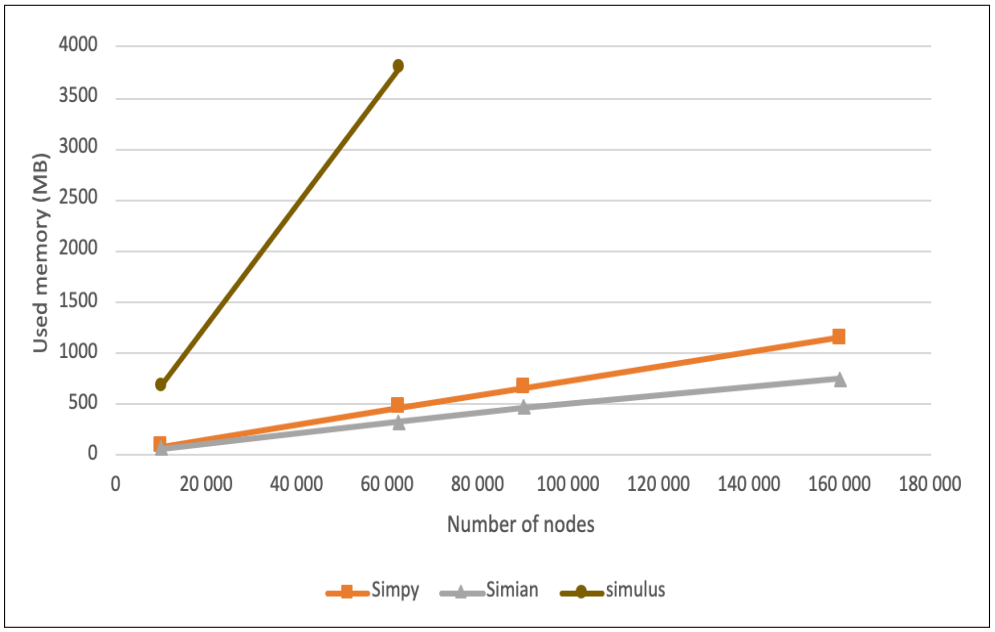
We started by testing Cluster 29, a machine with an Intel i3-2100 8GB HDD Processor, with the following specifications: 2 cores, 3.10GHz, 3 MB Intel® Smart Cache, 8GB RAM. We made several experiments on this machine under different input values for *lookahead, seed* and *distance* to measure the impact of each of these values on the simulation. Then we tried 2 other machines with an Intel i5 processor using the same values for *lookahead, seed*, and *distance*. Here, we tested the distributed versions of Simian

(using MPI) and Simulus versions (*multiprocessors, MPI* and *multiprocessors + MPI*). These versions were tested on a single machine and, using both, in a multi-node architecture. Finally, we performed specific tests using the tool that provided the best results.

Cluster 29



(a) Execution time using sequential and distributed versions



(b) Memory usage

Figure 3: Results for: Lookahead-0.1s Distance:100 Seed:10

Figure 3 shows the execution time and used memory for the 3 chosen simulators. We also considered the Simian MPI version and the 3 distributed versions of Simulus (2: using multiprocessors, 3: using MPI, 4: using MPI + multiprocessors). We tried different process numbers and selected the fastest to find the perfect number of processes for the distributed versions. *Simulus4 p2* means Simulus version using (MPI + multiprocessors) with 2 processors, and *Simulus3 p4* and *SimianMPI p4* refers to Simulus and Simian MPI versions with 4 processors.

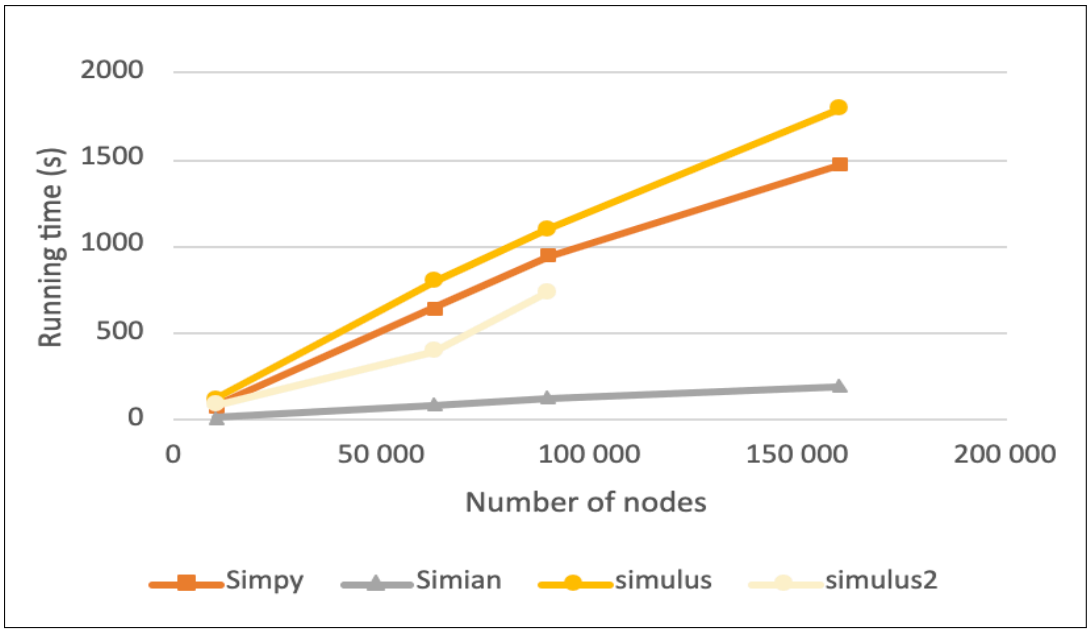
Under the base inputs used for these experiments (lookahead:0.1s distance:100 seed:10), in terms of execution time, the Simian sequential version was revealed to be the fastest one running in less than 6 minutes for 160,000 nodes. Then comes the Simian MPI version, almost 3 times slower. Simulus distributed versions didn't run for 160 000 nodes, but for 10 000, 62 500, and 90 000 the 3 versions obtained similar execution times, and these times were identical to the Simian MPI version.

Simulus sequential version and SimPy were the slowest and revealed similar execution times. However, Simulus wasn't able to run more than 90 000 nodes. For 160,000 nodes, SimPy took 40 minutes, which is seven times more than Simian.

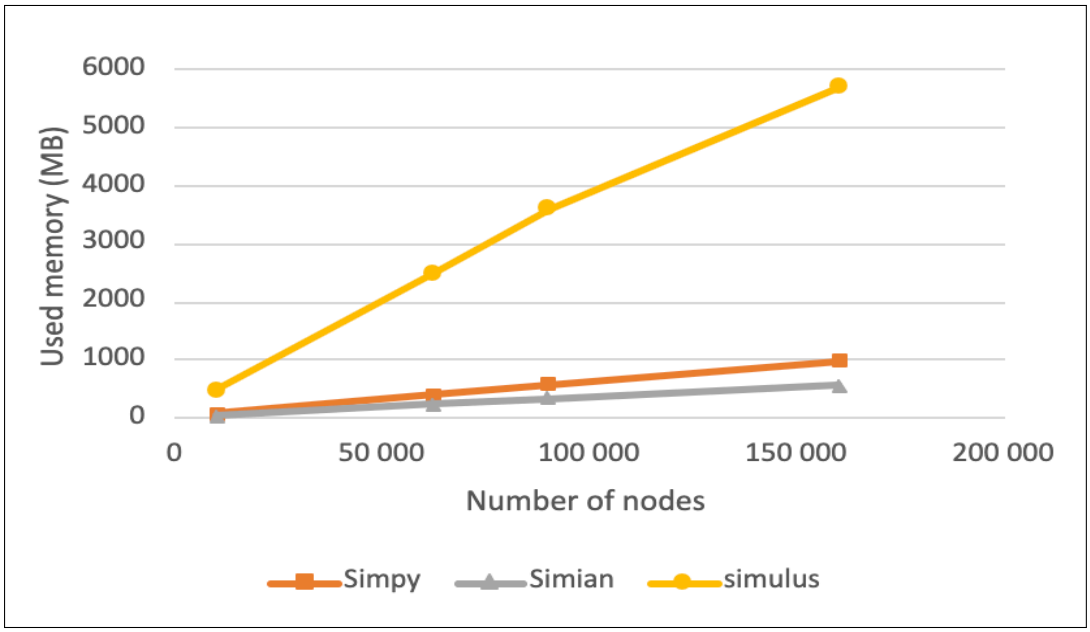
Simian was also the best on memory usage, with 750MB used for the 160,000 nodes simulation. SimPy revealed promising results, with 1150MB, and Simulus used a lot of memory, almost 10 times more than Simian.

All simulators revealed linear results for execution time and used memory.

After that, we tested simulations with different input values for seed, distance, and lookahead to see the impact of each value on the execution time and memory usage.



(a) Execution time when decreasing distance



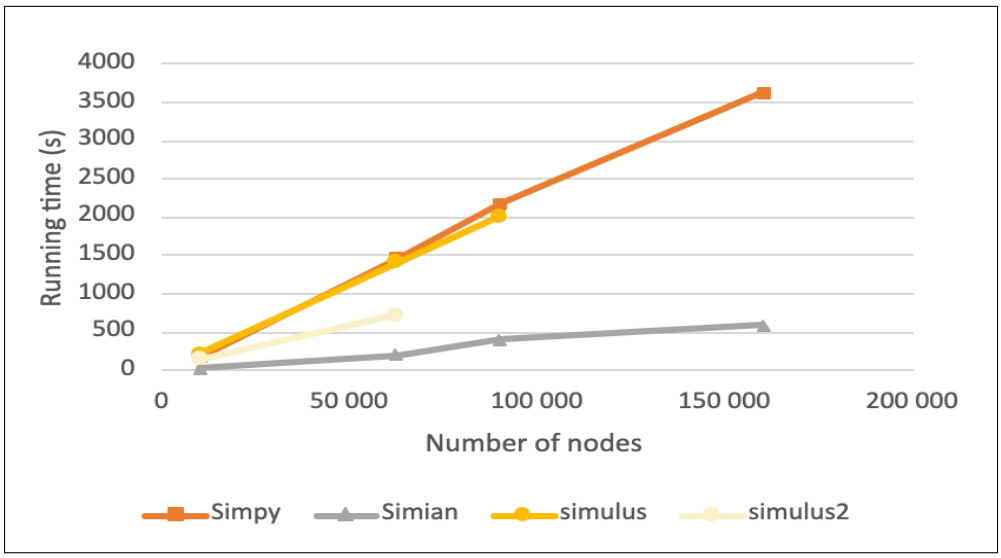
(b) Memory usage

Figure 4: Results for distance = 75

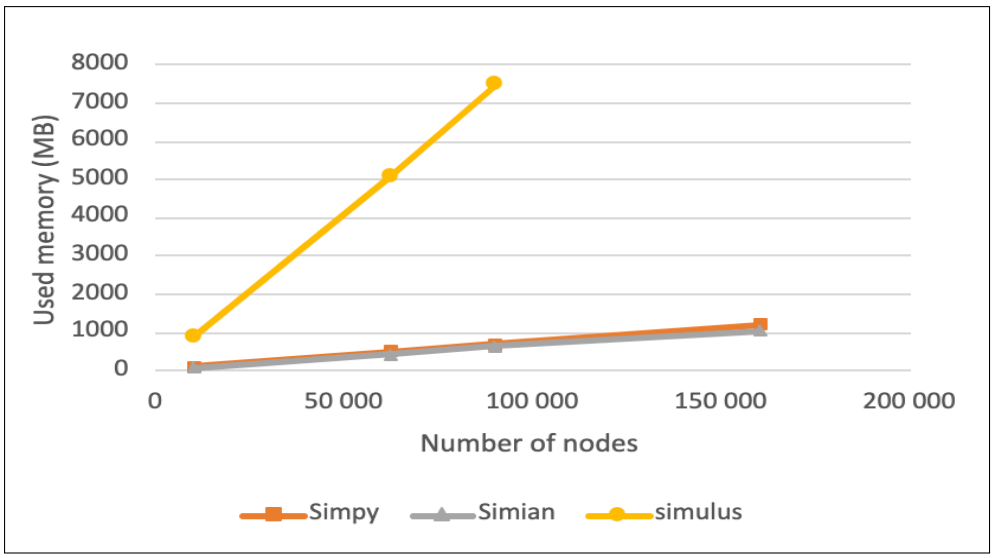
The results with decreased distance to select neighbors are depicted in figure 4. This will reduce the connection degree and the number of messages exchanged in the system.

As expected, the execution time and memory usage decreased, and Simulus was able to perform the simulation with 160 000 nodes. Simian simulation with a 160 000 nodes overlay changed from 6 to more than 3 minutes, and the memory used from 750MB to 570MB. As before, Simian was the fastest

simulation, followed by Simulus2, and then SimPy and Simulus. SimPy was faster than Simulus, but still too slow compared to Simian. Memory usage maintains the same order.



(a) Execution time when increasing distance

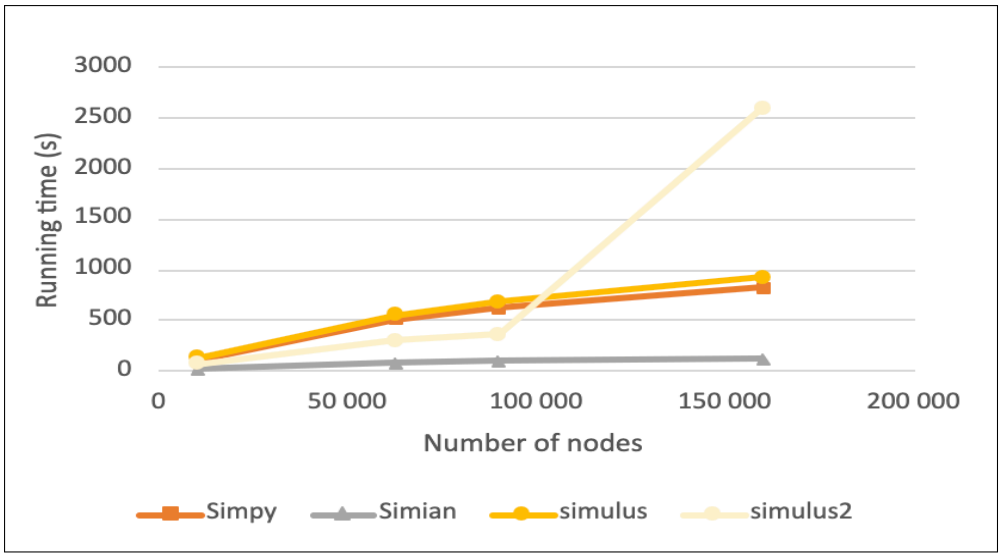


(b) Memory usage

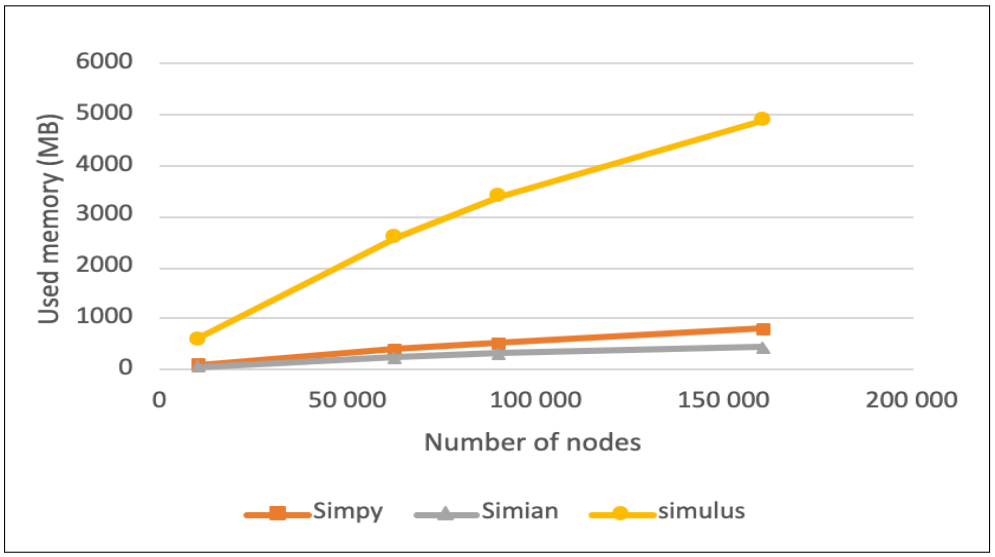
Figure 5: Results for distance = 125

Then, we increased the distance used to select neighbors, and the results are shown in figure 5. Contrary to the last experiment, the connection degree and the number of exchanged messages increased.

As we can observe, execution times and memory usage increased if compared to the initial experiment, and Simulus couldn't perform simulations with more than 90 000 nodes. Simian was the fastest simulation and used almost the same memory as SimPy.



(a) Execution time when increasing sending delay

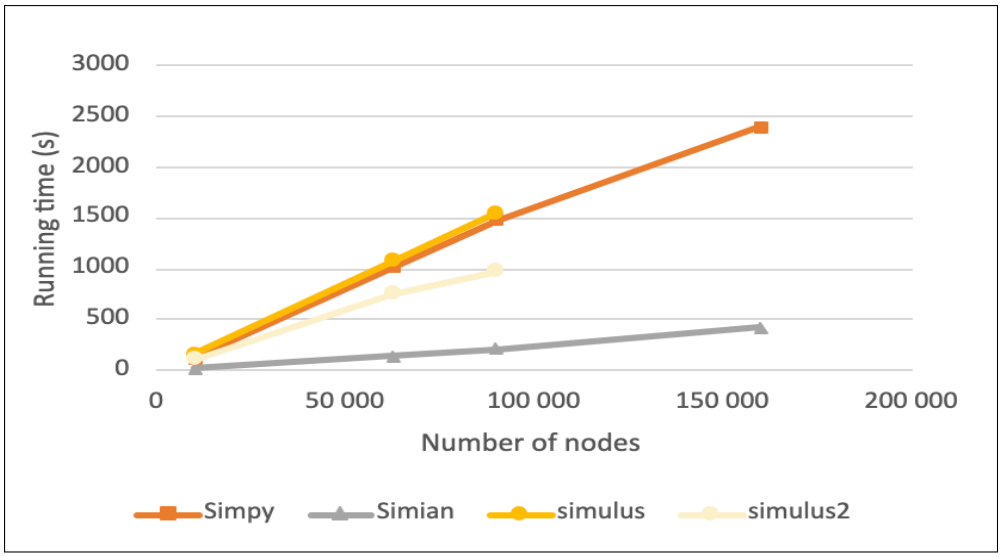


(b) Memory usage

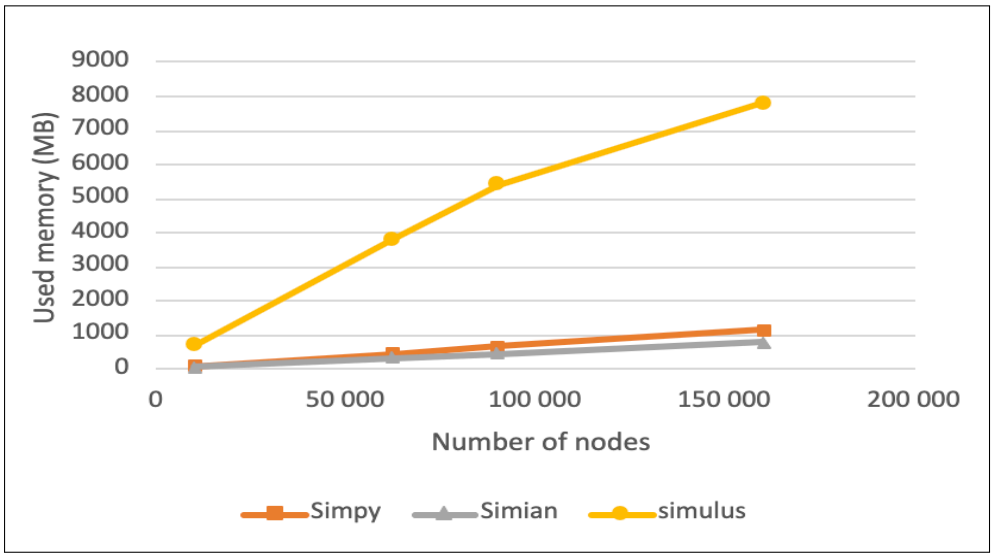
Figure 6: Results for lookahead = 0.3s

In this experiment, results are in figure 6. We increased the *lookahead* value corresponding to the message-sending delay from 0.1s to 0.3s. This will slow the message spreading, and it's expected that the execution time and memory usage will decrease.

As the number of exchanged messages decreased, Simulus could perform simulations with 160 000 nodes. Simian was still the fastest simulator and the one that used less memory.



(a) Execution time when using a different seed



(b) Memory usage

Figure 7: Results for seed = 123

In the last experiment on this machine, depicted in figure 7, we changed the *seed* used to generate the node coordinates so we could analyze the impact of the overlay on the simulations.

From the obtained results, we can see small changes in execution times and memory, but the overall results are very similar to the initial results 3.

Cluster 114 e 119

After the first phase, where we compared simulation tools under different values for sending delay, connection degree, and overlay coordinates, we moved to the next stage.

From figure 3, we notice that the Simian MPI version took almost 3 times more than its sequential version. The Simulus version using MPI revealed similar to the Simulus multiprocessors version, and the MPI + multiprocessors version was a bit faster.

In this phase, we are going to focus on running MPI. We want to find if the processing power of the used machine is limiting the execution time of distributed versions. We will run simulations on 2 different machines, with a better processor.

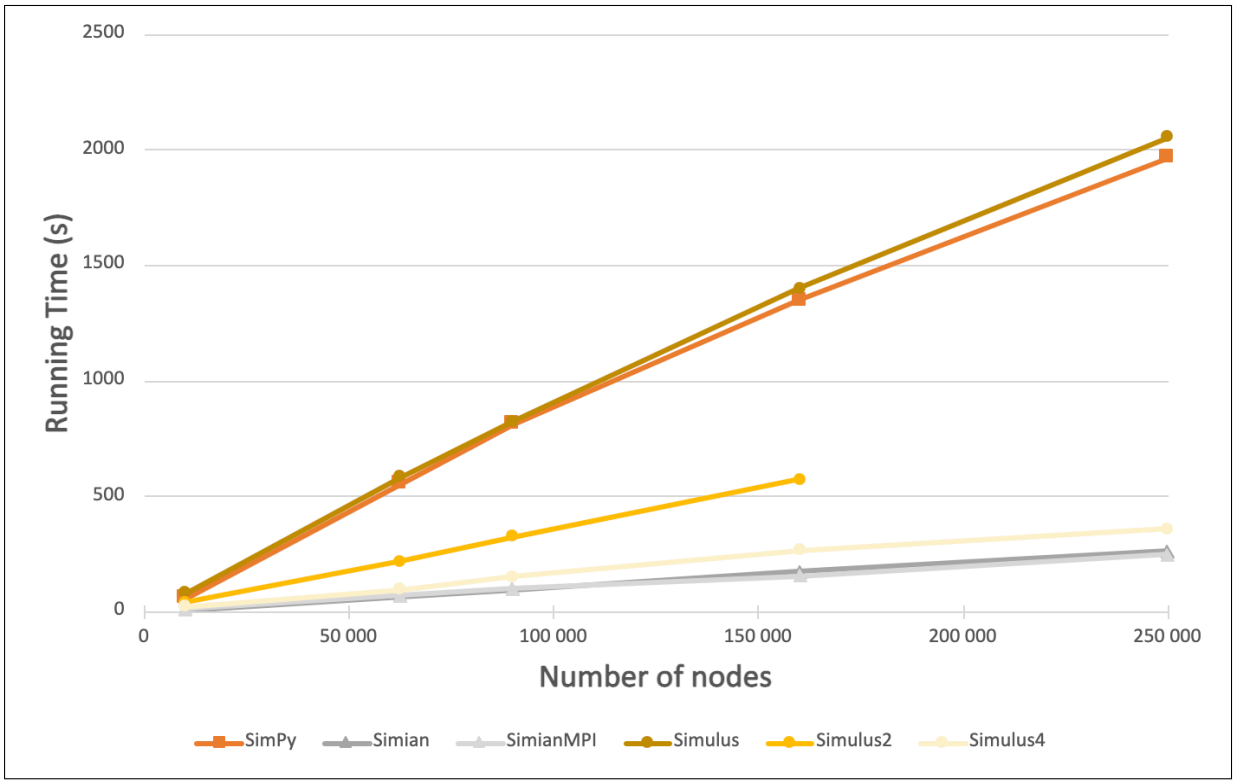


Figure 8: Execution time, sequential and multi-node versions

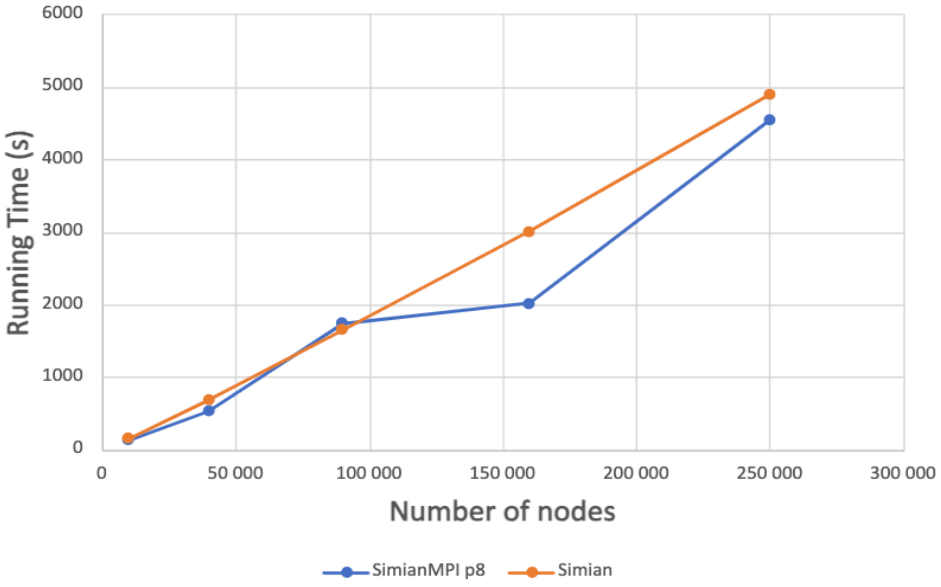
Figure 8 shows the execution times for sequential versions of the 3 chosen simulation tools and the distributed versions of Simian and Simulus. We run several simulations for the distributed versions using 2,4,8, and 16 processors. The 8 processors version was revealed to be the fastest on both Simian with MPI and Simulus4 with MPI + multiprocessors.

We can notice that all sequential execution times decreased significantly, and Simulus and SimPy could perform a 250 000 nodes simulation in less than 1 hour. Simian using MPI and Simian sequential versions reveal very similar times, with 249 and 263 seconds, respectively, for 250,000 nodes. Simian

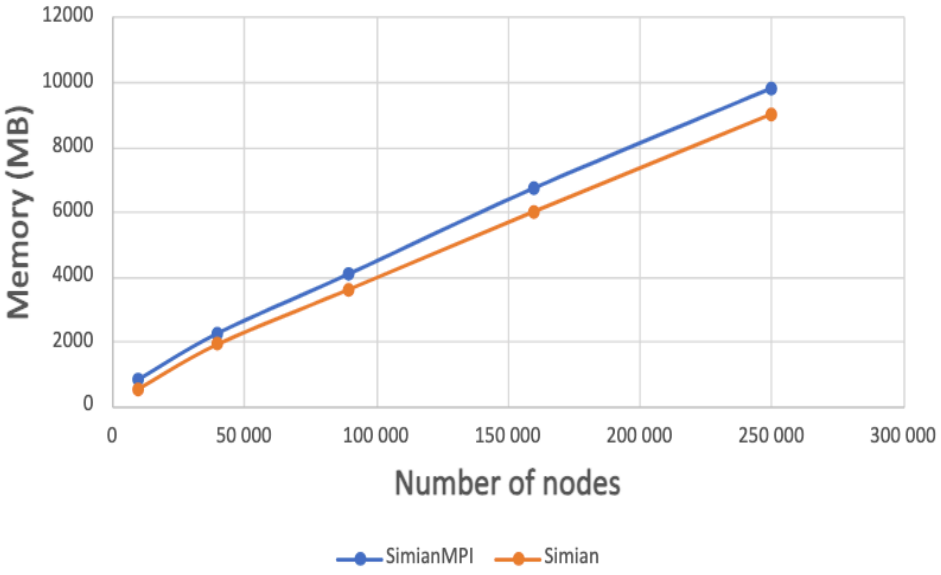
uses a conservative **PDES** strategy that, in the beginning, presented execution times 3 times slower than its sequential version, however when using more processing power it revealed competitive times. Simulus uses an optimistic approach to **PDES** and takes great advantage of **MPI** usage when increasing the processing power. Simulus using MPI + multiprocessors was really fast when compared to other Simulus versions and present execution times near to the Simian simulations, taking 359 seconds to perform a 250 000 nodes simulation. This Simulus version was the only one that could be compared to Simian versions despite, as we saw in the first phase, it consumes more memory.

Simian has been the best tool in terms of execution time and memory usage in all experiments, so we decided to choose Simian as the tool that better fits the simulations we want to perform.

To conclude, we decided to test Simian by increasing the exchanged messages to make a deeper comparison between its sequential and parallel versions. To do this, we ran the same experiments as above 8, but now a broadcast message is sent every second.



(a) Execution time when sending 1 msg/s



(b) Memory usage

Figure 9: Simian results: Sequential vs MPI versions

As shown in figure 9, both versions took similar execution times for under 100,000 nodes. Simian MPI was faster for 160 000 nodes, and 250 000 nodes took 75 minutes, faster than the 81 minutes of Simian.

In terms of memory usage, the Simian sequential version was a bit better, probably due to the extra work parallel execution has on managing communication between different processors.

Chapter 4

Simulation Model

Taking into account the benchmarking results shown in Chapter 3, all simulations will be conducted with Simian. We will present a generic model for epidemic protocols and we will detail its achitecture, components and implementation.

First of all, we need to define our model architecture. Then we will take a look at two peer sampling services (**HyParView** [18] and **Brahms** [19]) responsible for creating the overlay used by a chosen epidemic protocol for message dissemination, describing their implementation using Simian and all relevant decisions we made during this process. We will also mention all setting properties relevant to each protocol and how they could impact the experimental evaluation.

Afterwards, it's time to analyze dissemination protocols to combine with the peer sampling services mentioned above. We will use the PlumTree [16] gossip protocol. As this protocol has been implemented in the Simian simulation tool, we will also detail all key points and decisions we made during the implementation process and the relevant setting properties.

Finally, we will present and analyze some experimental tests. The main goal here is to evaluate our generic model for epidemic protocols, using two different peer sampling services (HyParView and Brahms) and a dissemination protocol (PlumTree), comparing their performances in terms of scalability, degree needed to achieve high reliability, performance in a stable environment, and in the presence of multiple failures.

4.1 Architecture

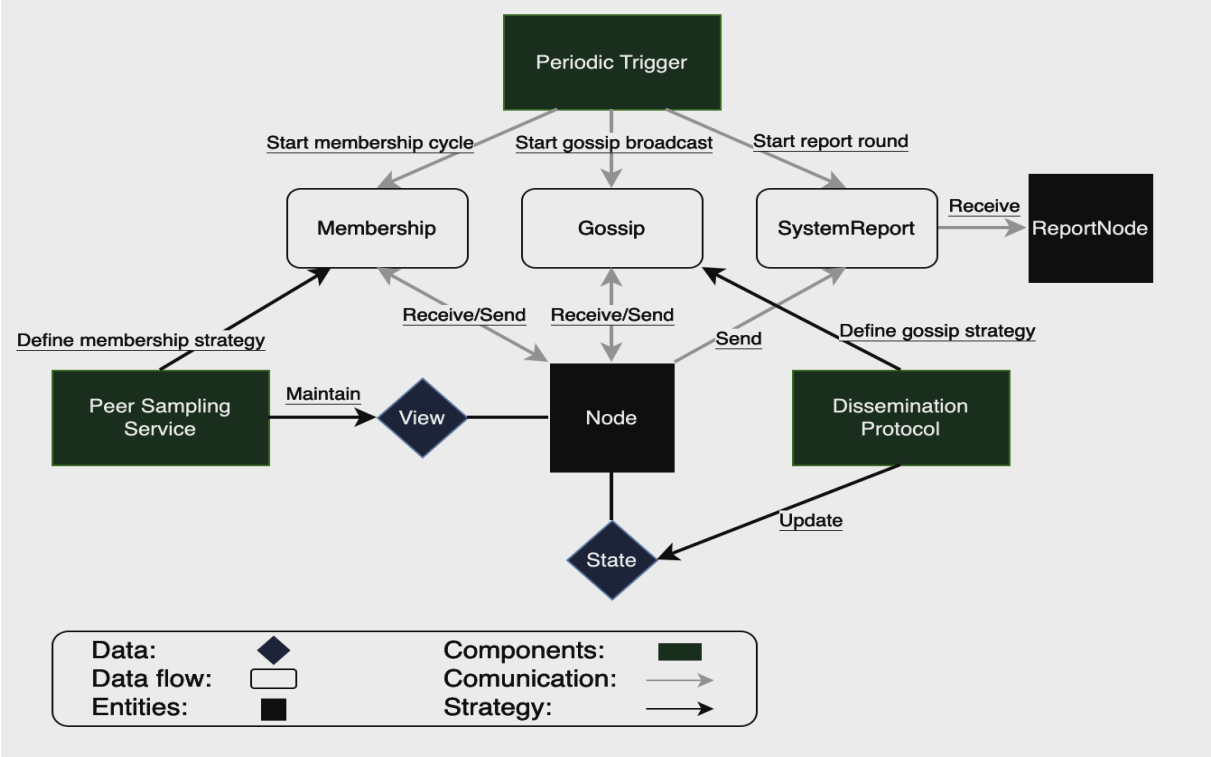


Figure 10: Model Architecture

Our model architecture compromises 3 main components:

- **Peer Sampling Service**: This component is responsible for creating and maintaining the overlay.
- **Dissemination Protocol**: This component is responsible for the dissemination strategy taking into account the views provided by the Peer Sampling Service.
- **Periodic Trigger**: This component compromises periodic triggers to execute membership cycles, required by the Peer Sampling Service, start broadcast messages or to evaluate the system state.

These components interact with 2 entities:

- **Node**: Stores data and communicate with other nodes (all communications and data updates follow the strategies of the 3 components mentioned above).
- **ReportNode**: Periodically receives system report messages from all active nodes and generates evaluation metrics values.

4.2 Implementation

The model implementation is divided into 3 sections. We start by describing the Peer Sampling services and respective membership cycles trigger implementation. This implementation compromises a description of the needed data structures and types of messages, peer sampling service strategy, and configuration variables. Therefore, the dissemination protocol implementation is described just by mentioning the small changes we made and some test settings. Finally, we explain how we measured simulations and calculated the evaluation metrics values.

4.2.1 Peer Sampling Service

HyParView

To implement the HyParView protocol using Simian, we started creating 4 data structures on the Node entity to store all information needed by HyParView.

- **ActiveView**: list of IDs that store active nodes
- **PassiveView**: list of IDs that store passive nodes
- **TimerTCP**: list of IDs that store nodes with pending **TCP** connections
- **NeighborQueue**: list of IDs that store nodes that rejected **TCP** connections

Then we defined the messages that will be exchanged by creating a class *msgHPV* with the fields *type*, *newNode*, *timeToLive* and *sender*.

The Message type could be one of the following:

- JOIN
- FORWARDJOIN
- DISCONNECT
- NEIGHBOR
- NEIGHBORREPLY
- TCPCONNECT

- TCPCONNECT_ACK

The function *HyParView* is responsible for implementing the desired behavior upon receiving each of the messages above.

When receiving a JOIN message, the node appends the message sender to its *activeView list* and, then, sends a FORWARDJOIN message to all neighbors on the *activeView*, using *reqService* to schedule events on the receiving nodes.

Upon receiving a FORWARDJOIN message, the node adds the new node to its *activeView* if the *timeToLive* field is 0 or its *activeView* is empty. Otherwise, the *passive view* is updated if *timeToLive* is equal to *PRWL*, and the message will be forwarded to a random neighbor, with the *timeToLive* being decremented.

A DISCONNECT message is used to remove a peer from the *activeView*, and the other four messages are used to simulate the use of **TCP**.

When a node detects a failure, it tries to send a TCPCONNECT message to a random node in its *passive view* and starts a timer to receive an *ack*. If TCPCONNECT_ACK is timely received, a NEIGHBOR message is sent, and the source node is added to the *neighborQueue*. Otherwise, a *TimerHPV* event will remove the peer from the *passive view* and try to establish another **TCP** connection. Finally, the node receives a NEIGHBORREPLY. If the request is accepted, it adds the new node to its *activeView*. Otherwise, it tries to establish a new **TCP** connection with a random node from its *passiveView* that isn't in the *neighborQueue list*.

To periodically update *passiveViews*, the event *TriggerPassiveViewMaintain* is responsible for starting the transmission of a SHUFFLE message with *ka* random elements from its *activeView* and *kp* random elements from its *passiveView*. Shuffle requests are propagated using random walks and have an associated "time to live".

The function *HyParViewShuffle* is responsible for implementing the desired behavior upon receiving SHUFFLE or SHUFFLEREPPLY messages.

Test settings

To configure **HyParView** to achieve the desired results, there are some settings to take into account:

- **Active view size**: $\log n + c$ (c is an input value)
- **Passive view size**: $k * (\log n + c)$ (k is an input value)
- **Active Random Walk Length (ARWL)**: specifies the maximum number of hops a *ForwardJoin* request is propagated. For 10,000 nodes, the recommended value is 6.

- **Passive Random Walk Length (PRWL)**: specifies at which point in the walk the node is inserted in the *passive view*. The ForwardJoin request carries a "time to live" field, initially set to *ARWL*, that decreases at every hop. *ARWL* value should be less than **PRWL**, and for 10,000 nodes, the recommended value is 3.
- **Ka**: number of active view elements sent in each Shuffle message
- **Kp**: number of passive view elements sent in each Shuffle message
- **Trigger time for shuffle messages**: Periodically, each node performs a shuffle operation with one of its peers, selected randomly, to update *passive views*.
- **TCP connections**: HyParView assumes that nodes use a reliable transport protocol to broadcast messages in the overlay. We need to set a timer to receive *TCP acks* and the delay time to send **TCP** connection requests.
- **Active view update**: The *active view* is managed using a **reactive strategy**. When a node suspects that one of the nodes present in its *active view* fails, it selects a random node from its *passive view* and attempts to establish a **TCP** connection. To implement this, all nodes reply with an *ACK* to all received gossip messages.

Brahms

Using the Simian simulation tool for Python, we started to implement Brahms protocol by creating four list variables:

- **V**: list of view nodes
- **S**: list of samplers
- **Vpush**: list of node IDs that send a push request
- **Vpull**: list of node IDs that send a pull reply

To define the messages that will be exchanged by Bramhs protocol, we create a *msgBrahms* class with three fields: (*type*,*view* and *sender*). Message type can take three values:

- PUSH
- PULL

- PULL_REPLY

Then, we started the implementation of Brahms's strategy described above.

The *Bramhs* method is responsible for receiving Brahms gossip, and *TriggerBrahmsSend* event is responsible for periodically (*TriggerBrahmsTime* value define the round duration) sending pull and push requests to random $\alpha\ell 1$ and $\beta\ell 1$ nodes, respectively. *TriggerBrahmsSend* is also responsible for updating *V* and *S* before the beginning of each membership round.

Upon receiving a PUSH request, the sender is added to *Vpush* list. When a PULL request is received, the node replies with a PULL_REPLY message containing its *V*.

Finally, when a node receives a PULL_REPLY, it appends the received nodes to its *Vpull* list.

The original design may contain duplicates on views and random subsets of a set. We decided to deviate from the protocol here as we want to increase the number of different peer IDs, and a duplicated node can be seen as a wasted slot of information. In this implementation, we aim to avoid duplicates in every list (*Vpush*, *Vpull* and V).

Test settings There are some parameters on *Brahms* [19] protocol that could impact the desired results:

- $\ell 1$: size of the dynamic views
- $\ell 2$: size of the **Sampler**, Brahms maintains a tuple of $\ell 2$ sampled elements in a vector of $\ell 2$ Sampler blocks. $\ell 1, \ell 2 = \sqrt[3]{n}$ suffices to protect even nodes that are attacked immediately upon joining the system.
- **Ping trigger**: sampled IDs are periodically probed (e.g., using pings), and a sampler that holds an inactive node is invalidated (re-initialized).
- **Brahms gossip trigger**: To update views, periodically, each node sends pull and push requests to random neighbors.
- α, β, γ: Brahms uses parameters $\alpha > 0, \beta > 0, \gamma > 0$ to satisfy $\alpha + \beta + \gamma = 1$ to control the portion of pushed IDs, pulled IDs, and history samples in the new view, respectively. $\gamma = 0.1$ is enough for protecting *V* from partitions.

4.2.2 Dissemination Protocol

PlumTree

PlumTree [16] aims to provide low overhead and support a large number of faults while maintaining reliability. For that, this protocol proposes a combination of two different approaches, an epidemic and a deterministic tree-based broadcast primitives, that allows the use of a low-cost scheme to build and maintain broadcast trees embedded on a gossip-based overlay.

Spanning tree structure depends on the stability of the partial views, which is the responsibility of the peer sampling service. When adding or removing a node from the partial view of a given node, it might produce changes in the links used for the spanning tree, which may not be desirable. The peer sampling service should employ a reactive strategy that maintains the same elements in partial views when operating in a steady state.

As the PlumTree implementation on Simian was tested and detailed in the previous chapter, we will only mention the changes made to support the two membership services mentioned above.

Implementation HyParView + PlumTree As HyParView [18] is based on the use of reliable transport protocol, we need to simulate **TCP** connections where we need to ensure the successful delivery of data and messages over the network. To detect node failures, each node replies with an ACK when receiving a GOSSIP or IHAVE message.

To implement the strategy mentioned, we add an empty dictionary *timersAck* upon the *Node* entity initialization. Afterward, we updated *LazyPush* and *EagerPush* events to, for each gossip message sent, add a new timer entry to the *timersAck* dictionary and start a timer to receive the corresponding ACK reply. The event responsible for this timer is *TimerAck*, and then an ACK isn't timely received, this method removes all timers for the receiver entry on *timersAck* and schedules a *NodeFailure* event.

The *HyParView* event, responsible for the strategy upon receiving each message, suffered some tiny changes. If a GOSSIP or IHAVE message is received, the receiver replies with an ACK message. When receiving an ACK message, the timer associated with that message on the *timersAck* dictionary is removed.

Implementation Brahms + PlumTree

Compared to *HyParView* [18], *Brahms* [19] deals with node failures differently. It periodically updates the view to isolate malicious and crashed nodes instead of relying on **TCP**.

For this reason, we do not need to make any changes to the previous *PlumTree* implementation to support *Brahms* protocol.

Test settings The Peer Sampling Service role used with this protocol is maintaining the overlay.

However, there are still some settings we need to take into account to achieve the best possible results:

- **Threshold**: when using Plumtree optimized version. The threshold value will affect the overall stability of the spanning tree.
- **Piggybacking policy for the IHAVE messages**: A scheduling policy is used to piggyback multiple IHAVE announcements in a single control message. The only requirement for the scheduling policy for IHAVE messages is that every IHAVE message is eventually scheduled for transmission.
- **Timeout**: When a node receives an IHAVE message, it marks the corresponding message as missing and starts a timer with a predefined timeout value. Therefore, it waits for the missing message to be received via eager push before the timer expires. *Timeout* value should be configured considering the diameter of the overlay and a target maximum recovery latency.
- **Timeout2**: When a GRAFT message is sent, another timer is started to ensure the message will be requested to another neighbor if it is not received. This second timeout value should be smaller than the first, in the order of an average round trip time to a neighbor.

4.2.3 Metrics

Simian doesn't have shared resources like stores, so we implemented the system evaluation by creating a new entity *Report Node*. Each node, before ending its execution, uses the Simian method *reqService* to send a report message to the reporting node. This report includes the out-degree and, for each received message ID, the number of IHAVE, GOSSIP, and GRAFT messages received. *Report Node* aggregates all information and calculates, for each broadcast message, the **reliability** (number of received messages / total nodes), **latency** (last delivery hop is given by the message with the highest round), **redundancy** (provided by the formula $\frac{m}{n-1} - 1$ where m is the number of payload messages and n is the number of nodes that received the message) and the number of messages received divided into types (HAVE, GOSSIP, and GRAFT). Ultimately, it presents the average of these values, considering the total of broadcast messages, and other relevant values (minimum and maximum out-degree values, average out-degree, and average shortest path).

4.3 Evaluation

We conducted simulations using *Simian* simulator [12] to implement *HyParView* and *Brahms* strategies and get comparable results. We compared the obtained results with published results for these systems

to validate our simulations.

Finally, we implemented the PlumTree gossip protocol, which will be combined with the two membership services mentioned above.

All simulations were conducted in three phases:

1. **Overlay creation**: *HyParView* nodes start joining the overlay one by one using the contact node 0. *Brahms* uses a different strategy, as it needs to receive an already connected overlay, all nodes start by having two neighbor nodes (*nodeID - 1* and *nodeID + 1*).

2. **Stabilization period**: Some cycles of membership protocol are executed on the first fifty simulated seconds to guarantee stabilization. *HyParView* uses this period mostly to fill passive views. Brahms updates all views several times to shuffle the basic starting overlay and fill *active views*.

3. **Broadcast**: The created overlay is tested by starting the transmission of 1000 broadcast messages.

The plumTree protocol creates trees optimized for a specific sender, which is the source of the first broadcast message. On multiple senders simulations, we decided to use a single shared plumTree where the *last delivered hop* and latency values may be sub-optimal for all senders except the one that created the tree. For this reason, we also conducted simulations using a single sender (this one should be more efficient and with a low relative message redundancy).

4.3.1 Experimental settings

Almost all experiments were conducted in a 10,000 node overlay, except scalability tests, where we also used overlays with 25,000, 50,000, 75,000 and 100,000 nodes.

HyParView was configured with the following settings: active membership set to *log n + c* (we present some tests using *c* between -1 and 2), passive membership size set to *Active view * 6*. *Active Random Walk Length* parameter was set to 6, and the *Passive Random Walk Length* was set to 3. In each shuffle message, *kp = 4* nodes were sent from the *passive view*, and *ka = 3* nodes were sent from the *active view*. The shuffle message length was set to 8, as nodes also send their identifier in each shuffle message. Each node sends shuffle messages at a rate of *0.2 msg/s*.

Brahms was configured with $\ell 1$ = $\ell 2$ = *log n + c* (we present some tests using *c* between 2 and 5 to find the optimal version) and α, β = *0,5*. As Brahms was designed to be resilient in the presence of byzantine attacks, it updates its views periodically, so we set this update time value to 1s.

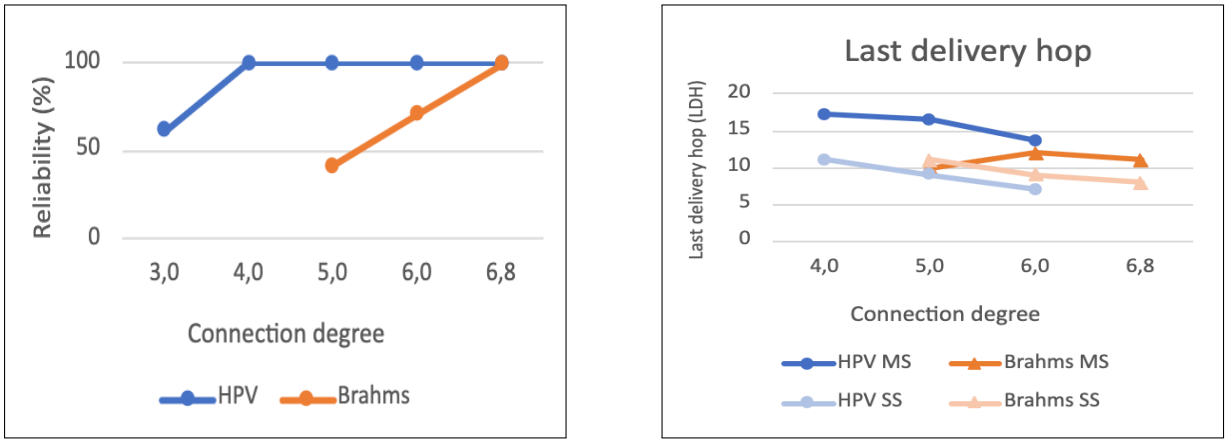
We do not use the plumTree optimized version. This version could improve **LDH** when new nodes are added, it could be interesting if we measured the **LDH** upon fail induction, but we used reliability as the main metric for fail recovery. All IHAVE messages are sent immediately, and *timeout* and *timeout2* values are set to *lookahead* and *lookahead / 2*, respectively.

4.3.2 Graph properties

On this evaluation, we will take into account the graph properties mentioned in 2 (connectivity, degree distribution, and average Path Length).

We started this experimental evaluation by testing different view sizes on a 10,000 nodes overlay to find the optimal view size value for each protocol. This view size value should be able to produce a connected overlay and a low *average path length* so that broadcast messages don't take too much time to be delivered.

HyParView's symmetric *active view* ensures that almost all nodes in the overlay are known by the maximum amount of nodes possible, which is the *active view* length (5). This means that all nodes, with high probability, will receive each message the same amount of times and that there is little probability for any node not to receive a message at least once. For that reason, it's expected that HyParView can produce a connected overlay with a small active view size.



(a) Reliability

(b) Last delivery hop (LDH)

Figure 11: Reliability and **LDH** for 1000 messages on a 10,000 nodes overlay

Figure 11a shows reliability using different *fanout* values. These values are the same for single and multiple senders because they use the same overlay. In figure 11b, we notice a significant difference on **LDH** between single and multiple senders versions (MS refers to multiple senders and SS to single sender).

As we can see in figure 11a, *HyParView* can produce a connected overlay with small fanout values. Due to its symmetric views, almost all nodes in the overlay are known by the maximum amount of nodes possible, so the connection degree is nearly the fanout value. **HPV** can produce a connected overlay with an impressive connection degree value (4). This simulation took less than 21 minutes using multiple senders and a *last delivery hop* value of 17,2. Using a single sender, the execution time dropped from 21 to more than 13 minutes with an **LDH** of 11.

When the *active view* was incremented to 5, the simulation took *10* more minutes with multiple senders and *6* more with a single sender. The *last delivery hop* shifted to 16,5 and 9, respectively. This significant difference between single and multiple senders simulation was because PlumTree creates trees optimized for a specific sender. For this reason, when simulating a single sender broadcast, the number of redundant messages drops drastically to near 0, and the *last delivery hop* to the optimal value.

We decided to choose the fanout value *5* as the optimal value. Despite not being the fastest, it reveals a better value for the *average shortest path* and *last delivery hop*. It also uses a higher connection degree and is more resilient in the presence of failures.

Brahms membership protocol creates asymmetric views, causing the overlay connection degree to be lower than the *fanout* value. Simulations with *fanouts* 7 and 8 generated overlays with partitions. Their connection degrees were 5.2 and 6,4, respectively, and the *last delivery hop* was 9 for both.

We then incremented the *fanout* to 9, and Brahms finally created a connected overlay with a 6.8 connection degree. This simulation took almost 19 minutes with a single sender and less than 40 minutes with multiple senders. The *last delivery hop* value was 8 and 11, respectively.

We can notice that the difference between multiple and single sender simulations was smaller than on HyParView. This is because every time Brahms updates its views, the tree created by the plumTree gossip is destroyed, and the subsequent broadcast message will repair them but will exhibit high *relative message redundancy*, causing the overall **RMR** to take a value around 0.8. The *fanout* value *9* was chosen as the optimal value for the next simulation.

Defined the optimal *fanout* values, let's take a look at other relevant properties:

Properties	Average shortest path	Maximum hops to delivery	out-Degree
HPV	9.40	16.5	3 - 5
Brahms	6.22	11.0	4 - 9

Table 1: Multiple senders

We notice that HyParView produced the highest values for the *average shortest path* and maximum hops to delivery, which is no surprise. It maintains small active views, so the number of distinct routes existing across all nodes is limited.

The *last delivery hop* suffered a significant impact when using multiple senders. The first message delivered took *8* hops to be delivered, but, as nodes use a shared tree optimized to the node that starts the first message, broadcasts starting on other nodes have sub-optimal **LDH** values, causing the average **LDH** to go up to 16,5.

As Brahms periodically updates its views, the broadcast trees are affected. This should be the reason why the impact on the **LDH**, using a shared tree for multiple senders simulation, was minimized, as every time the tree is repaired **LDH** takes optimal values (8 or 9 when views are updated and 11 for the 1000 messages average **LDH**).

Another crucial property is the out-degree distribution. As *HyParview* uses symmetric *active views*, practically all nodes in the overlay are known by the maximum amount of nodes possible. This means that all nodes should receive each message the same amount of times. For this reason, *HyParView* has a distribution of out-degree across a smaller range of values (between 3 and 5) than *Brahms* (between 4 and 9).

Properties	Average shortest path	Maximum hops to delivery	out-Degree
HPV	6.53	9	3 - 5
Brahms	5.47	8	4 - 9

Table 2: Single sender

Now, looking at the single sender simulation properties, we can see that the out-degree distribution remains the same, as the generated overlay is the same for both protocols, although the big difference in the *average shortest path* and *maximum hops to delivery* values.

LDH now took optimal values (*9* for **HPV** and *8* for Brahms) because the tree built by Plumtree is optimized for a specific sender. The *average shortest path* value also dropped for the same reason, but the difference between **HPV** and Brahms is less significant. This is because **HPV** takes advantage of the tree built by Plumtree since the first message and reveals an overall relative message redundancy near 0.

On the other hand, *Brahms* can not use the most optimized path continuously. Every time views are updated, the tree is affected and needs to be repaired.

4.3.3 Scalability

As mentioned above, scalability is one of the main concerns in distributed applications. Looking at HyParView and Brahms protocols, we can note some main concerns that could impact scalability.

HyParView's gossip strategy is based on using a reliable transport protocol, like **TCP**. To simulate the use of **TCP**, HyParView does some extra work, such as creating a new **TCP** connection every time a new node joins the *active view* and ensuring the successful delivery of data by sending acknowledgments.

On the other hand, Brahms updates all views periodically to isolate malicious nodes, which affects the number of overall membership messages exchanged and damages the PlumTree shared tree created. Every time a node updates its view, the broadcast tree is affected and needs to be repaired, causing the number of redundant messages to increase.

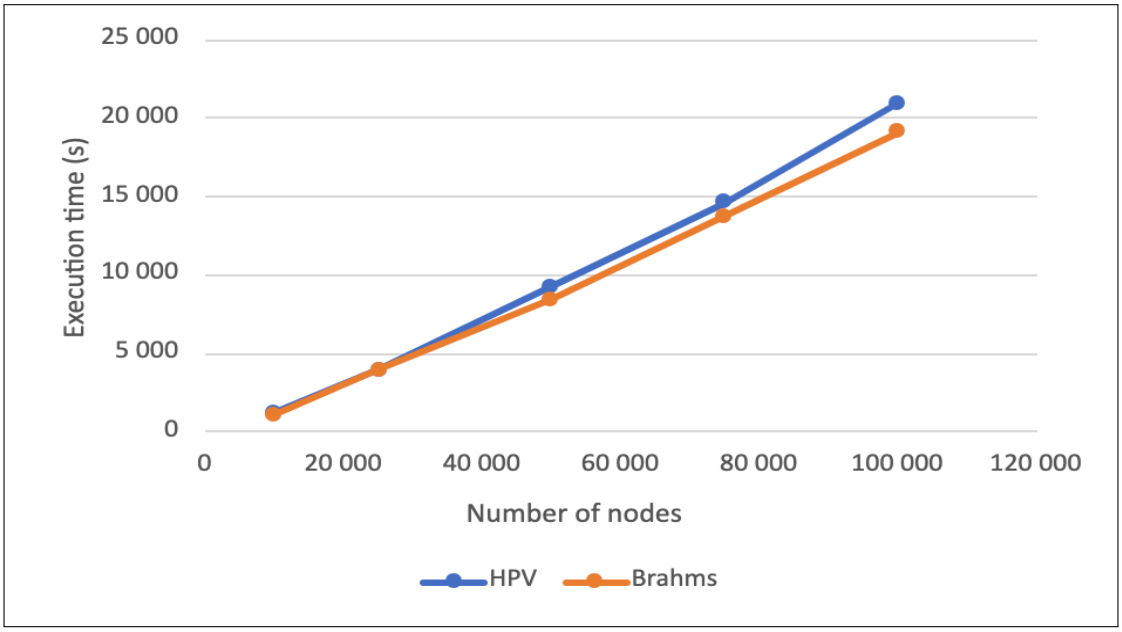


Figure 12: Single sender

From figure 12, we can notice that both protocols exhibit similar execution times under 25,000 nodes on single sender simulation with a small difference on the 50,000 and 75,000 nodes simulation. For the 100 000 nodes simulation, we can see some differences with Brahms taking 30 minutes less.

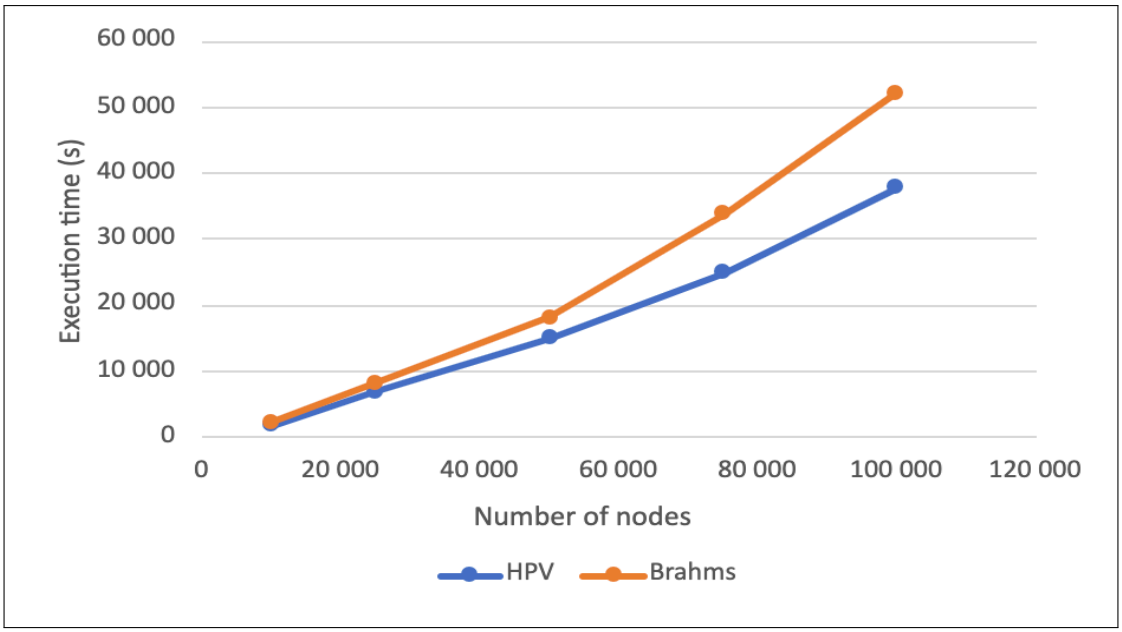


Figure 13: Multiple senders

The multiple sender simulations, figure 13, produced considerable differences. *Brahms* revealed slower executions and the difference between both protocols grew when increasing the overlay size. For 50,000 nodes, **HPV** took around 4 hours and 12 minutes, and Brahms took 5 hours and 5 minutes. This contrast becomes more significant when increasing the overlay to 100,000 nodes with Brahms taking 14,5 hours long, 4 hours more than the HyParView simulation.

To explain the obtained results, let's look at the average *relative message redundancy* exhibited by 1000 messages.

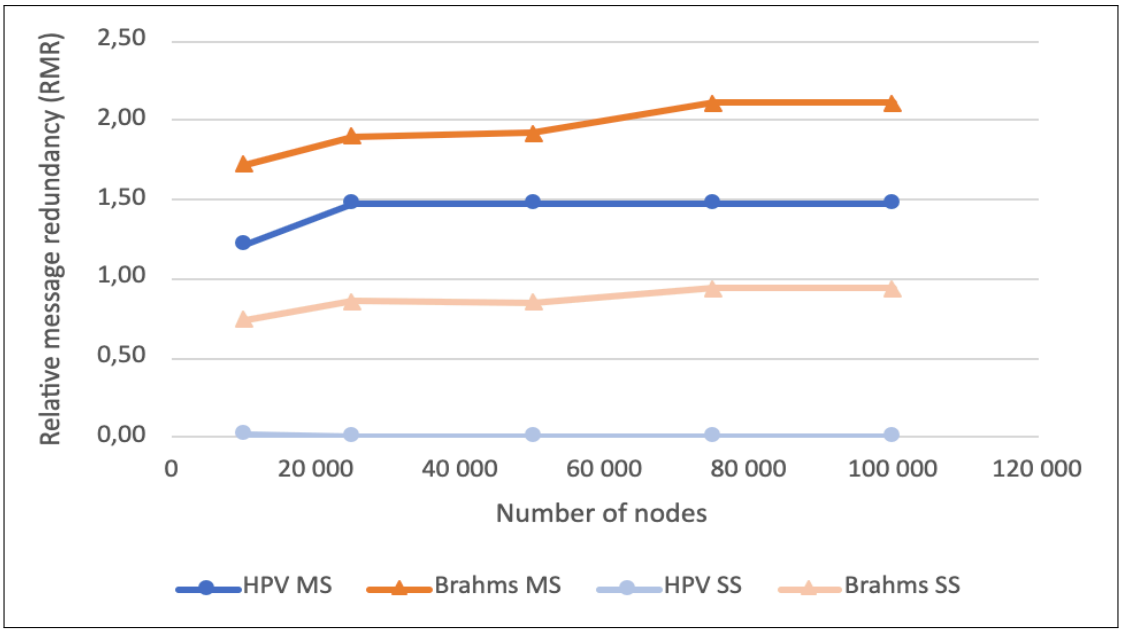


Figure 14: Relative message redundancy (RMR)

HyParView single sender simulation exhibits a **RMR** near zero, and the number of IHAVE messages exchanged is *4* times more than the number of nodes. *Brahms* produced a **RMR** of around 0.8, and each node exchanged approximately 6 IHAVE messages per broadcast message. *Brahms* simulations exchange more gossip messages for two reasons. A higher connection degree means more exchanged messages and damage on the shared tree. Every time views are updated, **RMR** goes to nearly 4.8 until it's repaired again, causing the average **RMR** to be about 0.8 and not near zero, as happens with **HPV**. We can also notice that, because for 100,000 nodes we needed to increase the fanout value to get a connected overlay, the **RMR** is higher. Although *Brahms* exchanges more gossip and membership messages, surprisingly, on single sender, the execution times are similar because **HPV** replies with an ack to all received messages, causing the number of exchanged messages to duplicate. On a more realistic system, **HPV** should produce faster simulations as gossip messages have a higher payload (IHAVE and ACK messages should have less impact on the execution time), but, on these simulations, all messages have the same length.

On multiple senders simulations, we can notice a significant difference between both protocols. *HyParView* revealed a **RMR** of around 1.5, and each node sends an average of a bit more than 3 IHAVE messages per broadcast. Brahms maintains a higher **RMR** with 1.9 and the same average of 6 IHAVE messages per broadcast for each node. For 75,000 and 100,000 nodes, because the fanout is increased, these values change to 2.1 and 6.5, respectively. *Brahms* revealed high *relative message redundancy* due to the periodic view updates that damage the shared tree. These were the expected results, as nodes use the same shared tree for each broadcast, **LDH** and **RMR** take sub-optimal values leading to slower

simulations.

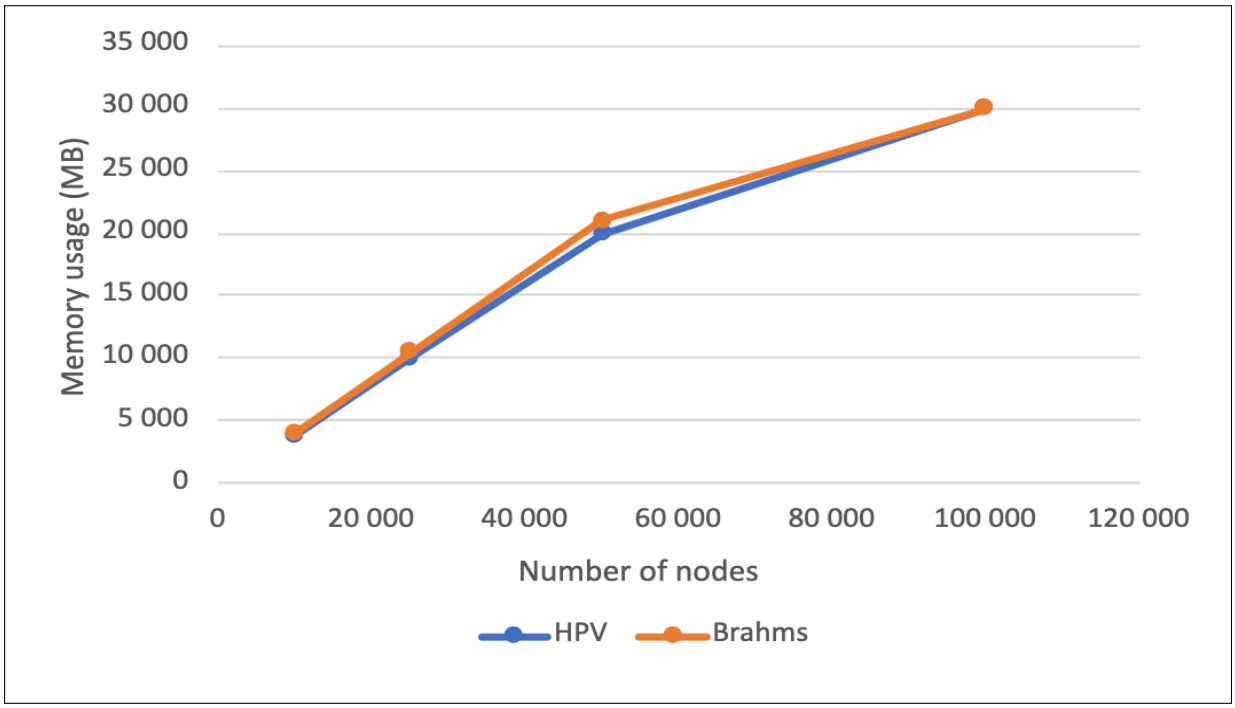


Figure 15: Memory usage - Multiple senders

Figure 15 shows the memory usage of the multiple sender simulations. *Brahms* uses a bit more memory, although this contrast is mitigated by the 100,000 nodes simulation where both use 30GB memory, which seems to be the simulator limit.

Overall, *HyParView* should be able to produce faster simulations. Almost half of the messages exchanged by **HPV** are acknowledgments inducted by **TCP**. We believe the difference in the execution times would be more significant if we could minimize the impact of ACKS and IHAVE messages on the execution, as these are small messages without any payload and should take less time to process.

4.3.4 Failure recovery

To make a good evaluation of how both protocols recover from node failures, we need to take a look at their recovery strategies. *HyParView* uses a reactive strategy. If a node sends a gossip message and doesn't receive an ACK message reply, it drops the receiver node from its view and selects a new one. On the other way, *Brahms* uses a cyclic strategy. Periodically, all nodes update their views considering all PUSH and PULL messages received. For this reason, using HyParView, a node needs to attempt to send a message to an inactive node to detect a failure. However, in Brahms, all nodes update their view periodically, isolating inactive nodes.

To simulate failures on the overlay, we select random nodes to fail, and then we initiate sending broadcast messages starting on active nodes.

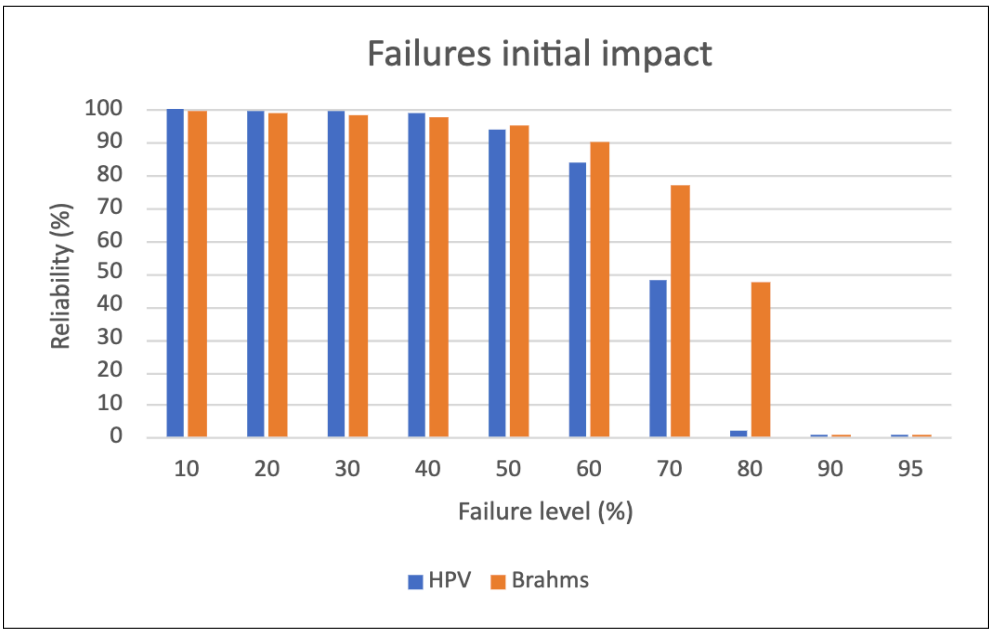


Figure 16: Reliability right after induction of failures

From the figure 16, we can see the initial impact the induction of failures has in both overlays. This is the reliability of the first broadcast message, before starting any recovery strategy.

For a percentage of failures up to 60%, both protocols can maintain more than 80% reliability. Then, when massive failures are induced, it starts to drop drastically with a more significant impact on *HyParView*, because it has a lower connection degree (view size is 5), than on *Brahms* (average view size is 6,8). On an 80% failure level, **HPV** becomes totally disconnected, but *Brahms* maintains around 50% reliability. For higher failure levels both become completely disconnected.

Afterwards, 1000 broadcast messages were transmitted by random nodes and distributed on 150 simulation cycles. As mentioned above, the two peer sample services use different recovery strategies. *HyParView* uses each message to test **TCP** connections, detect failures, and recover. On the other hand, Brahms will update *views* on each second (150 times for 150 simulated cycles), ignoring inactive nodes.

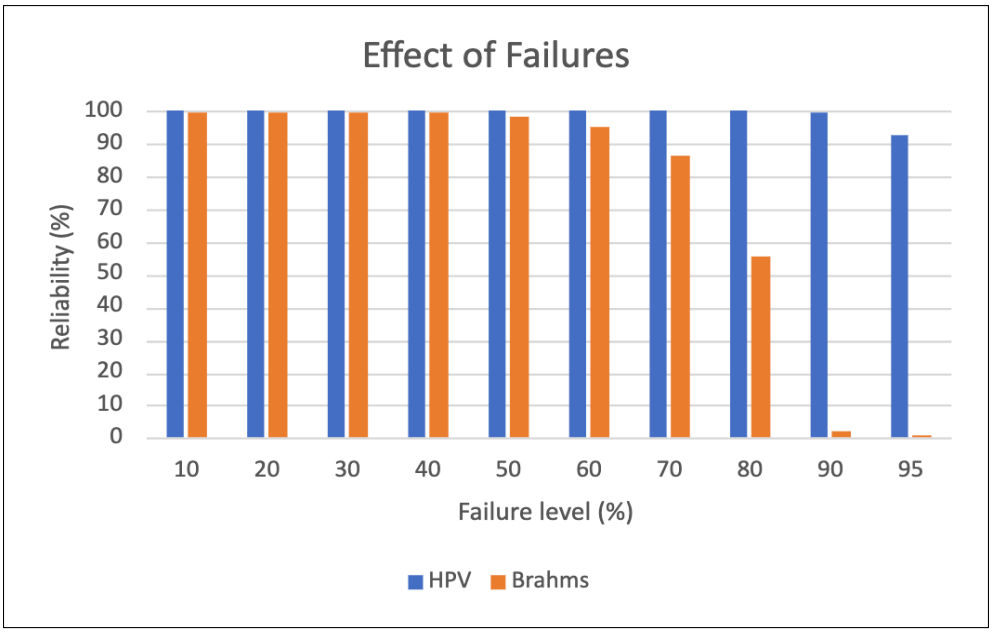


Figure 17: Reliability 1000 messages

Figure 17 shows the reliability achieved when sending 1,000 messages on experiments with several values, ranging from 10% to 95%, of node failures.

We can see that until 70% of failures level, *Brahms* reveals good reliability values (over 85%). At 80%, it drops to around 55% and for higher failure levels the overlay remains totally disconnected. *Brahms* can't recover more than 9% (at level 70 recovers from 77% to 86% and at level 80 from 47% to 56%) and, then, drops drastically until at 90% the reliability maintains around 1%.

In contrast, *HyParView* offers better and faster recovery usually near the 100%. It was able to recover from an overlay completely disconnected to 100% reliability, showing high reliability values even when massive failures were inducted (93% when 95% of failures).

This behavior highlights the importance of fast failure detection in gossip protocols. *HyParView* recovers almost immediately from the failures because all members of the *active views* are tested in each broadcast. We can also notice higher recoveries when using *HyParView* for two reasons. Large *passive views* maintained by each peer let them quickly switch a detected inactive node on the *active view* to another on the *passive view*. **HPV** also uses symmetric views, which means that if a node can reach another correct node in the overlay, it is necessarily reachable by messages sent by other nodes.

Brahms membership protocol does not use a failure detector like **TCP** and couldn't recover until the membership protocol is executed again. As it uses asymmetric views, some nodes may have outgoing links and no incoming connections. To maintain reliability under a massive percentage of failures, *Brahms* would have to be configured with higher *fanouts* (which is a cost-inefficient strategy in steady state) so the

initial impact on reliability could be mitigated.

Chapter 5

Conclusions

Epidemic protocols are highly important in the field of distributed computing and networking. They are used in **P2P** systems compromising thousands of peers, so studies need to rely on simulation. **Discrete-event simulation** (**DES**) plays a crucial and highly relevant role in these studies. We present a comprehensive comparison and analysis of **Discrete-event simulation** (**DES**) tools commonly employed in evaluating distributed protocols. This comparative study provided valuable insights into the strengths and weaknesses of these tools and let us select Simian as the most suitable **DES** platform for studying epidemic protocols.

Subsequently, we developed a generic model for epidemic protocols using the Simian tool to implement this model. Two peer sampling services were implemented (HyParView and Brahms) and one dissemination protocol (PlumTree). The integration of these protocols within the Simian environment offered a unique opportunity to evaluate our model in terms of performance, scalability, created graph properties, and membership and recovery strategies. Through meticulous analysis, we acquired deeper insights into how these protocols work together and identified potential areas for optimization and refinement.

We expect this work to contribute to the growing body of knowledge in epidemic protocol research and provide valuable guidance for researchers, developers, and network architects seeking to enhance the robustness and efficiency of distributed systems. We hope the findings and methodologies presented in this article could inspire further innovation and collaboration in the field, and lead to more reliable and resilient distributed systems.

The exploration of epidemic protocol simulation models has opened doors to several avenues for future research and development in the distributed computing field.

Additionally, future research can delve deeper into optimizing the performance of epidemic protocols. Already available and new peer sampling services and dissemination protocols should be added to the simulation model, leading to more robust, efficient, and resilient systems in this field. We also hope for more evaluation with bigger networks, adding the possibility of testing different types of overlay topology, and optimization of MPI simulations.

Bibliography

[1] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899–2917, June 2014.

[2] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12, 1987.

[3] Norm Matloff. Introduction to discrete-event simulation and the simpy language. *Davis, CA. Dept of Computer Science. University of California at Davis. Retrieved on August*, 2(2009):1–33, 2008.

[4] Martin Quinson, Cristian Rosa, and Christophe Thiery. Parallel Simulation of Peer-to-Peer Systems. In *CCGrid 2012 – The 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID '12 Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pages 668–675, Ottawa, Canada, May 2011. IEEE.

[5] Alberto Montresor and Márk Jelasity. Peersim: A scalable p2p simulator. In *2009 IEEE Ninth International Conference on Peer-to-Peer Computing*, pages 99–100. IEEE, 2009.

[6] Wojciech Galuba, Karl Aberer, Zoran Despotovic, and Wolfgang Kellerer. Protopeer: a p2p toolkit bridging the gap between simulation and live deployement. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, pages 1–9, 2009.

[7] Alberto Aguilar-Gonzalez, Camilo Lozoya, Carlos Ventura-Molina, Rodolfo Castelló, and Armando Román-Flores. Bit-siem: A packet-level simulation and emulation platform for bittorrent. *Journal of applied research and technology*, 15(6):513–523, 2017.

[8] Iftekharul Mobin, Sifat Momen, and Nabeel Mohammed. A packet level simulation study of adhoc net-

work with network simulator-2 (ns-2). In *2016 3rd International Conference on Electrical Engineering and Information Communication Technology (ICEEICT)*, pages 1–6. IEEE, 2016.

[9] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. The mpi message passing interface standard. In *Programming Environments for Massively Parallel Distributed Systems: Working Conference of the IFIP WG 10.3, April 25–29, 1994*, pages 213–218. Springer, 1994.

[10] Team SimPy. Simpy. https://simpy.readthedocs.io/en/latest/index.html, 2020. Accessed: 2023-01-02.

[11] Jason Liu. Simulus. https://github.com/liuxfiu/simulus, 2019.

[12] Pujyam. Simian. https://pujyam.github.io/simian/, 2022. Accessed: 2023-01-02.

[13] SimGrid Team. Simgrid. https://simgrid.org/doc/latest/Introduction.html#main-concepts, 2022. Accessed: 2023-01-02.

[14] Ayalvadi J Ganesh, Anne-Marie Kermarrec, and Laurent Massoulié. Scamp: Peer-to-peer lightweight membership service for large-scale group communication. In *Networked Group Communication: Third International COST264 Workshop, NGC 2001 London, UK, November 7–9, 2001 Proceedings 3*, pages 44–55. Springer, 2001.

[15] Meng-Jang Lin and Keith Marzullo. Directional gossip: Gossip in a wide area network. In *Dependable Computing—EDCC-3: Third European Dependable Computing Conference Prague, Czech Republic, September 15–17, 1999 Proceedings 3*, pages 364–379. Springer, 1999.

[16] Joao Leitao. *Gossip-based broadcast protocols*. PhD thesis, Master's thesis, University of Lisbon, 2007.

[17] Márk Jelasity, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten Van Steen. The peer sampling service: Experimental evaluation of unstructured gossip-based implementations. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 79–98. Springer, 2004.

[18] Joao Leitao, José Pereira, and Luis Rodrigues. Hyparview: A membership protocol for reliable gossip-based broadcast. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 419–429. IEEE, 2007.

[19] Edward Bortnikov, Maxim Gurevich, Idit Keidar, Gabriel Kliot, and Alexander Shraer. Brahms: Byzantine resilient random membership sampling. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, pages 145–154, 2008.

[20] Julius Bünger. Implementation and evaluation of brahms in the gnunet framework. 2015.

[21] Miguel Jorge Cardoso Branco. Godacen: Optimizing gossip for use in datacenters.

[22] François Taïani, Shen Lin, and Gordon S Blair. Gossipkit: A unified componentframework for gossip. *IEEE Transactions on Software Engineering*, 40(2):123–136, 2013.

[23] Gabriele D'Angelo and Stefano Ferretti. Lunes: Agent-based simulation of p2p systems. In *2011 International Conference on High Performance Computing Simulation*, pages 593–599, 2011.

[24] Luís Sobral. Epidemic simulation models. https://github.com/luissobral4/EpidemicSimulationModels, October 2023.

[25] Nandakishore Santhi, Stephan Eidenbenz, and Jason Liu. The simian concept: Parallel discrete event simulation with interpreted languages and just-in-time compilation. In *2015 Winter Simulation Conference (WSC)*, pages 3013–3024, 2015.

[26] Thomas P. Robitaille. psrecord. https://github.com/astrofrog/psrecord, 2013.