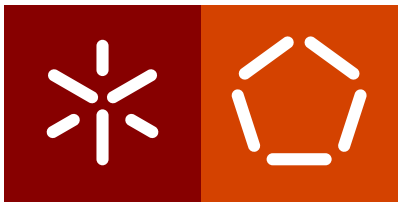
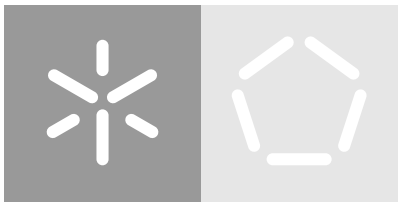


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Diogo André Teles Fernandes

Optimization of Epidemic Multicast Protocols

May 2021



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Diogo André Teles Fernandes

Optimization of Epidemic Multicast Protocols

Dissertação de Mestrado
Mestrado Integrado em Engenharia Informática

Dissertação supervisionada por
José Orlando Pereira
Ana Nunes Alonso

May 2021

AGRADECIMENTOS

Queria inicialmente agradecer ao meu orientador, José Orlando Pereira pela disponibilidade durante todo o processo de desenvolvimento desta dissertação, estando sempre pronto a reunir para esclarecer qualquer tipo de dúvidas, dar o seu conhecimento a nível tecnológico com excelentes sugestões para resolver os problemas e ajudar a atingir todos os objetivos propostos nesta dissertação, mesmo com as dificuldades impostas pela pandemia.

Em segundo lugar agradecer à minha coorientadora, Ana Nunes Alonso, que ajudou a tornar dissertação melhor e mais completa ao colocar à disposição a sua experiência e conhecimento.

Gostaria também de agradecer aos meus colegas de curso pelos momentos passados durante estes cinco anos de formação que tornaram o processo de formação muito mais agradável. Em especial agradecer aos meus colegas Catarina Carneiro, Daniel Silva e Tiago Lameira pela disponibilidade para trocar ideias, ajuda e motivação dada em vários momentos do desenvolvimento desta dissertação.

Finalmente quero agradecer à minha família, mas especialmente à minha mãe Maria Meneses por estar sempre presente, ajudar-me a crescer durante todo o processo de formação e dar o melhor suporte e educação durante estes anos.

DECLARAÇÃO DE INTEGRIDADE

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.

Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

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.

ABSTRACT

Epidemic multicast protocols, also known as gossip protocols, offer fault tolerance and good performance at large scale. Therefore, these are used in peer-to-peer (P2P) systems on the Internet and in NoSQL data management systems. Research has shown there are multiple variants of these protocols which are most efficient in certain environments and applications. Some protocols, such as Plumtree, even allow the application to configure to obtain different performance trade-offs. This dissertation aims at taking advantage of Machine Learning (ML) to configure these protocols, developing a solution that adapts in runtime to network conditions and evaluate it experimentally. The results obtained by using ML models to control the transmission strategy used when forwarding messages show that it is possible to achieve a better trade-off between bandwidth used and the time to reach the entire network. Moreover, this does not endanger the characteristics of epidemic multicast protocols, maintaining their reliability while becoming even more scalable.

Keywords: Epidemic Multicast, Machine Learning, Peer-to-peer

RESUMO

Os protocolos de difusão epidémica, também conhecidos como *gossiping*, oferecem tolerância a faltas e bom desempenho em grande escala. São por isso usados, por exemplo, em sistemas entre-pares (P2P) na Internet e em sistemas de gestão de dados NoSQL. A investigação feita mostrou que existem múltiplas variantes destes protocolos, adaptadas a diferentes ambientes e aplicações. Alguns protocolos concretos, como o Plumtree, permitem até que a aplicação faça uma configuração das suas características, de forma a obter diferentes compromissos de desempenho. Nesta dissertação apresenta-se uma abordagem que tira partido de tecnologias de aprendizagem automática para fazer a configuração destes protocolos, desenvolvendo uma solução capaz de se adaptar em *runtime* tendo em conta o estado atual da rede e posteriormente é feita uma avaliação da solução experimentalmente.
Os resultados obtidos com os modelos que controlam a estratégia de transmissão na distribuição de mensagens demonstram ser possível alcançar um melhor compromisso entre o número de mensagens enviadas e o tempo necessário para as distribuir. Além disso, não compromete as características dos protocolos de difusão epidémica, mantendo a sua confiabilidade e tornando-se ainda mais escaláveis.

Palavras-chave: Aprendizagem Automática, Difusão Epidémica, Entre-pares

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

A

AI Artificial Intelligence.

ANN Artificial Neural Network.

C

CNN Convolutional Neural Networks.

D

DBM Deep Boltzman Machine.

DBN Deep Belief Networks.

G

GBM Gradient Boosting Machine.

M

ML Machine Learning.

MOJO Model Object, Optimized.

P

P2P Peer-to-peer.

R

RL Reinforcement Learning.

LIST OF LISTINGS

1

INTRODUCTION

Epidemic multicast protocols, also known as gossip protocols, offer fault tolerance and good performance at large scale. They are used in peer-to-peer systems on the Internet and in NoSQL data management systems. These are also used as building blocks to help solve other distributed systems problems because they guarantee a message sent to members of an ad-hoc network has a very high probability of reaching every node. Gossip protocols are highly scalable and resilient because these distribute the load among all nodes and introduce redundancy that masks link or node failures. Peer-to-peer systems, where these protocols are frequently used, normally scale to thousands of nodes and cope with these frequently entering and leaving the system.

Experimenting and evaluating any variant of the epidemic multicast protocols can be hard and requires a simulator capable of supporting these properties and dynamism, in addition to allowing metrics to be collected and efficient memory usage.

Machine learning is an application of Artificial Intelligence (AI) that provides the ability to automatically learn and improve from experience without being explicitly coded. This aspect of AI focuses on the development of programs capable of accessing data and learning for themselves (16). At the same time, ML has been rising in popularity due to the increased volume and variety of data available, cheaper and more powerful computational processing and affordable data storage. It is now possible to quickly and automatically produce models that can analyze more complex data, with faster and most accurate results delivered (15).

1.1 PROBLEM

There are many variants of gossip protocols, which perform differently based on the environment, since these are normally customized for a certain situation or organization's particular needs.

P2P systems are dynamic due to churn, and the amount of data each node receives fluctuates during the normal execution, impacting links availability. Because of these factors, using rigid structures such as spanning trees is not efficient, due to the cost of the structure

reconstruction on failure. However, gossip protocols introduces an overhead in the number of messages sent to assure reliability.

For those reasons the optimal **transmission strategy**, to get the best trade-off of distribution time, number of messages sent and reliability depends on network conditions and application characteristics.

1.2 OBJECTIVES

The main objective is to know if it is possible to optimize the protocol performance through configuration using ML. We propose to use an inference model for the nodes decisions making the configuration automatic for a given environment under different approaches. To achieve that goal we need to: **implement the gossip protocol** in a P2P simulator, **select metrics** that impact positively the model's inference capability, collect metrics under several environments to gather the necessary data for the ML inference model and evaluate their impact on performance metrics.

1.3 DOCUMENT STRUCTURE

This document is structured as follows. Chapter 2 provides an overview of gossip protocols and artificial intelligence. Chapter 3 describes the implementation of a custom simulator including the gossip protocol, data gathering and the training process. In Chapter 4, the standard execution of the protocol is compared to the ones where nodes make decisions using ML models. Finally, Chapter 5 discusses the results and future work.

2

STATE OF THE ART

This chapter summarizes the state of the art in Gossip Protocols and Machine Learning.

2.1 GOSSIP PROTOCOL

Epidemic multicast protocols, also known as gossiping, are communication protocols that randomly forward information between peers. The term epidemic arises from the way the protocol spreads information, being similar to the way that a virus spreads. These protocols are known to offer high fault tolerance and good performance on a large scale. There are multiple variants adapted to different environments and applications.

There are three different types of protocols depending on participant number and roles:

- **Unicast** - Represents a one to one transmission with a sender and a receiver.
- **Multicast** - Transmission of data to multiple recipients simultaneously, using the most efficient strategy. Uses a group containing some nodes in the network.
- **Broadcast** - Transmission of data to all recipients present on the network, simultaneously.

In **epidemic multicast**, all nodes receive each message at least once with high probability. To achieve this, nodes send messages to only a part of their neighbourhood, randomly chosen. As in this type of diffusion, messages are sent in a uniformly random way, connections are not used in the most efficient way and many redundant payloads are sent, leading to a poor trade-off between latency and bandwidth. In addition, as all nodes send and receive approximately the same number of messages, there is no benefit obtained from nodes and links with greater capacity. However, load balancing is an essential characteristic of the protocol to achieve resilience and scalability.

In contrast, **structured multicast protocols** use spanning trees to take advantage of more efficient paths and consequently make better use of resources. Structure is created according to an efficiency criterion and used to transmit several messages. This way, the notion of super-nodes arises, defined as nodes with greater capacity and which consequently contribute more to transmissions. Selected nodes in the leaves of the generated tree will

not contribute to the dissemination. This implies a greater overhead in the reconstruction of the trees, when failures or reconfigurations occur in the network, which makes these protocols less useful in systems with a high number of nodes and in constant change, as is characteristic of P2P networks (22).

The gossip protocol is then based in two distinct primitives:

- **Eager Push** - Sends the payload to a neighbour without knowing if the receiver already has that message.
- **Lazy Push** - Sends a notification to a neighbour of knowing a given message and only sends the payload upon request.

From these two primitives come two general approaches: the **eager push gossip protocol** that only uses the `Eager Push` primitive and consequently uses a lot of connection bandwidth, but minimizes latency, and the **lazy push gossip protocol**, which uses the `Lazy Push` primitive, allowing content to be transmitted only once to each destination, at the cost of an additional round. It is possible to combine the two approaches, in a single round of gossip, obtaining different trade-offs between latency and bandwidth, depending on which messages are transmitted with `Eager Push`.

`Plumtree` (35) first proposed a solution to reduce the number of payloads sent while maintaining reliability by optimizing the number of messages passed and the distance to travel. The solution uses an eager gossip approach, which sends payloads to all neighbours, but whenever a duplicate message is received the node responds to that source to not send any following messages. This is done to avoid in the following rounds to send payloads via duplicated paths and reduce the total number of payloads sent on the network, still ensuring all nodes receive the message.

Taking into account these two versions of multicast a message, it is possible to explore knowledge of the environment to create an "emergent structure" to improve the dissemination of data, bringing performance closer to that attained with the **Structured Multicast** approach (30).

Structure arises from the implicit creation of a `Spanning Tree` when using `eager push` transmissions to create message distribution paths. The creation of this structure is done using a probabilistic method, since nodes and connections are selected with different probabilities to transmit the content of messages. As the structure depends on the nodes in which `eager push` is used, the structure emerges from the strategy used to schedule the sending of payloads, in a mixed usage of `eager` and `lazy push`.

The **transmission strategy** has two main objectives: to reduce the transmission of redundant information as much as possible and the latency of message delivery. This translates to

deliver messages to all nodes quicker but also avoid sending message payloads to nodes that already have that message.

- **Flat** - For probability π, use `eager push` and with probability 1 - π utilize `lazy push`, being π a variable between 0 and 1.
- **Time-To-Live (TTL)** - Uses `eager push` until round **n** from message origin which can be useful considering that in the first rounds the probability of a node being targeted by more than one copy of the message content is low and there is rarely any reason to use `lazy push`.
- **Radius** - Uses `eager push` if, for a defined metric, the value is less than a ρ variable, called radius. It uses the intuition to distribute the message payload with close neighbouring nodes. Creates an emergent structure that approximates to a mesh structure where the majority of the message content is transferred by links between neighbouring nodes.
- **Ranked** - Aims at a Hub-and-Spoke structure (12), by selecting some nodes as best that will serve as points in the communication between nodes in the network. Therefore, if one of the nodes related to the transfer of a message is a "central point", `eager push` will be used.

That said, the goal is to have a better trade-off between latency and bandwidth by selecting nodes and connections with larger capacity in a decentralized way.

2.2 MACHINE LEARNING

Machine Learning is an artificial intelligence application that provides the system with the ability to automatically learn and improve from experience without being explicitly programmed (27). In general the idea is, the greater the amount of data used for learning, the more accurate the predictions can be, assuming that the data used is well processed and no redundant or misleading data is given.

ML has basic paradigms related to their learning style: **supervived** which is task-oriented, **unsupervised** that is data-driven and **reinforcement learning** (RL) that learns from mistakes. This way of organizing ML algorithms is crucial to understand the role of the data to be used in the learning and in the process of preparing the model to identify the most appropriate style for the problem, to obtain the best result possible.

On the other hand, it is common to also group algorithms by similarity in the way they work. Examples of this are: regression, artificial neural networks and deep learning.

In the following sections these styles are explained in detail.

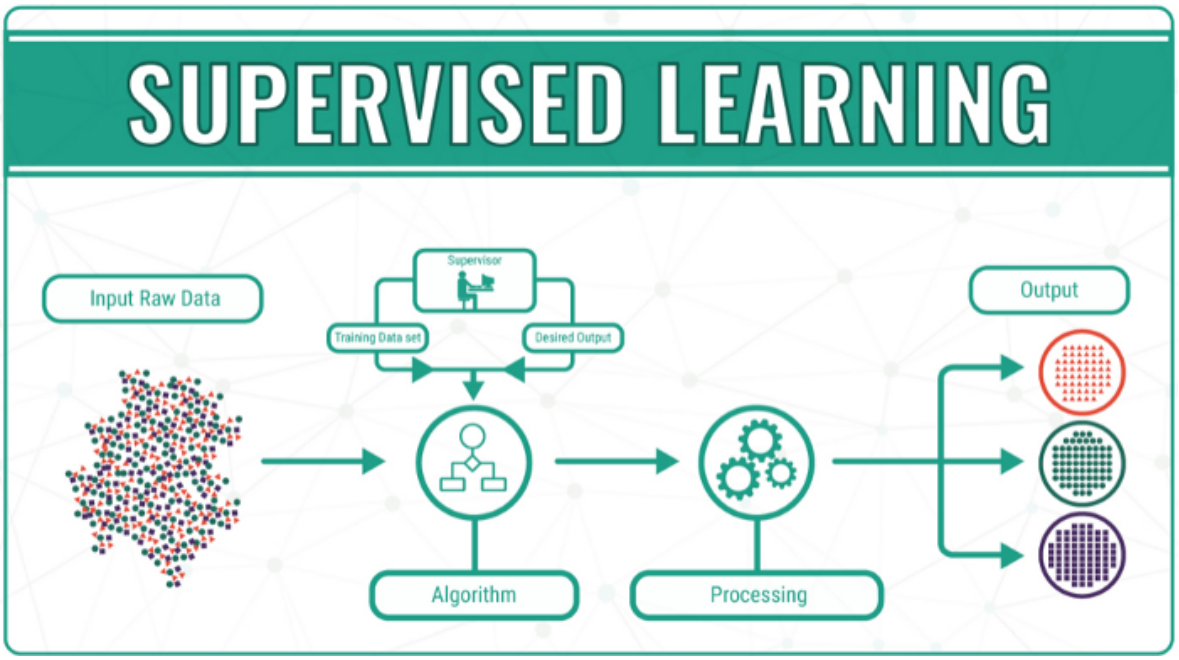


Figure 1: Example of the Supervised Learning paradigm (6).

2.2.1 *Supervised Learning*

In this paradigm the `input data` is called **training data** and there is a column with the **result** or action taken. This way, it requires **human action** to provide the desired input and output. That process is depicted in Figure 1 where raw input data with the human interaction from the supervisor, provides the desired input and output.

The algorithm measures its accuracy through the loss function, adjusting until the error has been sufficiently minimized meaning that the model reached a designated level of accuracy in the training data (13).

Artificial neural networks (ANN) are models inspired by the structure and functioning of biological neural networks. They represent simplified models of the human central nervous system (31). It is an extremely interconnected structure of computational units, often called neurons or nodes with learning capacity.

These networks are characterized by (34), (28), (32), (37) : **high parallelism** that comes from an extremely large number of simple processors; **distributed computation**; **generalization** and **learning** ability with **evidential response** being able to quantify the confidence on the decision made; **adaptivity** which means being able to change their topology and synaptic weights as the environment changes; **fault tolerance** since ANN's require extensive damage on the network before the overall response is degraded seriously due to the distributed nature of information stored in the network; **flexibility** and **transparency** since ANN's can

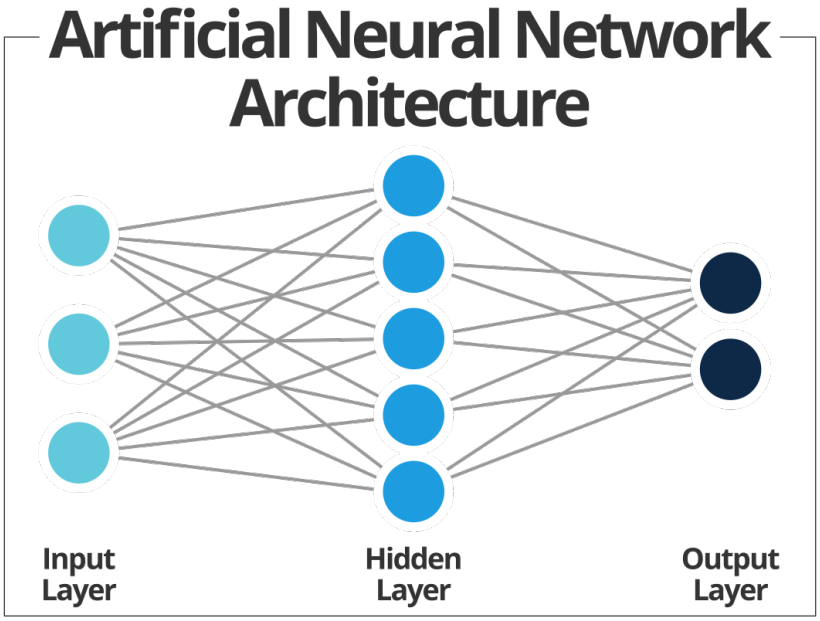


Figure 2: Architecture of an artificial neural network (20).

be seen as a "black box" transforming input vectors into output vectors via an unknown function.

Figure 2 depicts the Artificial Neural Network (ANN) architecture that consists of an input layer, an output layer and the middle layer that can have several parallel layers stacked that form a **deep neural network**. These neural networks are a class of pattern patching, often used in supervised machine learning problems such as classification and regression.

Deep learning methods are an update to ANNs that exploit abundant cheap computing. Compared to ANNs, these methods build much more complex and larger networks that tend to better simulate the way the human brain works. This way, networks with more than three layers are then referred to as **deep neural networks** but deep learning tends to have at least 6 or more layers. Instead of teaching the computer how to process and learn from the data, in deep learning the computer trains itself to do so. A deep learning system is self-teaching, learning as it goes by filtering information through multiple hidden layers, similarly to humans (8) (14) (25).

According to Andrew Ng, founder of Google Brain, it is the first class of algorithms in ML that is scalable. Comparing to older algorithms, the performance just keeps getting better as you feed them more data while the other algorithms stagnate their performance after a certain quantity of data used.

The most popular algorithms in deep learning are: Deep Boltzman Machine (DBM) (38), Deep Belief Networks (DBN) (33) and Convolutional Neural Networks (CNN) (1).

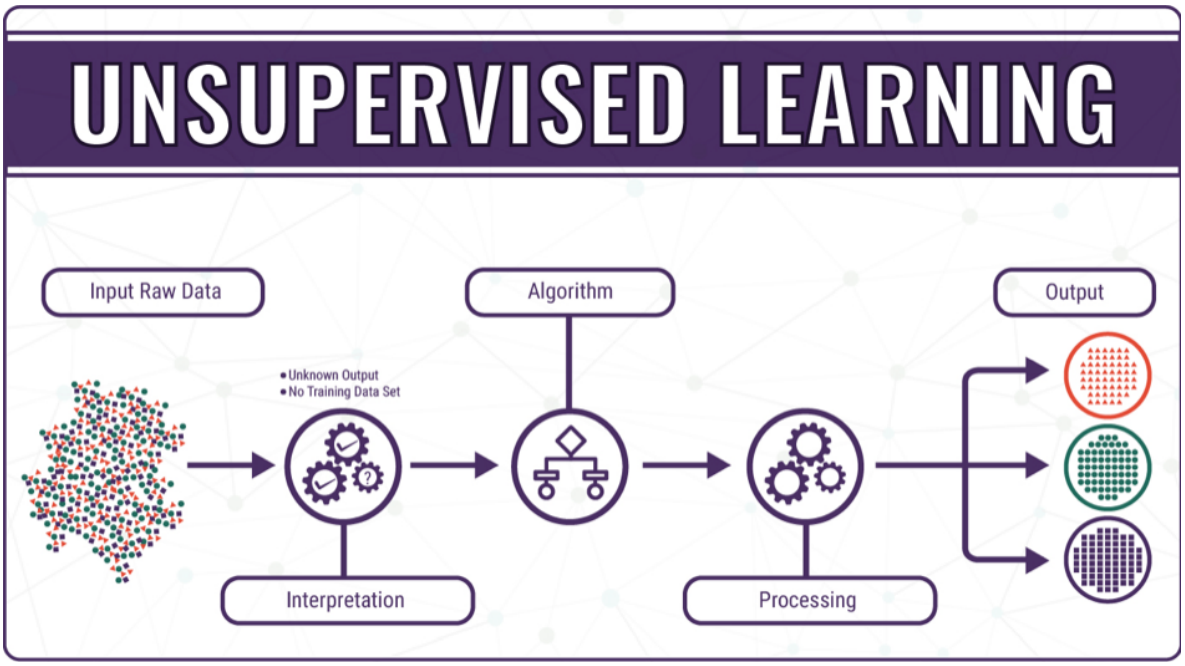


Figure 3: Example of the Unsupervised Learning paradigm (6).

2.2.2 *Unsupervised Learning*

Figure 3 depicts the structure of unsupervised learning where the input data is not categorized nor is the desired result known. The model is prepared by deducing structures present in the input data. This can be used in order to find general rules, reduce redundancies or even organize data by similarity.

Knowledge discovery problems are characterized by **segmentation** if looking for groups of data that share identical characteristics or **association** if searching for strong similarity relationships between the data.

2.2.3 *Reinforcement Learning*

This paradigm allows the machine and software agent to automatically determine the ideal behaviour within a given context in order to maximize performance. This performance is calculated using rewards and punishments as signs of positive and negative behaviours – simple reward feedback (3).

Such as Figure 4 depicts, in this paradigm the agent is exposed to an environment where it trains itself and continually experiments and makes mistakes. This machine learns from past experiences and tries to capture the best knowledge to make accurate decisions within an iterative process.

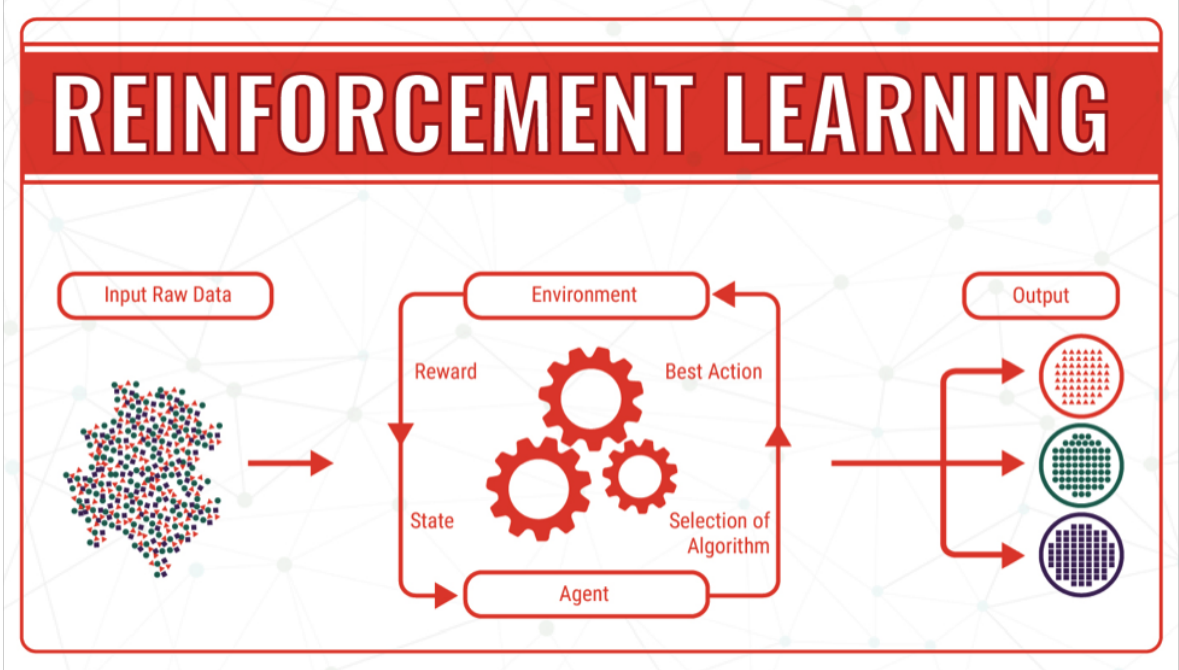


Figure 4: Example of the Reinforcement Learning paradigm (17).

Problem-solving in reinforcement learning balances *exploration* of new search spaces to obtain new knowledge with *exploitation* of knowledge already obtained by the system during the execution.

2.3 SUMMARY

With this information we can conclude that epidemic multicast has several transmission strategies and these variants have different performances in different environments. In addition, P2P systems are constantly changing and the best strategy may not necessarily be to follow any of the studied strategies, but rather to follow alternative algorithms that mix approaches, at any time taking into account the current knowledge of what is happening in the vicinity of the network. This creates the opportunity to use ML to create a model capable of making predictions by the nodes of the type of push to be used, taking into account the current knowledge of the neighbourhood. Since a prediction is desired where on each message transmission a choice has to be made between an `Eager` or `Lazy push` the supervised learning paradigm is the one used.

3

DEVELOPMENT

This chapter describes the design and implementation of an ML model to configure epidemic multicast protocols. Then, it introduces the simulator where the protocol is implemented, how it works and the options made for the implementation. Following that, it covers the metrics used for the construction of the dataset and how they are collected. Finally, it covers the necessary steps to build and test the model in the environment, the kind of model used and the metrics chosen for comparing and analysing runs.

3.1 GLOBAL VIEW

The main goal for this work is to evaluate if gossip protocols can be optimized using ML models. Figure 5 shows the iterative process used. This process runs over a P2P simulator where the protocol is implemented. To test a configuration, a script is run queuing a number of simulations, with a randomly generated topology **(1)**. At every run, a new row is printed for the summary file **(1.1)** and the history of their decisions and effectiveness appended to the dataset file **(2)**. This dataset can contain information of decisions taken during one or multiple runs by appending the content to a specific dataset that contains information of runs within the same configuration type. That data is then pre-processed and fed into the ML instance, `H2O.ai` **(3)**, to start the training process and generate a model **(4)**. This model is then integrated into the simulator **(5)** to make decisions for nodes. Following that, the new configuration file is used to queue new executions using the new model to obtain model performance results in the run summary file **(6.1)**. Finally, the results analysis is done by comparing the performance of each approach **(6.2)** and if they are not good enough or other strategies can be tested, the process from step 3 to 6 can be repeated using a different training process.

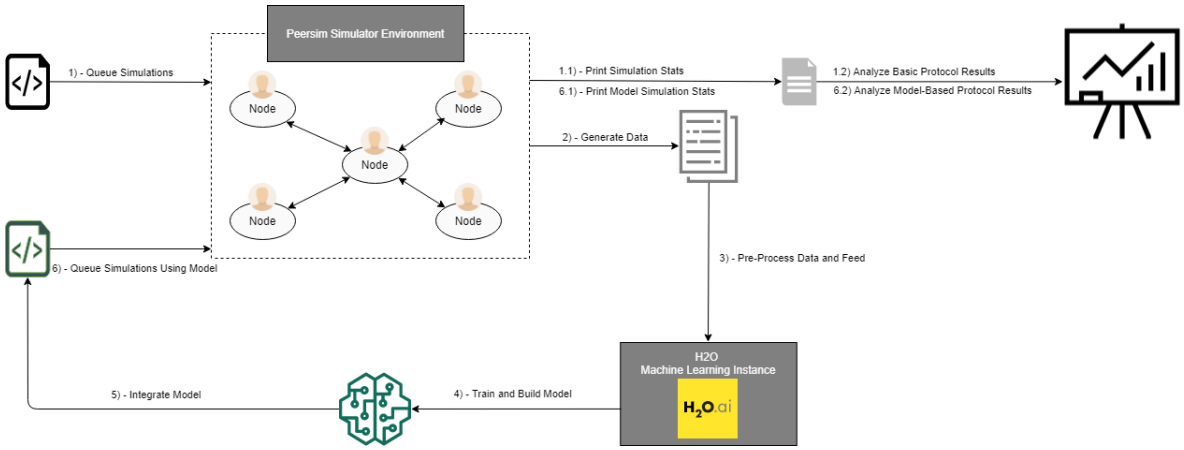


Figure 5: Overview of the process needed to obtain results.

3.2 GOSSIP SIMULATOR

The protocol is implemented in Java, using the component-based `Peersim` simulator (19). This makes it easy to prototype a protocol, combining different pluggable building blocks and adapt them to our needs. The following sections cover how the simulator works, the metrics that make up the summary and dataset of the simulations, and how they are obtained.

3.2.1 *Overview*

`Peersim` is a Peer-to-peer simulator that allows the user to only code two kinds of components, controllers and initializers, and customize them to a specific use case. In summary, it offers: Two simulations engines, a simplified **cycle-based** version where there is only a coarse-grained notion of time and an **event-driven** version which better simulates real world interactions; network topologies following a power law such as Watts-Strogatz model and Barabási–Albert preferential attachment (36) (29); simple message system that allows the transfer of `Java` objects between nodes; flexible configuration system; scalability and dynamism allowing complex networks with variable number of nodes.

Running a simulation on this framework requires at least the following parameters:

- **Network size** - Number of nodes in the network.
- **Node Protocol** - Protocol to run in the nodes assigned to the simulation. Java class where we have to implement how the protocol creates new messages and how he processes and distributes them.

- **Initializers** - Classes that only run before the start of the simulation to generate the network topology or initialize some variables of the protocol class.
- **Controllers** - Classes that monitor all the protocol properties we are interested in, such as the number of payloads sent the previous round, and allows accessing and updating some variables on the protocol class chosen.

Since in a cycle-based simulation there are some simplifications, lacking a transport layer and concurrency, the event-driven simulation is the version used. Sending a message in the **event-driven** simulation uses the transport protocol set in the configuration file. With that and a delay assigned to each message, there is a better notion of time and a more realist simulation since there is a logical time to give control to each node to take action and every node processes their messages at the moment they receive a new event. This transport layer is used when we need to send messages during the gossip protocol: whenever a message is created and when a new message is received meaning the node needs to publish and forward the message.

In an event-driven simulation it is necessary to implement two methods to cope with the interface implemented within `PeerSim`. First one is **nextCycle** and it is useful to make some activity within the simulation periodically. In this case, for the gossip protocol, it is useful to periodically make some nodes on the network create a new message and start to spread them to their neighbours. It is possible to define the frequency of this in the scheduler defined on the configuration file which in this case happens every round. The second method is the **processEvent** which is where it is implemented what to do with received messages, that are the events of the simulation.

In the next listing is shown a simple configuration to better understand the potential of the configuration system.

```
1  # PEERSIM EXAMPLE 1
2  random.seed 1234567890
3
4  # number of nodes
5  SIZE 200
6  network.size SIZE
7
8  # parameters of message transfer
9  # eg 50 means half the cycle length, 200 twice the cycle length, etc
10 MINDELAY 40
11 MAXDELAY 60
12
13 # defining number of cycles
14 CYCLES 400
15 CYCLE SIZE*10000
16 simulation.endtime CYCLE*CYCLES
17 simulation.logtime CYCLE
18
```

```
19 ################### protocols ===========================
20 protocol.urt UniformRandomTransport
21 protocol.urt.mindelay (CYCLE*MINDELAY)/100
22 protocol.urt.maxdelay (CYCLE*MAXDELAY)/100
23
24 protocol.lnk IdleProtocol
25 protocol.avg example.eagerpush.MixedGossip
26 protocol.avg.linkable lnk
27
28  ################### initialization ======================
29 init.rnd WireKOut
30 init.rnd.k 5
31 init.rnd.protocol lnk
32
33 init.peak example.eagerpush.GossipInitializer
34 init.peak.protocol avg
35 init.peak.value 20
36
37 include.init rnd peak sch
38
39 ################ control ==============================
40 control.avgo example.eagerpush.NewRoundQueuer
41 control.avgo.protocol avg
42 control.avgo.step CYCLE
43
44 include.control avgo
```

Listing 3.1: Simple simulation configuration.

The configuration in Listing 3.1 starts by assigning some global variables to the simulation such as the network size, delay of messages, number of cycles in the simulation and the corresponding logical time assigned to each cycle, from line 1 to 17. Then comes the definition of the protocols. In this case, from line 20 to 22 it assigns the delay values to the transport layer and following it the protocol class is chosen in line 25, `example.eagerpush.MixedGossip`, and associated with a protocol name **avg**. The next line associates the node protocol to the PeerSim class responsible for storing every link, giving the `MixedGossip` protocol access to another class that is a static link-structure and a source of neighbourhood information, allowing nodes to communicate with their neighbours.

Next, the configuration focuses on the initializers. The first initializer is responsible for the topology creation associated to the variable **rnd**, in line 29. The method chosen for this example is **WireKOut** in which every node will have **K** neighbours. In the next line we assign K a number and finally, associate the protocol with the linkable class created for nodes to access that information.

Following that, to initialize the structure of the protocol class `MixedGossip` and to set some configuration variables such as the chance to create a message each round it needs an initializer assigned. Therefore, the class `GossipInitializer` is set and obtains the chance of creating messages as the value field in line 35.

Line 37 specifies the selected initializers by the names assigned before to them. In this example it uses the topology generator **rnd** and the node initializer **GossipInitializer**. This allows different initializers configured for several purposes in the same configuration and switching between them by changing this line, using the **include.init** prefix.

Finally, a controller is set that accesses node information every round and queues new activity for the next round. So, in line 40 the controller chosen is **NewRoundQueuer** and is associated with the selected protocol class **MixedGossip** in the last line.

In summary, this configuration files creates an event-driven simulation with:

- 400 Cycles.
- 200 Nodes.
- Topology following **WireKOut** model, assigning every node with 5 neighbours
- Protocol initializer **GossipInitializer** with a 20% chance of creating a message.
- Simulation main protocol, **MixedGossip**, a gossip protocol implementation that chooses between eager and lazy transmissions randomly.
- Controller **NewRoundQueuer** that queues the creation of new messages every round.

3.2.2 *Message*

The message class is used to convey the necessary information between nodes. It contains variables which allow nodes to identify the sender and creator of the message, know about the timestamp and round of when it was created, the size of the payload and other statistics for decisions and dataset information. In detail it contains:

- **Id**: Numeric identifier of the message. Useful to identify if nodes already have that message published or requested.
- **Msg**: Content of the message. Textual representation of the message sent.
- **PayloadSize**: Size of the payload. It is important for the calculations of link capacities.
- **CreatorId**: Numeric identifier of the node that created this message.
- **SenderId**: Numeric identifier of the node that sent this message. Useful to notify the sender of the message about the effectiveness of his decision.
- **CreateTimestamp**: Simulation logical timestamp of the moment the message was originally created.

- **LastSendTimestamp**: Simulation logical timestamp of the moment this message was sent from. With the difference obtained between **CreateTimestamp** and **LastSendTimestamp** nodes can measure for how much time the message is already circulating being a useful metric to consider on a transmission decision.
- **Sender**: Node object of the sender of the message. The sender node object is stored to ease the access and do some processing.
- **RoundSent**: Round in which this message was sent. Can be another metric able to quantify the time that message is circulating in the system.
- **PayloadCounter**: Counter of the times this payload was received. Starts at 1 whenever each node receives that message the first time and gets incremented every time he receives a repeated payload.
- **NotifyCounter**: Counter of the times a notification for that message was received. Starts at 0 whenever he publishes the message and gets incremented every time a new notification of that message is received.
- **Hops**: Number of hops the message has taken from the message creator to that node. Incremented on each message reception by every node of the path taken between the message creator and that node.
- **Type**: Type of the message. Available types are `PAYLOAD`, `NOTIFICATION`, `REQUEST` and `FAILED`.

When a message is received with type `PAYLOAD` and the corresponding **Id** is not in the list of published messages, it means that message needs to be forwarded. For that, a new Message object is created with all variables but updating the fields SenderId, Sender, LastSendTimestamp, RoundSent and Hops. When the desired type is a `NOTIFICATION`, `REQUEST` or `FAILED` nodes create a lightweight version with only the variables needed to process that type of messages which are Id, SenderId, Sender, Round and Type.

3.2.3 *Node*

Figure 6 depicts the architecture of the implementation in the simulator where nodes use **MixedGossip** class, the epidemic multicast protocol. It is the most extensive component of the implementation because it defines how to spread messages to neighbours and how frequently and with which specifications each message is created. The class also has other data structures such as counters or mapping structures to make it possible to obtain statistics

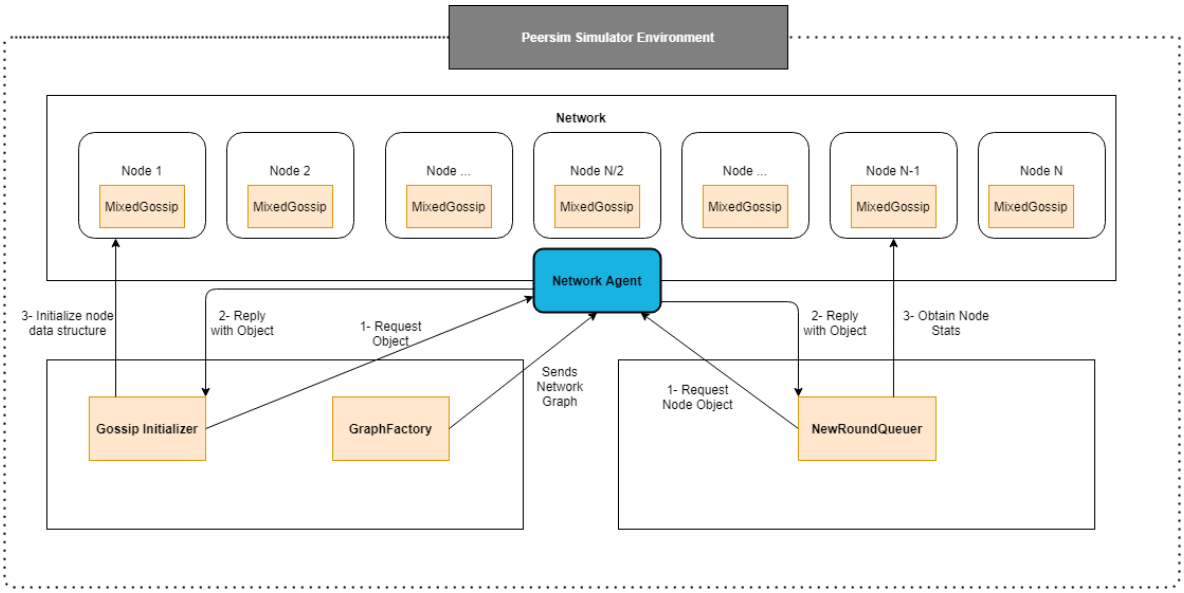


Figure 6: Architecture of the protocol implementation in Peersim.

and metrics to evaluate runs or to collect the dataset.

The basic behaviour of the protocol requires each node to store in their structure a list of their published messages, a neighbour list and a set of arrays that contain information to simulate the bandwidth limits following the leaky bucket algorithm: the links maximum capacities, last read capacities and rounds of last visits. On each round a set of nodes is selected to create a new message, based on a configured parameter **chance**. After the selected nodes create their messages, they start the distribution using the protocol transmission types, `eager` and `lazy push`. Whenever a payload emission is attempted, the leaky bucket parameters are used and updated to check if the payload can be sent as follows:

1. The `capacity`, `lastVisitCapacity`, `leaking` and `lastTimeVisited` arrays are read to calculate the link capacity on that precise moment. Being the round variable the number obtained from the current round in the simulator, the current capacity is calculated as:

$$\text{currentCapacity} = \min(\text{Capacity}, \text{lastVisitCapacity} + \text{leaking} * (\text{round} - \text{lastTimeVisited})) \quad (1)$$

2. The node then checks the size of the message in bytes, payloadSize and the condition **currentCapacity > payloadSize** is performed to check if the payload can be sent.
   a) If true, the payload can be sent. The node updates the corresponding `lastVisitCapacity` to **currentCapacity** − **payloadSize**, `lastTimeVisited` to the current round and the payload is then sent.

b) If false, it means the message is lost by overflow. `LastVisitCapacity` updates to the `currentCapacity` value from the previous calculation and the lastTimeVisited updates to the current round.

On each message payload reception, if the node does not already have this message, it has to spread that payload to the remaining neighbours. On a notification, if the node does not have that message yet, it has to request the payload from the sender. If it already had that message the node ignores that notification and sends a negative impact feedback. After a while, it is expected for every node to receive a copy of each message created. The simulation blocks the creation of new messages 10 rounds before the end of the simulation so every message created can be fully distributed.

The **isEagerSending** method is the most important one because it defines, under a certain Gossip Protocol algorithm, the circumstances in which some message i should be sent as a payload from that node to a neighbour j.

The **nextCycle** method creates a message with probability given by the `protocol.chance` parameter. The message is created with a limited payload size using a lower limit obtained from the config parameter `init.peak.messageLowerlimit` plus a random number between 0 and a bound defined in the parameter `init.peak.messageBound`. The size of the payload is calculated by:

$$PayloadSize = lowerLimit + random[0, messageBound] \tag{2}$$

After creating the message, the **isEagerSending** method is used for each node that belongs to the neighbourhood to determine if, for that node, the payload should be sent immediately. For observation purposes, statistics counters are incremented depending on the decision made, taken into account if the message was lost and the number of notifications sent.

The **processEvent** method is used upon a message reception and it behaves differently depending on the type of the Message received:

- **Payload** - If the message received is a payload, containing the whole message, recurring to the message identifier it checks if the message is new and if so, publishes it and spreads the payload or notification between his neighbours following the **isEagerSending** method.

- **Notification** - Node checks if it does not have that message published and if so, he adds that node to a request queue. If that queue is empty, he immediately sends a request message to get the desired payload from the node that sent that notification, recurring to the **SenderId** variable.

- **Request** - Makes a security check to verify if it has the content for that message. If that checks, it tries to send the payload to the requester node. In case the payload cannot

be successfully sent due to the link being overflown of usage, it sends a message notification with type **Failed** to the requester indicating the payload transmission failed.

- **Failed** - Simply checks if it still needs the content of that message and if so he sends a request to the next node resorting to the request queue.

In order to track which decisions were more worth to send the payload, which means that the receiver didn't have that message published yet, or on the other hand it should have sent a notification because on the reception moment that node already had published that message, it is implemented on the simulator a backtracking system to notify senders about the correctness of their decisions and the overall number of impacted nodes each decision originated.

3.2.4 *Network Generation*

The main goal of the network topology generator is to obtain a heterogeneous network, making nodes degree a decision factor and increase optimization possibilities. Therefore, it is desired for the network to have some super-nodes, nodes that have a considerably higher number of links, and in general a lot of nodes with a degree close to a minimum desired fan-out.

`Peersim` already has implemented several models to generate topologies such as **Wire-out**, **Watts-Strogatz** (36) (18) and **Barabási–Albert** model (29).

Wire-out configures every node with the same amount of neighbours, so the final topology does not have any super-nodes neither simulate interactions. Therefore, it was not used in the simulations.

Watts-Stroggatz model is a random graph generator that creates a scale-free network where the degree distribution follows a power law, leading to high clustering and the "small word effect" where the distance between any node of the network is surprisingly small. This model limitations resides in the requirement of having a fixed number of nodes and an unrealistic distribution of degree.

On the other hand the **Barabási-Albert** model also creates a scale-free network but with preferential attachment. This attachment is based with probability proportional to its degree. This model also follows a power law in degree, a property that is very frequently observed on social networking, citing networks and other real life situations (29).

Taking into account *Watts-Stroggatz* and *Barabasi-Albert's* model characteristics, the model generator option fell into the second due to the fact it follows social networking interactions making a more realistic distribution of degree for the case study where gossip protocol is used.

The Peersim's implementation of this model works as it follows: It initially connects an initial set of nodes with the same number as the degree with no links between each other. Then, the first added node is connected to all the initial nodes and after that the Barabási-Albert model algorithm is used normally, following a probability proportional to each node degree.

3.2.5 *Communication*

Communication in the simulation occurs on the message exchange that happens during the protocol normal behaviour, using the message class presented in Section 3.2.2.

In event-driven simulations, nodes send messages through a transport layer protocol. Peersim's interface in this event-driven simulation requires the usage of the transport layer to send messages, as shown in Listing 3.2 by specifying the receiver node, and the implementation of a method to process each message received, considered an event with the processEvent() method as shown in Listing 3.3.

```
((Transport)node.getProtocol(FastConfig.getTransport(protocolID))).
                                                    send(
                                                        node,
                                                        peer,
                                                        new Msg(msg),
                                                        protocolID);
```

Listing 3.2: Example of sending a message using the Peersim's transportation layer.

```
public void processEvent( Node node, int pid, Object event ) {
    Msg message = (Msg)event;
    if(message.getType()== Msg.Type.PAYLOAD){
        if(!this.publishedMsgs.containsKey(message.getId())){
        //Processing of the message
        ...
        //Spreading Payload
         spreadPayload(node, pid, message);
         }
    }
    else if (message.getType()== Msg.Type.NOTIFICATION){
        if(!this.publishedMsgs.containsKey(message.getId())){
            queueRequestMessage(node, pid, message);
        }
    }
}
```

Listing 3.3: Simplified version of processEvent, only containing the basic functionality of **Payload** and **Notification** types.

The loss of messages by overflow limits the traffic that goes through the transport layer by using the leaky bucket algorithm (7) on every connection. This algorithm is based on the notion of a bucket constantly leaking that will overflow if the quantity of water poured exceeds the capacity of the bucket. In this case, on the protocol implementation this translates to the bucket capacity being the parameter **Capacity**, the leaking rate of each link in which they process its bytes the **Leaking** parameter and the water, pouring, is the number of bytes contained in each payload that goes through the link. The leaking can be seen as bandwidth represented in bytes per round.

3.2.6 *Initializers*

Initializers are `PeerSim` components that run at the start of a simulation. They are useful to initialize some data structures of the protocol, define global variables and generate the topology. The implemented gossip protocol simulation is configurable with the following parameters:

- **Size**: Number of nodes in the network.
- **Simulation Time** - Total logical time of the simulation. Defined with the time of a cycle and the number of cycles.

  Number of Cycles: Number of cycles in the simulation where new messages are created.

  Cycle Time: Time of a cycle in logical time.
- **DELAY**: Percentage of the round duration that messages take to be sent where 10 means 10% of a round and 100 means one entire round. Needs to set minimum and maximum delays and each message has a random delay between the percentages set.

  DELAYMIN - Minimum delay a message takes to be delivered from one node to another.

  DELAYMAX - Maximum delay a message takes to be delivered from one node to another.
- **Protocol**: Java class that defines how to create activity and process events. It is where the gossip protocol is implemented.
- **Topology Initializer**: Name of the network generator model and its required parameters to create the network topology at the start of the simulation.
- **Protocol Initializer**: Name of the `Java` class that initializes the required variables of the **Protocol** chosen. Uses specific parameters to initialize the protocol class which are described in more detail in the next itemization.

- **Controllers**: Classes that can run every round with the objective of gathering information to print statistics, do some data cleaning to free unnecessary memory, dataset generation and queue next round activity.

The **topology initializer**, in order to produce the desired topology, requires the specification of the model for the graph generator `WireScaleFreeBA` which is the `Barabási-Albert` graph generator implementation made by `Peersim`. Following this it is required to set a minimum degree and if the graph will be directed or undirected.

The protocol initializer implemented uses the following parameters:

- **Chance**: Chance for a node to create a new message whenever a new cycle starts.
- **blockMessageCreation**: Number of rounds before the end of the simulation to block the system from creating new messages.
- **modelPercentage**: Percentage of nodes, randomly selected, that will use the ML model to make transmission decisions for nodes.
- **lazyChance**: Chance of deciding by the `lazy push` on a message emission, for the nodes not selected to use the ML model.
- **Capacity**: Defines the capacity limits in bytes of the simulation links.
- **Leaking**: Defines the bandwidth limits in bytes, representing the number of bytes links can send per round.
- **MessageSize**: Defines the limits of message sizes in bytes.
- **modelPath**: Local path where the Model Object, Optimized (MOJO) model is for the initializer to load them into a simulation global variable, so nodes can access the model for predictions.
- **datasetGenerator**: Defines if the simulation should create or append to a dataset from the events happening in the simulation, based on the selected version.
- **version**: Version of dataset to use in the simulation. Impacts the metrics observed and written into the dataset, as described in Section 3.3, and the necessary parameters used for the model to make predictions.

The **Gossip Initializer** class begins by initializing for all nodes, the variables that are used on the simulation normal execution, such as the published messages `HashMap`, list of nodes to request for each message, message origin tracker and some personal counters for statistics that will be used to create the dataset, help with decisions and obtain simulation summaries for comparisons.

In order to have links better than others and turn the link current state and the size of a message decision factors for model decisions, the initial bootstrapping of the leaky bucket algorithm parameters, link capacities, leaking and payload sizes are made with the **GossipInitializer** based on two numbers: a lower limit of the parameter and a random number up to a bound added to that lower limit. As a consequence, to create the range of values of the leaky bucket algorithm parameters, those limits need to be set into the configuration file assigning the respective lower limit of what that variable can be and a maximum bound that can be added to the lower limit. So, let $\boldsymbol{b}$ be the bound defined for that parameter and $\boldsymbol{l}$ the corresponding lower limit, for every link the capacity and leaking is calculated by:

$$Capacity = l_{capacity} + random[0, b_{capacity}] \tag{3}$$

$$Leaking = l_{leaking} + random[0, b_{leaking}] \tag{4}$$

Both links capacity and leaking scale their values based on the number of nodes in the network because the message flow that passes through each connection is directly proportional to the network size and this property allows the links to maintain the desired behaviour in different size configurations.

The nodes also obtain from the initializer the configured payload minimum size and bound and on each message creation the size is also calculated following the same logic, as stated in equation 2:

$$PayloadSize = l_{payloadSize} + random[0, b_{payloadSize}]$$

3.2.7 *Controllers*

Controllers are classes that monitor properties in the network and create new activity within the simulation periodically. `DegreeStats` controller verifies in each round the degree of the network and prints it under a certain method. For this we use two different print methods: **Frequency** and **List**. The first one prints degree frequency in the network, i.e. for each degree that exists in the network, informs about the number of nodes that have that degree. The second method, list, for each node identifier, indicates the corresponding node degree.

The output obtained from the frequency method can be converted into a bar graphic, as shown in Figure 7, to have a better representation of the distribution in the number of neighbours obtained from the topology generator model. Those `Peersim` controller classes are modified to write this output to a file instead of the console, to allow future analysis.

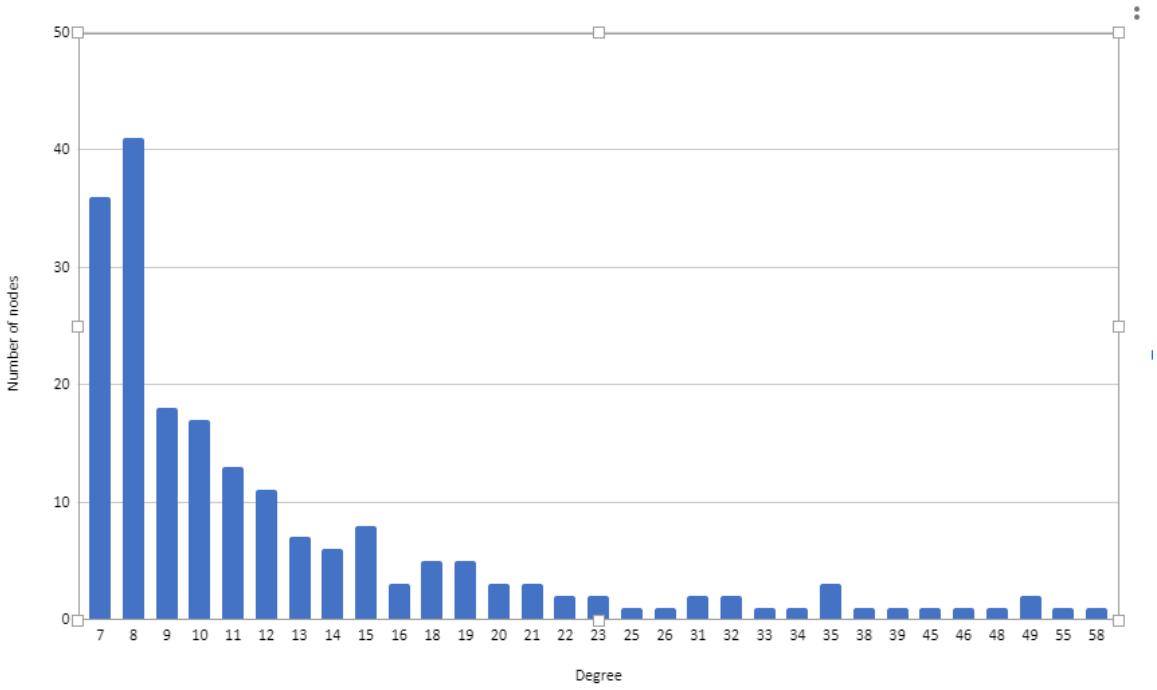


Figure 7: Degree distribution obtained from frequency method on a network with 200 nodes.

The **NewRoundQueuer** controller is responsible for queuing new activity in nodes by executing the **nextCycle** method and therefore, allow the normal functioning of the protocol by having new messages to start distributing. Besides that it is also responsible for:

- Blocking the creation of messages **n** rounds before the end of simulation.
- Identifying the moment the network circulation has warmed up the links state and activate the nodes to start the observation phase by collecting information about the simulation such as payloads or notifications sent.
- Write to the dataset file of that simulation if option `isDataset` is set.
- Write to the joined dataset file of that configuration, if option `isJoinedDataset` is set. Joined dataset is a different dataset containing an extra row identifying the simulation to which the event relates, and includes the events of several simulations of similar configurations.
- Clear events of messages that already were fully distributed in the network.
- Obtain simulation performance statistics of the observation phase in the last round and append them to the run comparison file.

- Initialize dataset headers with the columns names of the dataset version selected, if the file did not exist.

The way this controller clears events of messages that stopped being spread in the network is as follows. The transport layer takes about 40% to 60% of the round duration to deliver a message from one node to another so, if a message was not sent during an entire round that means it has stopped its distribution and their corresponding rows can be removed from the data structure into the dataset. Nodes keep a shared structure of active messages in the simulator that adds a new message identifier whenever a message is created. Nodes also share other data structure where they store identifiers of messages that circulated in the simulation during that round. So, every time a new round starts, every new message that circulates on the system (filtered by their message identification **messageId**) is added to that list. At the end of every round, the `NewRoundQueuer` controller creates a list of numbers that store the message identifiers of messages that stopped circulating that round which are the identifiers contained in the *activeMessages* list, but that do not appear on the *roundMessages* list. After that the controller accesses all the nodes and obtains dataset rows associated to those messages and writes them to the corresponding dataset files and at the end, clears the *activeMessages* list for the next round. This needs to be done because `Java` has a physical limit of memory that the simulation can not exceed and in each round thousands of dataset rows are created on a simulation and would end up on a memory leak.

To only consider statistics of when the simulation is working properly with the channels of communication not empty and in a stable state the system only starts observing metrics after the simulation passes a certain number of rounds, which by default is set to 20 (5). Not only that, it also excludes the events from messages created after the **Nth** round prior to the end of the simulation, to avoid collecting metrics during the cooling down period of the system. This is done by stopping message creation and waiting for the whole network to finish delivering the remaining messages, avoiding fallacious results and misleading rows on the dataset. This observation phase is done by registering the minimum and maximum message identifiers to observe the actions in the simulation. Since a message is created using an **AtomicLong** counter to give a unique incremental identifier to each message, whenever round **N** is reached, the controller checks the current counter value, **minimumIdStatistics**, and informs all the nodes in the network to start collecting metrics for events of messages with identifiers equal or higher to that one. Likewise, whenever the simulation reaches N rounds away from ending the simulation, it checks the current value of the counter, **maximumIdStatistics**, and informs all the nodes to not collect metrics or observe statistics for messages with identifiers higher than that value.

In the last round, the controller gets any remaining events and writes them to their respective files. After that, the controller processes a number of counters obtained from the protocol nodes and prints them to a file, like a summary of the execution that allows

analysis and comparisons. The metrics used for these summaries are described in more detail in the next Section.

3.2.8 *Run Summary*

Statistics are stored for analysis at the end of every simulation. They include metrics that identify the type of simulation used and other metrics that are related to the performance of the protocol and its type of behaviour. That said, the summary of runs has the following parameters:

- **Size** - Number of nodes in the simulation.
- **Rounds** - Number of rounds of the simulation.

In order for runs to be comparable they should have the same **number of nodes**, since the protocol performance metrics depend and scale based on that number.

The number of **rounds** is a complementary parameter since it only brings to the analysis a perspective for how long the simulation ran but as long as the observation interval considers a decent amount of rounds runs can be comparable even if a run has gone for 200 rounds and other for 1000 rounds since most performance metrics are averages based on the number of messages created.

- **Lazy Probability** - Probability for a non-model node to opt for a lazy transmission in each occasion.
- **Model Percentage** - Percentage of nodes in the network that will use the ML model to decide in each transmission the type to use.

The **lazy probability** and **model percentage** should be analysed together since they are correlated. The lazy probability represents the probability used in nodes not selected to use ML models to make a lazy transmission on each transmission. The amount of nodes that use the basic gossip algorithm is calculated with the model percentage since it is represented by the remaining percentage of nodes.

- **Model Predicts** - Enumerates the ratio of eager/lazy transmissions taken by the model.

  Model Eager Predicts - Number of predictions made for eager transmissions.

  Model Lazy Predicts - Number of predictions made for lazy transmissions.

Model eager and **lazy predicts** allow comparing model preferences for the type of transmissions and see the approach taken.

- **Correct Percentage** - Percentage of correct decisions taken under the perspective of the receiver having that message on the transmission moment, **predicted effectiveness**.

  Model Correct Percentage - Percentage of correct decisions taken by nodes that use the ML model to make decisions.

  Normal Correct Percentage - Percentage of correct decisions taken by nodes that use the normal algorithm to choose the transmission option.

- **One Hop Correct Average** - Percentage of correct decisions taken based on the feedback the receiver **nodeTo** gives. This differs from the first **Correct Percentage** since this feedback is given upon the arrival of the message being a more realistic metric than the prediction of a good attempt.

  One Hop Eager Correct - Percentage of correct eager decisions based on the feedback received after the hop.

  One Hop Lazy Correct - Percentage of correct lazy decisions based on the feedback received after the hop.

The **correct percentage** metric shows the percentage of decisions for each kind of nodes, under the evaluation type of the predicted effectiveness. In comparison, the **One Hop Correct** gives the effective correct percentage of nodes decisions basing on the feedback received from the backtracking system. In Section 3.4.3, it is specified the reasoning for having these two approaches.

- **Overflow Percentage** - Percentage of payload transmissions that got lost on the network due to overflowing the corresponding link capacity.

- **Decisions per Message** - Number of decisions made per message created in the network. It can also be related to the number of occasions nodes receive either a payload or notification for each message created.

- **Payloads per Message** - Number of payload transmission attempts per message created. Ideally the number should be close to the number of nodes in the network. Includes payloads sent from requests.

- **Successful Payloads sent per Message** - Number of payloads that were successfully sent per message created.

- **Failed Requested Payloads Per Message** - Number of requested payloads that failed to be delivered due to overflow, per message created.

The **overflow percentage** parameter shows the percentage of payload transmissions that got lost due to overflowing the link capacity and is a great indicator of the risk nodes are

taking or the frequency the same links are used. An eager dominant simulation is expected to have a higher overflow percentage than the opposite lazy dominant simulation.

The parameters Decisions, Payloads, Successful Payloads and Failed Requested Payloads per message are the main protocol metrics that allows the observer to understand, with more detail, what happened during the simulation. For example on a full lazy simulation, with lazy probability set to 100, it is expected to have the lowest number possible for decisions, successful payloads sent being the number of nodes in the simulation and attempted payload transmissions being very close to the number of successful payloads since payloads are only sent if really needed. On the other side, with a full eager simulation, the number of decisions, payload attempts and successful ones are expected to be considerably higher.

- **Should Eager** - Percentage of times the correct option should be Eager, basing on the moment the message is being sent if the receiver has that payload.
- **Network Percentage** - Percentage of the network reached by all the messages created. This is calculated by dividing the total number of messages published across the network over the number of messages created times the size of the network.
- **Hops Average** - Average of hops necessary to fully distribute a message across the network.
- **Time Average** - Average logical time necessary to fully distribute a message across the network.
- **Round Average** - Average number of rounds to fully distribute a message across the network. In comparison to time average this gives the number of cycles between message creation and the last reception instead of the logical time behind that interval.

The **should eager** parameter is a complementary metric that gives the perspective of how many times during the simulation, on the moment of the decision, the node should have opted for an eager transmission.

Finally, the last four metrics are very important to evaluate if a model improved the protocol or not. First, the **network percentage** is crucial to understand if the reliability that the gossip protocol is characteristic about remains true. It needs to maintain a very high percentage to guarantee it while improving the remaining metrics. The **hops** translating the number of necessary hops, in average, to fully distribute a message is a good metric to analyse if a shorter path has been found by the nodes, with some link preferences the model may find. The **time** and **round** metrics give a logical time of how long messages take to reach every node and are also essential to understand if the distribution became more efficient.

3.3 DATA COLLECTION

To create a machine learning model able to predict the correct type of transmissions, data needs to be collected periodically from the simulation with variables that can help with decisions for the ML training process. Therefore, on every round a controller collects a set of information from decisions of messages that ended their distribution within the simulation. It has two different configurations with different levels of detail in the information used and required observation.

3.3.1 *Simple Dataset*

The first version of the dataset aims at observing how well can the model improve the protocol performance using only simple and easily obtained metrics without much processing or observation. The data collected for each message transmission contains:

- **idEvent** - Unique identifier of the event. Also works as a simulation logical time tracker of when that event happened.
- **nodeFrom** - Identifier of the node that sent the message.
- **fromDegree** - Number of neighbours of the node that sent the message.
- **nodeTo** - Identifier of the node the message was sent to, the receiver.
- **toDegree** - Number of neighbours of the receiver.
- **messageId** - Unique identifier of the message this event is related to.
- **payloadSize** - Size in bytes of the message sent.
- **hopsTaken** - Number of hops that message has taken from the creator until reaching this node.
- **capacityLink** - Maximum capacity of the link between node **nodeFrom** and **nodeTo**.
- **actualCapacityLink** - Available capacity of the link between node **nodeFrom** and **nodeTo**.
- **leaking** - Leaking of the link between node **nodeFrom** and **nodeTo**. Translates to the number of bytes a link can send per round.
- **round** - Round of the simulation this event happened.
- **timePassedFromMessageCreation** - Logical time passed from the moment that message was created until this decision is being made.

- **decision** - Boolean value that tells if the message was sent as a payload by an eager decision, with a `TRUE` value, or as a notification by a lazy decision, with a `FALSE` value.
- **overallImpact** - Numeric value that tells how much impact making this decision ended up having in the network after feedback from the receiver.
- **usefulOption** - Boolean value indicating whether the decision is correct if the message could be delivered instantaneously.

The **usefulOption** field is set at the moment of that message is sent. On that moment the decision is correct when:

- The sender opts for sending the payload and the receiver `nodeTo`, at the moment of the transmission, does not have that message.
- The sender opts for sending a notification and the receiver `nodeTo`, at the moment of the transmission, already had that message published.

For the backtracking system, in order to give feedback to senders about the effectiveness of their decisions, that being payload emissions or notifications, every time a node receives a message of type Payload or Notification, it sends a feedback to the message origin, using the Node object of the sender directly, similarly to the way communication is made in cycle-based simulations. It is preferable to communicate this way instead of using the transport layer since this functionality does not belong to the common functioning of the protocol.

Based on this feedback, the **overallImpact** field is updated on message arrival. Whenever a new payload or notification is received, the sender will be notified of impacting positively or negatively a node in the network. Events of payload transmissions that do not come as a consequence of a request are incremented by one. If the sender was not the creator of the message it will propagate the feedback backwards until the message creator is reached, increasing by its path the corresponding events also by one. Following the same logic, events of notifications sent for nodes that already had that message published, when the message was received will also be incremented by one. With this logic, on eager transmissions the impact value translates to the number of nodes that received the message as a consequence of sending this payload to that node. Negative backtracking feedback is represented on this field as negative numbers as well. Figure 8 depicts 5 rows from the simple dataset used in the model training process.

3.3.2 *Complex Dataset*

The most complex version includes metrics obtained from averages of the current simulation performance to give more insight and try to help the ML model increase its accuracy. Those

	Ts	Id	payloadSize	nodeFrom	degreeAtual	nodeTo	toDegree	hops	messageID	capacity	actualCapacity	leaking	decision	useful	impact	timePassed
1:	1609081802210	226630	136	2	0	7	67	0	119	2700	2700	2300	FALSE	FALSE	0	0
2:	1609081802210	226631	136	2	0	8	58	0	119	2259	2259	2862	TRUE	FALSE	-1	0
3:	1609081802210	226632	136	2	0	9	43	0	119	2331	2331	2559	TRUE	FALSE	-1	0
4:	1609081802210	226633	136	2	0	10	42	0	119	2006	2006	1878	FALSE	FALSE	0	0
5:	1609081802210	226634	136	2	0	11	46	0	119	3915	3915	2021	TRUE	FALSE	-1	0

Figure 8: Example rows from the dataset of the simple version.

metrics intend to give some information about global execution performance but also how the links are performing. As considering the performance averages of only the entire execution could be misleading due to the volatile state of links, it is also important to have other metrics that reflect the recent performance from the last **N** occasions. Based on these facts this version of the dataset extends the simpler one with the following variables:

- **fullyDistributedEagerImpact** - Average impact obtained from eager transmissions by this node in the simulation.
- **fullyDistributedLazyImpact** - Average impact obtained from lazy transmissions by this node in the simulation.
- **fullyDistributedEagerConnectionImpact** - Average impact of eager transmissions in the link between **nodeFrom** and **nodeTo**.
- **fullyDistributedLazyConnectionImpact** - Average impact of lazy transmissions in the link between **nodeFrom** and **nodeTo**.

The fully distributed metrics show the global simulation impact averages for the specific node to all their neighbours and specific to only the intended connection for both transmission types. Those metrics don't give recency feedback since it only updates after the messages stop circulating throughout the network.

- **lastMessagesEagerMean** - Percentage from the previous **N** eager transmissions that had a positive one hop impact.
- **lastMessagesLazyMean** - Percentage from the previous **N** lazy transmissions that had a positive one hop impact.
- **lastMessagesEagerConnectionMean** - Percentage from the previous **N** eager transmissions that had a positive one hop impact in a link between **nodeFrom** and **nodeTo**.
- **lastMessagesLazyConnectionMean** - Percentage of the previous **N** lazy messages that had a positive one hop impact in a link between **nodeFrom** and **nodeTo**.

These metrics are evaluated at the time the receiver gets the message and give performance feedback on recently sent messages. It only considers the last **N** occasions and by default

it is set for 20 messages and are also separated into global neighbourhood and connection averages.

- **connectionLostMessagesAverage** - Percentage of payload transmissions lost by overflow in the link from **nodeFrom** to **nodeTo**.
- **connectionLastNLostMessagesAverage** - Percentage of payload transmissions lost by overflow by that node, from the previous **N** payload transmissions, in the link from **nodeFrom** to **nodeTo**.
- **totalLostMessagesAverage** - Percentage of payload transmissions lost by overflow by that node in the current simulation.
- **lastNLostMessagesAverage** - Percentage of messages from the previous **N** payload transmissions that were lost by overflow by this node.

To evaluate the connection availability and state of links during the whole simulation and in recent times four metrics are used: **connectionLostMessagesAverage**, **connectionLastNLostMessageAverage**, **totalLostMessagesAverage** and **lastNLostMessagesAverage**. They can be crucial to avoid sending payloads on unavailable links during that round.

- **connectionTotalCorrectDecisionsAverage** - Percentage of correct decisions by this node in the connection from **nodeFrom** to **nodeTo**.
- **connectionLastNTotalCorrectDecisionsAverage** - Percentage of correct decisions by this node, in the previous **N** situations, in the connection from **nodeFrom** to **nodeTo**.
- **totalCorrectDecisionsAverage** - Percentage of correct decisions by this node.
- **lastCorrectDecisionsAverage** - Percentage of correct decisions from the previous **N** decisions by this node.

Finally, to have a metric able to give some insight in how obvious it is to make the right choice in that node, and more specifically from that node to a specific neighbour, upon feedback on the usefulness of a node decision some personal counters are updated for that purpose. From all the extra metrics of this version, in the perspective of helping the ML model to choose the right transmission type, the correct decision metrics should probably be the less impactful compared to the other ones but still serve as a research target.

All of these metrics require the simulation to be running for a while since most of them use counters and lists of `Boolean` values based on the protocol's activity. Therefore, since there needs to be enough sample size in order for those values to be reliable, they are not used until a set minimum of situations is reached. Not having the counters updated can have a big impact in the choices the nodes will do at the start of simulations whenever using

	Ts	Id	nodeFrom	round	nodeTo	toDegree	messageID	payloadSize	hops	capacity	actualCapacity	leaking	timePassed	decision	useful	impact	fullyDistributedEagerImpact
1:	1607011424722	11095236	199	203	21	35	4091	267	3	2759	2759	2471	3744406	TRUE	FALSE	-1	0.07575098
2:	1610908810393	237750	6	6	7	71	115	273	0	3462	3462	2813	0	FALSE	FALSE	0	0.02127660
3:	1610908810393	237751	6	6	8	64	115	273	0	3948	1715	1468	0	TRUE	FALSE	0	0.02127660
4:	1610908810393	237752	6	6	10	48	115	273	0	3180	3180	2284	0	FALSE	FALSE	0	0.02127660
5:	1610908810393	237753	6	6	13	21	115	273	0	4740	4554	1529	0	TRUE	FALSE	0	0.02127660

	fullyDistributedLazyImpact	lastMessagesEagerMean	lastMessagesLazyMean	fullyDistributedEagerConnectionImpact	fullyDistributedLazyConnectionImpact	lastMessagesEagerConnectionMean
1:	0.6978425	0.05	1.00	0.01340694	0.8250556	0.10
2:	0.4518519	0.00	0.95	0.00000000	0.8750000	0.00
3:	0.4518519	0.00	0.95	0.00000000	0.8333333	0.00
4:	0.4518519	0.00	0.95	0.12500000	0.5000000	0.00
5:	0.4518519	0.00	0.95	0.00000000	0.3333333	0.05

	lastMessagesLazyConnectionMean	connectionLostMessagesAverage	connectionLastNLostMessagesAverage	totalLostMessagesAverage	lastNLostMessagesAverage
1:	1.00	0.00000000	0.00	0.09647997	0.20
2:	0.95	0.00000000	0.00	0.03194321	0.20
3:	0.95	0.05882353	0.15	0.03191489	0.20
4:	0.95	0.00000000	0.00	0.03191489	0.20
5:	1.00	0.00000000	0.00	0.03188663	0.20

Figure 9: Example rows from the dataset of the complex version.

a model. In practice, the ideal situation would be, at the start of the simulation opting for the eager transmission a considerable amount of times so whenever system performance is observed all the simulations begin with the network links at more or less in the same state and the nodes having their personal counters giving some more realistic insight of what's surrounding him, like knowing the links with less probability of dropping a message or with a higher impact average.

Taking that logic into consideration, there are several ways of setting a workaround to solve this problem and start making the warm-up of the system but the simulator has been implemented as follows: until there is enough information about the network, which in this case is to have 10 samples of each decision, the methods give perfect values for every metric and make the node choose from these values. That way nodes will explore their options and eventually these metrics will converge to more realistic values before the observation phase starts. Figure 9 depicts 5 rows from the complex version dataset used in the training process.

3.4 MACHINE LEARNING MODEL

This section describes how AI is used to improve the protocol and the changes to the gossip implementation to use the obtained model and be able to evaluate the protocol.

3.4.1 *Learning Framework*

The framework that generates the ML model uses the `H2O.ai` platform (10). It allows running in R and facilitates the use, test and validation of models since the programming language supports inspect, cleaning and modelling data to better prepare the data that will end up feeding the ML model, supporting the decision-making. It has interesting features such as memory efficiency in data manipulation and the ease with which prediction models can be generated and integrated in Java applications, using MOJOs. Furthermore, it is also useful because with access to `AutoML` the ML workflow gets automated with automatic training and tuning of various models within a time limit set and the distributed in memory processing can speed up to 100 times without degradation of the model accuracies. This is useful because there are several options on what set of variables to use as prediction factors or the definition labels of a correct decision, requiring this iterative process to be repeated a considerable amount of times. Therefore, `R` is used for the process of data cleaning and preparation and `H2O.ai` features for the training process and deployment.

3.4.2 *Global Architecture*

Figure 10 shows how the model is integrated in the simulator and used for evaluation of the protocol performance. Initially, some runs are queued using a baseline configuration **(1)**, such as a common `flat` gossip algorithm, to generate a dataset **(2)**. The dataset is then utilized in the training process to generate a model **(3)** able to predict the decision on each transmission. This model is obtained as a MOJO to be used inside the simulator. The evaluation is done by exporting the MOJO from the `H2O`'s instance and importing it in the simulator by entering its location path in the `Peersim`'s configuration file **(4)**. Later, a run is queued using the updated configuration file **(5)** and the **Gossip Initializer** class accesses the configuration path of the model and loads the model into the `Peersim`'s environment **(6)**, making it accessible for every node to obtain predictions. After every run a new line is printed with a summary of simulation configurations **(7)**, to identify the run, and performance metrics making it possible to analyse and compare the results from the original gossip runs with the performance of the ML model.

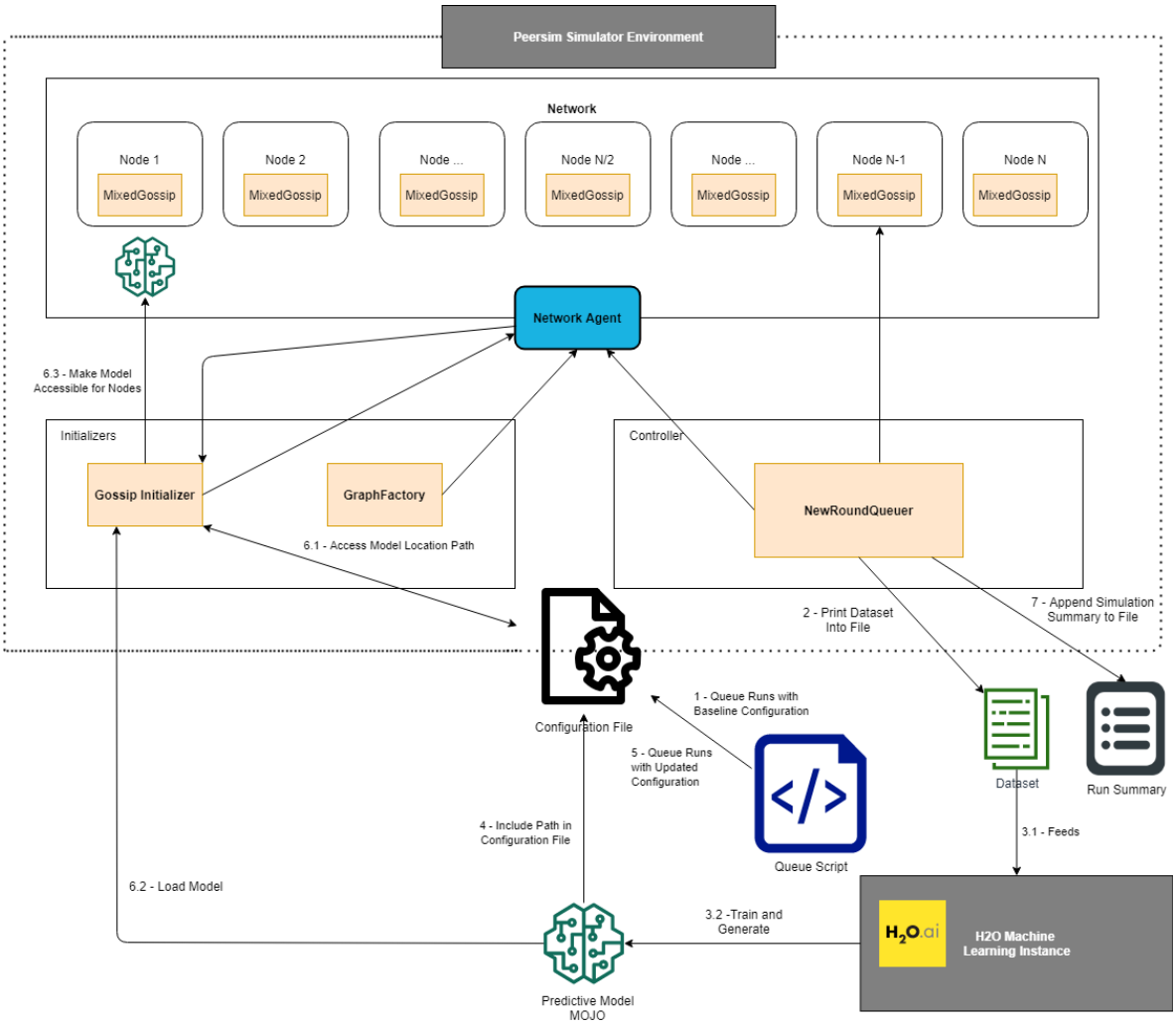


Figure 10: Global view of `Peersim` and `H2O.ai` interaction.

3.4.3 *Training Process*

ML learns from previous experiences to find patterns within data and help to make decisions. Human intervention only occurs to ensure the quality of the data and therefore, guarantee the model trains well and produces accurate results. That's why the data preparation can even have a more important role than the model construction or the algorithm chosen to build it.

Using the supervised learning paradigm, most specifically classification on the type of transmission to use, requires a transformation on the dataset to precisely define what is a correct decision. This impacts the behavioural approach of models.

The datasets used for this purpose are described in Section 3.3. They include environment metrics and performance averages and one of the goals is to find which metrics are more

Id	nodeFrom	round	nodeTo	hops	capacity	decision	useful	impact	...
2057	199	379	13	2	2103	TRUE	FALSE	-1	...
6078	6	6	7	0	4794	FALSE	FALSE	0	...
2215	6	6	8	0	3463	TRUE	TRUE	1	...
2274	6	6	9	0	2020	TRUE	FALSE	0	...
1577	6	6	10	0	3494	TRUE	FALSE	0	...
1578	6	6	13	0	3403	FALSE	TRUE	1	...

Table 1: Sample of the dataset obtained from a simulation.

important, to increase the overall decisions accuracy and as a consequence improve the protocol performance by showing some of the following factors: distributing the messages faster, sending less messages, having a higher reliability and obtaining an "emergent eager structure" based on the selected factors.

The aim is to show the potential of using an ML model under a certain environment. To achieve that, some runs are made in the simulator under the same configuration to build a dataset containing all the decisions during the observation phase of all simulations, such as Table 1 exemplifies. With the dataset having network parameters such as capacity or leaking of the links and performance averages such as percentage of lost messages in the node, the goal is to use different combinations of the dataset as an input and try to get the best combination able to improve the performance of the common protocol. With this approach, the dataset encompasses all kinds of possible situations within the simulation with various levels of network saturation, number of neighbours, time of the message in the simulator and decisions.

In order to obtain a model able to improve the protocol giving accurate predictions, some important steps need to be followed (11). Those are:

1. Identify business problem and definition of success

The problem is not having an algorithm that adapts to the topology state in P2P networks and that finds the emergent eager structure in real time. In this work the goal is to improve the gossip protocol by using the network links to their full potential. This can be achieved in different ways and the success resides in predicting the correct transmission strategy on each situation to increase the distribution efficiency.

2. Identify and collect data

Having a diversified dataset, with good quantity and quality data is key to make the predictive model effective and accurate. On every decision a node takes, it creates a dataset row that contains network and performance metrics. On every round of the simulation,

rows correlated to decisions of messages that ended their distribution on the network are appended to the dataset. To build that dataset, it is necessary to run one or more executions of the simulator, associated to a configuration file that defines the environment it will run on. That configuration file, named in this example as `mixed-mean.txt` needs to have the **isDataset** option active to generate the dataset.

The execution of the simulation is done as shown in Listing 3.4.

```
java -cp "peersim-1.0.4/peersim-1.0.4.jar:jep-2.3.0.jar:djep-1.0.0.jar:h2o-genmodel.jar
:target/peersim.tutorial-1.0-with-dependencies.jar" peersim.Simulator mixed-mean.txt
```

Listing 3.4: Simulation execution script.

3. Transform the data

Raw data needs to be transformed and prepared to serve as basis for training the ML model. First, ML algorithms expect numbers, so we need to represent all the fields of our dataset as numbers. Since most of the chosen metrics already have numeric format, such as integers or float numbers, the remaining `Boolean` can easily be represented as 0 or 1 in binary representation, and string representations labelled as categories. Some algorithms can manage non-numeric representations but preferably this situation should be taken care before the model training process starts.

After loading the data and transforming non-numeric fields, the response field of the model should be prepared. In this case, since the protocol is based on two transmission types, the response field **ShouldEager** is a numeric 0 or 1 field where 0 represents a lazy transmission and 1 an eager transmission. That said, the response field **ShouldEager** is calculated using the **decision** taken and one of two other fields, **useful** and **impact**, depending on the type of approach intended. After creating the **ShouldEager** field for training purposes, those three fields are removed since they don't represent a decision factor. All the remaining fields from the original dataset can be used as predictors for the model and, as the response, the new field **ShouldEager** is used.

Another important detail in data preparation is shuffling the data frame to prevent learning problems that arise from the order of the dataset. This approach prevents the model from overfitting as a result of the bias of the dataset being ordered from the simulations and allows the algorithm to converge faster since the randomness allow gradients to be more variable.

Finally, data is split into three different sets to be used in the model training process: the training, test and validation sets. The training set contains the data that is used to train the model and has impact in the calculation of the weights. The validation set contains the data used to evaluate the models and helps to tune the model hyper-parameters, during the training process. The test set is used to assess the models performance and help to choose

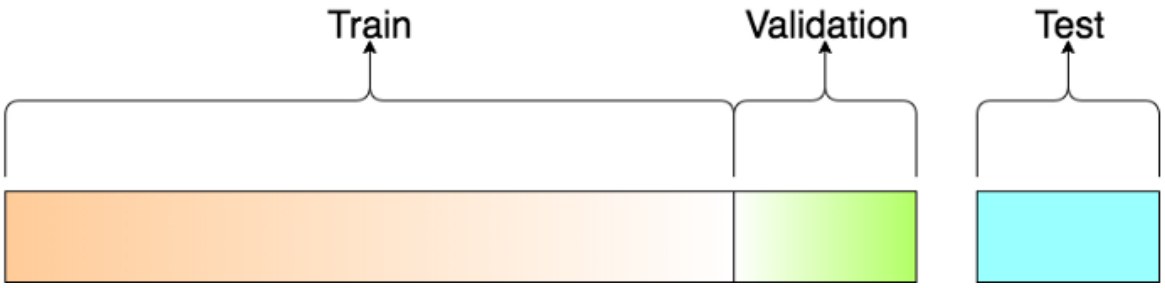


Figure 11: Dataset split ratio.

the best models to use and test in the simulator. Therefore, train and validation sets are used during the training phase and the test set only is used once the training is complete (2).

Since the dataset used is large giving a good portion to the validation and test sets won't affect the training process. Following that logic, 70% is assigned to the training set and the remaining 30% equally divided for the test and validation sets, such as Figure 11 depicts.

4. Determine the model's features and training

The next step to obtain the model is to choose an algorithm for the type of problem we have in our hands and the type of output intended. Since the goal is to have a model that can predict the correct transmission type this is a classification problem, which the target value is discrete and can be either **Eager** or **Lazy**.

To obtain the predictive model we have to choose a classification method to assign to each row of the dataset what transmission type should be chosen. To define a correct decision there are two specific ways of classifying a correct decision in R:

- Basing the **ShouldEager** response on the **impact** number, a metric of the legitimate effectiveness when the receiver processes the message.
- Basing the **ShouldEager** response on the **useful** metric that translates if the node has that message published at the moment of payload transmission.

The first one gives the effective perspective of the decision based on the feedback from receiver nodes. This version might result in a conservative approach since nodes with high degree have a higher chance of already having that message, lowering the frequency that the `eager push` is used to increase the accuracy. The other perspective, under the view of the receiver having that message on the moment the transmission is starting can benefit more eager transmissions because it "dissimulates" the model into believing that the eager transmission is the correct option even if during the transmission the receiver obtains the same message from another node. This makes eager transmissions that were really close, in time, to being effective as **ShouldEager**, considering them as good tries. There is

another way of forcing a more eager dominant model but required a tweak on the impact implementation scoring algorithm. The implementation stores, after publishing a message, the amount of notifications and payloads received of that message. For the tweak, instead of giving immediately a 0 **impact** value to eager transmissions of messages the node already has, it checks the amount of times that payload was received and gives a positive 1 **impact** whenever that payload was received until a pre-defined number of times, which in this case is set to 2. That way, the model can influence nodes to make more eager transmissions with some logic behind the decisions.

The algorithm used to train the models is the `Gradient Boosting Machine`, GBM (9)(23). That algorithm is a machine learning technique used in classification and regression problems that produces a prediction model in the form of an ensemble of weak prediction models. The predicted results improve by increasingly refined approximations. The reason this is used is because it represents a very powerful technique to build predictive models and also addresses concerns about multicollinearity problems even though later those variables are analysed and highly correlated variables removed (26).

The script made in R, using H2O's functionalities uses a **Grid** search in order to generate models with different hyper-parameters, stored in the `hyper_params` variable and 10 of them are randomly obtained from the set of all possible hyperparameters value combinations following a stopping criterion of a maximum number of models and runtime limit, as shown in Listing B.3. At the end of the training process all the models generated are ordered by their accuracy, and the most accurate downloaded to be tested in the simulator, as explained in the next point.

5. Evaluate the model

To be able to test the model performance in the simulator, after the training process finishes the best models need to be exported from `H2O.ai` as a MOJO, as shown in Listing B.4, to verify if they in fact obtain better results than the baselines obtained from the protocol common performance and how it fits within the possible performance limits discovered.

The model is evaluated by analysing the run summary metrics obtained, preferably as an average of a number of runs, and comparing the obtained metrics to the previous baselines and goals of improvement. In general it should focus on protocol metrics such as the number of payloads sent (that translates to the bandwidth used), rounds needed to distribute messages, ratio preference between the transmission types, percentage of lost payloads and percentage of messages that all nodes published and identify the models with the best trade-off of each type, depending on the use-case. Besides that, it allows observing

the accuracy of the model inside the simulation since it takes into account the correctness of one hop feedback and the projection of a correct decision when sending the message.

Some modifications need to be made to use model model predictions in the simulator: First, having a way to select the nodes that use the model, allowing to switch between common simulations, all nodes using the model or a mixed version using a percentage for each type of nodes, by the `GossipInitializer`. The second step is to implement a way to use this model in the selected nodes and obtain the desired prediction in every occasion. This is done is by creating a **RowData** object, as shown in Listing 3.5, where we assign for each predictor field used in the model the corresponding value of the actual state and obtain a prediction from the MOJO model.

```
RowData row = new RowData();
row.put("Ts", String.valueOf(ts));
row.put("Id" , String.valueOf(IdEvento));
row.put("nodeFrom" , String.valueOf(nodeFrom));
row.put("round" , String.valueOf(round));
row.put("nodeTo" , String.valueOf(nodeTo));
row.put("toDegree" , String.valueOf(toDegree));
row.put("messageID" , String.valueOf(messageID));
row.put("payloadSize" , String.valueOf(payloadSize));
row.put("hops" , String.valueOf(hops));
row.put("capacity" , String.valueOf(capacity));
row.put("actualCapacity" , String.valueOf(actualCapacity));
row.put("leaking" , String.valueOf(leaking));
row.put("timePassed" , String.valueOf(timePassed));

// OBTAIN PREDICTION
BinomialModelPrediction p = this.model.predictBinomial(row);

if(p.labelIndex == 1)
    response=true;
else
    response=false;
```

Listing 3.5: Necessary code to obtain a prediction from a model stored in the **model** variable.

The model access has been implemented with the following logic: The model object is loaded into a shared variable across all the nodes to access predictions directly. With this approach there is no problem of the Java memory exceeding the usage limit since it only loads one model, and it remains loaded during the whole simulation making the simulation execute more smoothly and faster.

6. Improve the previous iteration model

After evaluating the model in a specific scenario, it can be revised to achieve better results or another trade-off. This can be done by changing hyperparameters, use only a sub-set of the dataset or even changing other configurations on the model training process. These tweaks may be crucial to obtain an optimal model in the configured environment and is a necessary step to show the potential ML can have in improving these protocols.

4

MODEL EVALUATION

The evaluation of the system was conducted using the `Peersim` simulator, with the necessary adjustments approached in Section 3.4.3. In each use-case the simulator configuration used to generate the dataset is the same as the one used for the model performance evaluation. On the same note, every performance metric shown, due to the existence of randomness of several variables such as network topology or link capacities, is an average of 5 runs due to the intrinsic stochastic nature of simulations (21). It is also important to note the results obtained are related only to the observation phase which excludes the activity of the first and last 20 rounds avoiding the warm-up and cool-down deterioration of results and assuring the results obtained are reliable under the same network state conditions, with previous usage of the network and all the messages created during the observation phase taken into account.

Initially, this chapter shows the performance limits the protocol can achieve and the interval of values where models can obtain different performance trade-offs from. After this, for each use-case it is presented the initial setup for the dataset generation, training improvement attempts and performance analysis with the goal of having the most optimal trade-off within the limits obtained while maintaining the protocol main properties.

4.1 BASE CONFIGURATION

One crucial factor to understand the results obtained is to know the environment configuration the simulator is running on and how it might impact the results we get, such as the limits obtained. The base configuration has the following fixed parameters:

- 200 Nodes.
- 400 Cycles.
- Message transmission taking between 40% and 60% of round duration and there is a 5% chance of a message being created every cycle on each node.

- Network topology following Albert-Barabási graph generator algorithm, using a minimum fan-out of 7 nodes.
- Observation phase excluding messages created during the first and last 20 rounds.
- Capacity of links ranging from 1800 to 5000 bytes.
- Bandwidth of the links ranging from 1200 to 3000 bytes / round.
- Payload size of messages ranging from 100 to 350 bytes.

The number of nodes paired with the topology chosen creates a network with a certain number of super-nodes and aligned with the minimal fan-out value it impacts the diameter of the graph that is the maximum "shortest-distance" between two nodes existing in the network. This may impact the number of hops metric, and consequently, time to fully distribute messages.

The message transmission duration impacts the logical time to distribute messages. The randomness obtained from the duration on this message can also make small changes to the optimal path to send the payloads in each occasion.

The capacity, leaking and payload size are parameters related to the leaking bucket algorithm that simulates the bandwidth and burstiness capacity of the network. The way this bootstrapping is done in an attempt to simulate the real world by asserting for a full eager simulation a 50% chance to lose a message by overflow. This way, the protocol still maintains their properties but under a very big rate of overflown messages. If the chance of creating a message gets modified it is crucial to also modify the leaking bucket algorithm configurations in order to not lose this property. The choice for the network bootstrapping property of 50% lost messages was made with the objective of turning the link state a decision factor for nodes, otherwise always using eager transmissions would be optimal since it would deliver the messages faster with no cost at all.

```
1  #!/bin/bash
2
3  for j in {0..4..1}
4    do
5          STR1="random.seed [0-9]*"
6          RANDOM=$$
7          STR2="random.seed $RANDOM"
8          sed -i "s/$STR1/$STR2/g" mixed-mean.txt
9          java -cp
               "peersim-1.0.4.jar:jep-2.3.0.jar:djep-1.0.0.jar:h2o-genmodel.jar:target/
               peersim.tutorial-1.0-SNAPSHOT-jar-with-dependencies.jar" peersim.Simulator
               mixed-mean.txt
10   done
```

Listing 4.1: Script to execute 5 simulations with randomly generated topologies.

Metric	Full Eager Simulation	Full Lazy Simulation
Message OverFlow	52.43%	0.04%
Payloads Attempts	2671	199.65
Successful Payloads Sent	1270	199.54
Failed Requested Payloads	0	0.0764
Hops Average	3.044	3.005
Rounds to Distribute	1.9821	4.5551
Predicted effectiveness	23.820%	50.099%
One Hop effectiveness	7.32%	74.348%
Network Percentage	99.06%	99.98%

Table 2: Performance limits of full lazy and full eager simulations.

Listing 4.1 shows the script used for obtaining metrics in the evaluation process. The results obtained are an average of 5 executions with the described configuration stored in the `mixed-mean.txt` file. From line 5 to 8 a new random seed is generated and included in the configuration for the topology generator component to obtain a new random topology in each execution, preventing the model from overfitting from always using the same topology.

4.2 PROTOCOL PERFORMANCE LIMITS

We test the opposite ends of the protocol to understand the performance limits in which the model will try to obtain the best trade-off from. This means the performances of full eager and full lazy simulations are obtained and analysed. The full eager simulation, after publishing a new message, will spread amongst neighbours always using eager transmissions while the full lazy simulation will always send a notification to their neighbours. Others tests are made on some metrics by the range of lazy percentages used to understand how changing that parameter can affect the protocol behaviour within the network.

Table 2 shows the performance from the opposite ends of the protocol. In the worst case scenario, the full eager simulation has an average of 52.43% lost messages and in a full lazy simulation, where all the payload transmissions are previously requested there are only a few messages lost representing nearly 0.04% of payload transmissions. Figure 12 shows the variation in the percentage of messages lost per chance used for lazy transmissions.

No conclusions can be made about the failed requests metric since the full eager simulation makes no requests and therefore can't have any requested payload to fail. On the other side, on a full lazy simulation the links tend to be always available due to the low number of payload emissions and there is a very low chance of any requested payload to fail.

Meanwhile, there is a big disparity on the number of payloads emission attempts and successful ones that assign the limits for those metrics. On a full eager simulation the network attempts 2671 payload emissions per message created. This translates to a waste

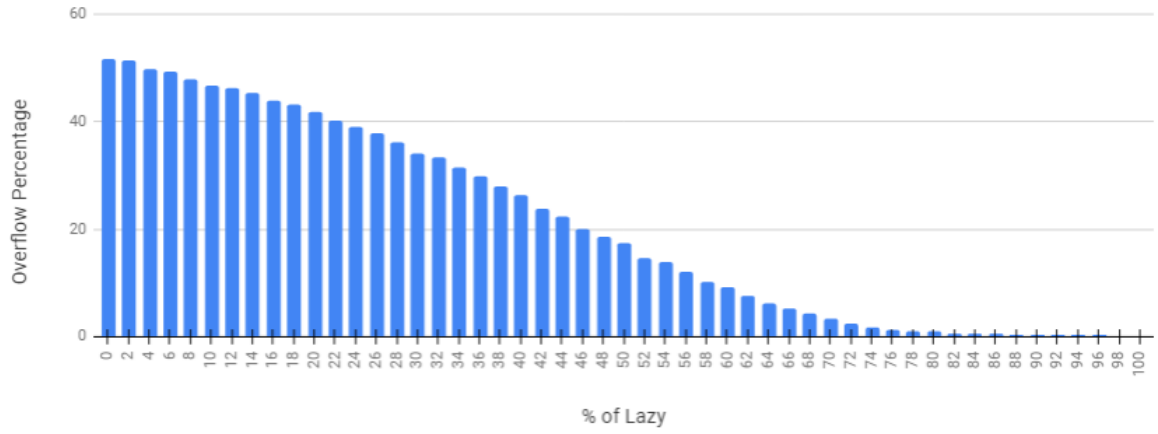


Figure 12: Percentage of overflow on different configurations.

of 2472 payloads that were unnecessarily sent during the simulation and represents the maximum limit of payload attempts, since at every node it sends the payload for all of their neighbours. On the other side, the full lazy simulation obtains nearly the expected 199 payload emissions since the links are rarely overflown by the payload emissions and is necessary to have 199 correct payload emissions to fully distribute a message in a network of 200 nodes. Figure 13 shows the disparity between the number of payload attempts and successful transmissions per lazy transmission rate. It is interesting to highlight that the number of payloads successfully sent is nearly the same from 0 to 50% and starts decreasing to 200 payloads from that point until the full lazy simulation.

The necessary hops to fully distribute the payloads to the entire network under this topology seems to not be affected by simulations that use the same strategy every time, having similar averages around 3 hops.

The necessary time to distribute payloads, based on the number of rounds passed from the moment the message was created is where a big difference can be highlighted. The full eager simulation takes nearly 2 rounds from the moment the message is created until the last node receives its message. This might be really close to the minimum time in which this protocol can distribute messages to the entire network based on their necessary hops and transmission time, assuming some optimal path might not be available due to links capacity limitations. On the other side, on a full lazy simulation messages take around 4.6 rounds to end their distribution. This disparity is expected because for each notification of a new message in a full lazy simulation, the node has to request the payload and wait for the reception of the requested payload, bringing this overhead in time. The remaining distribution times are depicted in Figure 14.

Finally, comparing the network percentage reached, in a full eager simulation in average 2 nodes don't receive each message created, since only 99% of the expected number of

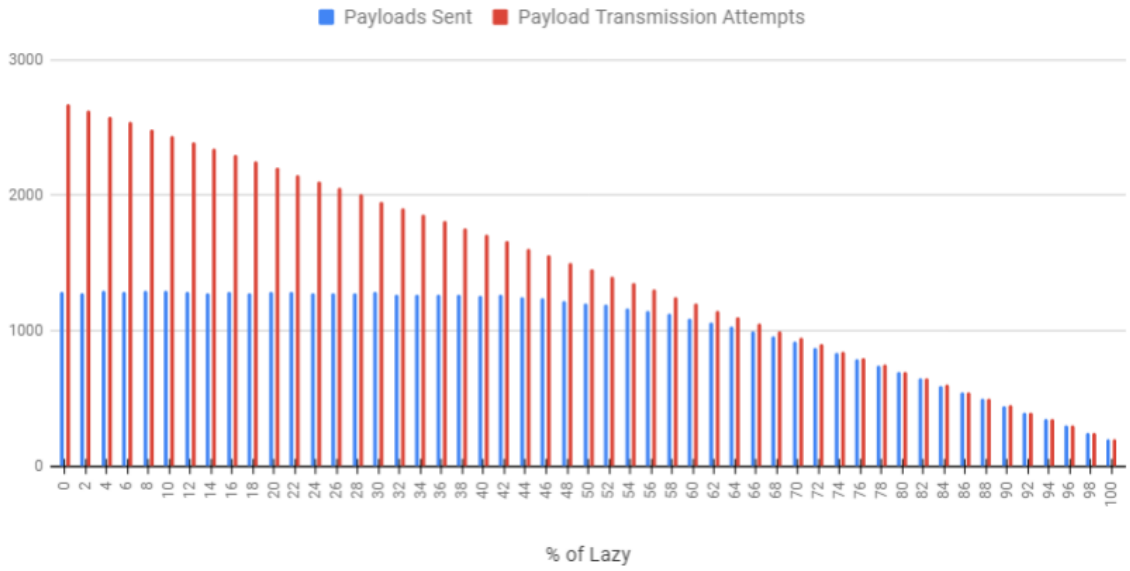


Figure 13: Payload attempts and successful transmissions on different configurations.

published messages is found. In a full lazy simulation, the reach is higher hitting 99.98% which means that very rarely a node in the network doesn't receive a created message. Based on this we can conclude that by using the full eager simulation, even though the messages get distributed faster the protocol loses some reliability.

The one hop and predicted effectiveness metrics give perspective of how they translate the two definitions of what are **correct decisions** used on the ML training process to the simulation. It doesn't set any upper or lower limits on that metric, but it is also interesting to analyse them. The one hop effectiveness is way higher on a full lazy simulation than a full eager simulation since after the first two hops of distribution, it is expected for a lot of nodes to already possess that message published at the moment the message is received. Also, on a full eager simulation, with 2671 tries of eager transmissions, only 199 of them can be correct which means $\frac{199}{2671} * 100$ translates to the 7.4% of correct decisions obtained. Since the predicted effectiveness does not take into account the message transmission time and assumes an instantaneous delivery of the message, the percentage of correct eager decisions tends to be higher than in the legitimate one hop version.

4.3 Mixed Simulation Baseline

Mixed simulation uses a configuration that has 50% chance of using either of the transmissions types. Since this is the configuration used to build the dataset with, it is is valuable to know how it performs for comparisons in the evaluation of the model's performance. They

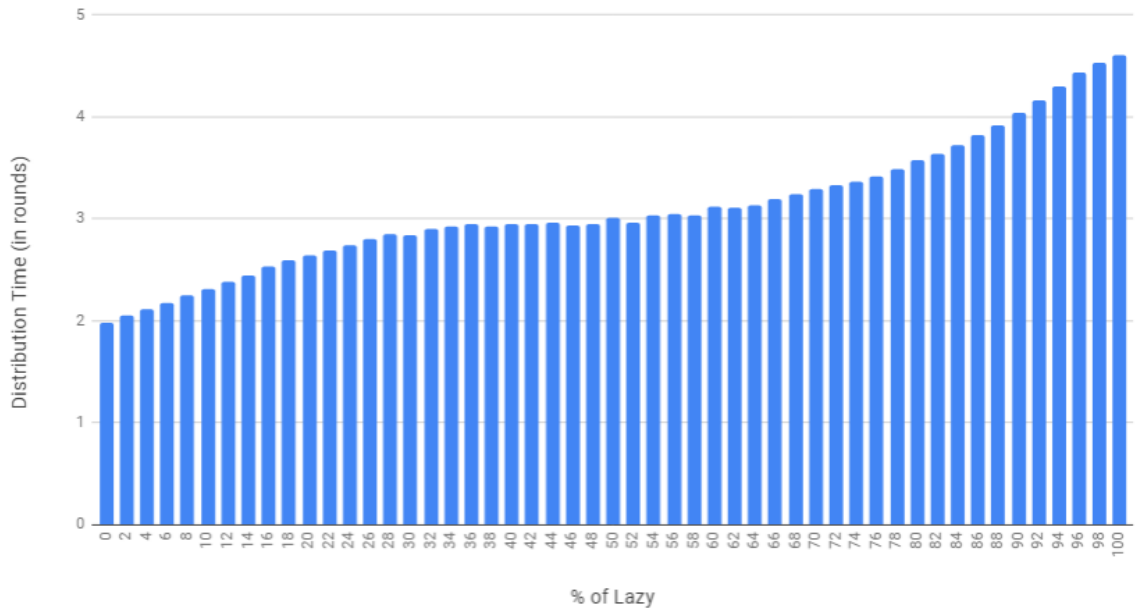


Figure 14: Necessary distribution time per percentage of lazy.

will also serve as baselines to know what the model approach is, where improvements are made and the resulting degradation of other metrics from the trade-off. The baseline results are shown in Table 3.

In this case, the protocol performance is in between the full eager and full lazy configurations. Only 16.87% of the payloads sent are lost by overflow. Since links are used to their limits, whenever a request is made the chance of the link not being able to send the payload is certainly higher than in the full lazy simulation which translates to having an average of 22.65 requested payload lost per message created. The number of payload emission attempts is 1453.44 which is nearly half than the full eager configuration. The most optimal path is not always used since the average number of hops to distribute messages increased from the full lazy and full eager configurations, coming from 3 to 3.79 hops. Relatively to the number of rounds needed to distribute it takes in average 3 rounds, which is 1 round more than in a full eager simulation and less 1.55 rounds than a full lazy simulation. In this configuration, both one hop and predict effectiveness are close to 50% and are an important baseline metric to see if models improve the protocol performances based on the correct decision definitions. Finally, the network percentage reached by messages is very high, 99.99%, similar to the full lazy simulation.

Metric	Mixed 50% Simulation
Message OverFlow	16.87%
Payloads Attempts	1453.44
Successful Payloads Sent	1208.29
Failed Requested Payloads	22.65
Hops Average	3.79
Rounds to Distribute	3.00
Predicted effectiveness	50.00%
One Hop effectiveness	46.87%
One Hop Eager Correct	13.42%
One Hop Lazy Correct	82.38%
Network Percentage	99.99%

Table 3: Performance obtained from the mixed gossip simulation.

4.4 ONE HOP EFFECTIVENESS

In this initial case the goal is to see how the model performs basing the **ShouldEager** label on the **Impact** column that stores the one hop effectiveness of decisions, based on feedback from the receivers, as shown in Listing 4.2. After generating a dataset running several 50% eager, 50% lazy simulations using the base script of Listing 4.1, it is reordered, and then split following the ratio of 70% for training, 15% to the valid and the remaining 15% to the test set. After training the model the two most accurate models generated, with the accuracy obtained from the test set, are downloaded and tested in the simulator.

```
response <- "ShouldEager"

table(dataset$ShouldEager)

dataset$ShouldEager[dataset$decision == "TRUE" & dataset$impact >1]<- 1
dataset$ShouldEager[dataset$decision == "TRUE" & dataset$impact <= 1]<- 0
dataset$ShouldEager[dataset$decision == "FALSE" & dataset$impact == 1]<- 0
dataset$ShouldEager[dataset$decision == "FALSE" & dataset$impact != 1]<- 1
```

Listing 4.2: Defining the ShouldEager columns based on the one hop effectiveness in R.

4.4.1 *Simple Model*

The goal is to assess if a model based only on link state metrics and the degree of the receiver is able to improve the accuracy and get a better trade-off in time and bandwidth used. This version lacks information that might be important for decisions, with little to no observation, making it easier to implement in the real world.

Therefore, the two most accurate models generated, with the accuracy obtained with the test set shown in Figure 15, are downloaded and tested in the simulator.

```
> for (i in 1:search_criteria$max_models) {
+   gbm <- h2o.getModel(sortedGrid@model_ids[[i]])
+   print(h2o.auc(h2o.performance(gbm, valid = TRUE)))
+ }
[1] 0.8795302
[1] 0.8781926
[1] 0.8761991
[1] 0.8741855
[1] 0.87069
[1] 0.8699547
[1] 0.8665934
```

Figure 15: Accuracy on the test dataset of the simple models during the training process.

Metric	Mixed 50% Simulation	Best Model	Flat 10% Eager
Message OverFlow	16.87%	0.96%	0.31%
Eager Percentage	50%	9.66%	10%
Payloads Attempts	1453.44	335.64	449.51
Successful Payloads Sent	1208.29	332.41	448.11
Failed Requested Payloads	22.65	0.47	1.05
Hops Average	3.79	2.13	4.71
Rounds to Distribute	3.00	2.93	4.04
Predicted effectiveness	50.00%	57.74%	50.15%
One Hop effectiveness	46.87%	73.70%	65.92%
One Hop Eager Correct	13.42%	48.57%	23.41%
One Hop Lazy Correct	82.38%	76.52%	71.63%
Network Percentage	99.99%	99.99%	99.97%

Table 4: Results of the most accurate simple model.

Since one of the models obtained performed in every relevant metric better than the other, only that model performance results is shown, using a simulation with all nodes using the model for predictions, averaging the metrics obtained from 5 different simulations with randomly generated topologies.

Table 4 shows the behaviour and performance results of the most accurate model of this iteration, comparing with the strategy used to generated the dataset, `Mixed 50% Simulation` and the corresponding strategy that uses the same amount of resources. **Eager Percentage** metric indicates the model considerably reduced the eager transmission rate, which is used in 9.66% of the situations as a way to increase the one hop effectiveness by the model. As a consequence the number of payloads attempts is only 335 and 332 from those are successfully sent. This translates to better usage of resources as the payload transmissions are reduced to approximately a quarter of the 1208 obtained in the mixed simulation.

Comparing the necessary hops to distribute messages, a considerable improvement is obtained. The model from the simple dataset reduced the average of hops by 1.66 to the

mixed simulation and by 2.58 to the flat 10% eager, needing only 2.13 hops to distribute messages. This indicates better paths to distribute messages are used. But the highest improvement is obtained in the necessary rounds to distribute messages where the average time needed reduces by 0.07 rounds while using a lot less resources. This result is also better if a comparison is made with the flat strategy that uses nearly the same percentage of eager transmissions, recurring to a **10% flat** strategy. In the 10% flat strategy the average time in rounds to distribute messages is 4.04 meaning that the model accelerated the distribution by 1.11 rounds.

Comparing the one hop effectiveness, a metric that evaluates the current iteration training process **ShouldEager** definition, we can conclude that the model has improved the accuracy achieving its goal. The model does not achieve the same level as the 88% accuracy of the training process but still increases the accuracy from 46.87% to 73.70%. It is important to highlight the gain is not only a consequence of lowering the eager transmission rate because the 10% flat obtained an accuracy of 65.92%, nearly 8% lower than the model's accuracy. The model's 8% accuracy improvement is from having more than double of the **eager one hop** accuracy and a 4.89% higher **lazy one hop** accuracy. Finally, the model obtains the same **network percentage** reach as the mixed simulation, achieving good reliability.

In general the model achieves improvements in all the desired metrics. As a mean to increase the effectiveness the model uses a strong conservative approach but comparing the message overflow rate to the 10% flat it has triple of the chance of losing a message by overflow. This means the paths where eager transmissions are sent is more preferential, having a small pool of connections selected to make the "emergent structure". This model obtained good performance improvements since it maintained the same level of network reach and increased the distribution speed and path while using less resources being a good conservative option for the protocol's distribution.

4.4.2 *Complex Model*

In this case the goal is to improve the baseline protocol performance by using historical information of what happened to the previous messages. The metrics include state but also the performance of the entire simulation, as described in Section 3.3. The models obtained from these datasets are referred to as **complex models**.

After the training process is over, the two most accurate models shown in Figure 16 are tested by using a model percentage of 100% of the simulator network, meaning all nodes use the model to make their decisions.

Table 5 compares the performance of the best model obtained, from the two tested, in this iteration with the **mixed 50% simulation** that generated the dataset and the corresponding strategy that uses the same level of resources, the **flat 24% eager**. The model once again opted

```
> for (i in 1:search_criteria$max_models) {
+   gbm <- h2o.getModel(sortedGrid@model_ids[[i]])
+   print(h2o.auc(h2o.performance(gbm, valid = TRUE)))
+ }
[1] 0.8694336
[1] 0.8690436
[1] 0.8647446
[1] 0.8609879
[1] 0.8602019
[1] 0.8568176
[1] 0.8557113
[1] 0.785137
[1] 0.7833692
```

Figure 16: Accuracy on the test dataset of the complex models during the training process.

Metric	Mixed 50% Simulation	Best Model	Flat 24% Eager
Message OverFlow	16.87%	10.91%	1.31%
Eager Percentage	50%	24.50%	24%
Payloads Attempts	1453.44	693.67	798.11
Successful Payloads Sent	1208.29	618.03	787.58
Failed Requested Payloads	22.65	14.63	4.29
Hops Average	3.79	2.33	4.61
Rounds to Distribute	3.00	3.65	3.41
Predicted effectiveness	50.00%	70.03%	50.44%
One Hop effectiveness	46.87%	75.56%	60.85%
One Hop Eager Correct	13.42%	27.79%	20.54%
One Hop Lazy Correct	82.38%	91.30%	75.64%
Network Percentage	99.99%	99.99%	99.98%

Table 5: Results obtained from the best complex model.

for a conservative approach to increase the **one hop effectiveness** but with a considerably higher percentage from the previous models, reaching an eager percentage of 24.50%. For that reason the number of payload transmission attempts reduced to 693.67 and only 618 of those are successfully sent. This means that the model uses nearly half the resources of the mixed 50% simulation that sends 1208 payloads per message created. The **message overflow** percentage is lower than the mixed simulation, but that is an expected consequence of a lower usage of eager transmissions. Comparing it with the corresponding **flat 24% eager** strategy, the percentage in which messages are lost by the model is considerably higher from 1.31% to 10.91% indicating a preferential and strict path to send payloads through.

Regarding the performance metrics, the model is able to considerably improve the path close to the diameter of the network. It reduces the necessary number of hops by 1.46 to the mixed simulation and by 2.28 to the corresponding flat strategy, with 24% eager percentage. But, contrary to expectation, the gain obtained in the **hops average** does not contribute to

Metric	Simple Model	Complex Model
Message OverFlow	0.96%	10.91%
Eager Percentage	9.66%	24.50 %
Payloads Attempts	335.64	693.67
Successful Payloads Sent	332.41	618.03
Failed Requested Payloads	0.47	14.63
Hops Average	2.12	2.33
Rounds to Distribute	2.93	3.65
Predicted effectiveness	57.74%	70.03%
One Hop effectiveness	73.70%	75.56%
One Hop Eager Correct	48.57%	27.49%
One Hop Lazy Correct	76.52%	91.30%
Network Percentage	99.99%	99.99%

Table 6: First iteration models comparison.

a faster distribution of payloads. Not only it does not improve the mixed simulation time taking more 65% of a round to finish the distribution in average the model also takes more time than the corresponding flat strategy, taking more 24% of a round.

Analysing metrics that show the accuracy of decisions, the model obtains the desired increase on the definition it was trained for. The **one hop effectiveness** increases from the 60.87% to 75.56% when comparing strategies that use the same percentage of eager transmissions. That gain is mostly obtained from the high accuracy of lazy decisions where the model rarely chooses wrongly for a lazy transmission. This iteration also increased the **predicted effectiveness** by 20% but it remains 5% lower than the one hop effectiveness.

Finally the model maintains the same level of reliability as the normal mixed simulation by reaching 99.99% of destinations.

4.4.3 *Models Comparison*

In both simple and complex versions of this iteration, the two best models show a conservative approach but with different behaviours and performances.

Table 6 allows comparing the accuracy and behaviour of the models obtained from the two types of dataset. The complex model is able to reach a higher **one hop** accuracy, almost 2% higher than the simple model accuracy but that didn't necessarily translate to a better performance. The simple model is able to obtain a lower number of hops needed to finish distribution and takes less 72% of a round duration to distribute messages while using less resources, saving about 285.62 payloads unnecessarily sent. Nodes rarely lose a message with both models, with a chance lower than 0.01%. This might indicate that basing the choice on having a higher accuracy on good decisions does not necessarily contribute to a better performance in the protocol since the complex model has a higher accuracy in the

```
> mtx = cor(num)
> drop = findCorrelation(mtx, cutoff = .8)
> drop = names(num)[drop]
> print(drop)
[1] "timePassed" "Id"
```

Figure 17: Variables that should be removed from the simple dataset version for having big pair-wise correlation.

simulation, but the simple model clearly has better performance even though the average number of hops is considerably lower in this approach.

4.5 Removing Highly Correlated Variables

The goal of the second iteration is to obtain better accuracy on the **ShouldEager**, and as a consequence optimize the message distribution, by removing highly correlated variables from the dataset.

This exploratory data analysis was based on methods where the removal of highly correlated variables is done with special caution to not remove the features needed, so it is possible to obtain more accurate results than the first iteration (4) (24).

4.5.1 *Simple Model*

For the simple dataset version, using a cut-off ratio of 0.8 in the `findCorrelationMethod` over all the numeric variables, the columns that should be removed to reduce the pair-wise correlations are **Id** and **TimePassed** as depicted in Figure 17.

Based on this information, those columns are removed from the dataset after the initial load and the same training process is done with the remaining variables, testing the two most accurate models in the simulator and comparing them with the flat strategy that used the same amount of resources, flat 8% eager.

Table 7 shows that the two most accurate models from this iteration considerably improve one hop effectiveness by nearly 5% when compared to the corresponding flat strategy. Even though the eager percentage is similar, the number of payload attempts and payloads successfully sent are considerably lower ending up with a better usage of resources. Message overflow is considerably higher, with the second model being 17 times higher than the flat 8% strategy which again indicates that there is a preferential attachment in links where eager transmissions are made.

Both models obtain improvements in the necessary hops to finish distribution lowering from 4.57 to 2.05 and 2.09 respectively. It is also lower than the previous model from the first iteration that had an average of 2.13 hops. Comparing the necessary time to distribute

Metric	Flat 8% eager	1st Model	2nd Model
Message OverFlow	0.27%	2.79%	4.77%
Eager Percentage	8%	6.90%	7.69%
Payloads Attempts	399.81	289.39	316.53
Successful Payloads Sent	398.72	281.30	301.44
Failed Requested Payloads	0.87	0.57	1.34
Hops Average	4.57	2.05	2.09
Rounds to Distribute	4.17	3.44	3.51
Predicted effectiveness	67.05%	55.65%	55.83%
One Hop effectiveness	67.05%	71.85%	71.89%
One Hop Eager Correct	23.41%	53.17%	47.69%
One Hop Lazy Correct	71.64%	73.30%	74.08%
Network Percentage	99.97%	99.9997%	99.9990%

Table 7: Results of the two most accurate models from the simple model without highly correlated variables.

```
> num = data %>% dplyr::select(where(is.numeric))
> mtx = cor(num)
> drop = findCorrelation(mtx, cutoff = .8)
> drop = names(num)[drop]
> print(drop)
[1] "fullyDistributedLazyImpact"    "connectionLostMessagesAverage" "hops"   "messageID"    "round"
```

Figure 18: Variables that should be removed from the complex dataset version for having big pair-wise correlation.

messages it is also considerably lower, 73% and 66% of a round duration respectively. Takes longer than the mixed simulation to distribute messages, but the eager percentage is considerably lower with fewer risks being taken.

Looking more in depth into the specific eager and lazy correct decisions it is possible to verify that the increase in the one hop accuracy from models is mainly due to the increase in the percentage of correct eager decisions, which is more than 2 times higher than the flat 8%. Most specifically the first model is able to achieve 53.17% of correct eager decisions.

Finally, comparing the network reach by messages both models increase considerably that percentage making the loss of a message very rare.

4.5.2 *Complex Model*

For the complex model that uses performance averages of the current simulation, the following variables were removed for having a correlativity value over 0.8 to other variables: **fullyDistributedLazyImpact**, **connectionLostMessagesAverage**, **hops**, **messageID** and **round**, as depicted in Figure 18.

Metric	**Mixed** 50% **Simulation**	**Best Model**	**Flat 24% Eager**
Message OverFlow	16.87%	9.85%	1.32%
Eager Percentage	50%	24.87%	24%
Payloads Attempts	1453.44	699.47	798.11
Successful Payloads Sent	1208.29	630.62	787.58
Failed Requested Payloads	22.65	12.56	4.29
Hops Average	3.79	2.21	4.61
Rounds to Distribute	3.00	3.55	3.41
Predicted effectiveness	50.00%	70.55%	50.44%
One Hop effectiveness	46.87%	75.46%	60.85%
One Hop Eager Correct	13.42%	23.30%	20.54%
One Hop Lazy Correct	82.38%	91.67%	75.64%
Network Percentage	99.99%	99.998%	99.990%

Table 8: Results of the best performing model from the complex model without highly correlated variables.

From the two most accurate models one performs significantly better than the other so only that model is shown in Table 8 comparing it with the **mixed 50% simulation** and the corresponding flat strategy, **flat 24% eager**. The model once again uses a conservative approach reducing the eager rate to 24.87% to achieve a higher one hop accuracy. This lowers the number of payloads sent per message created to almost half maintaining the link conditions in better state overall.

As seen before the model is able to optimize the distribution path by reducing the number of hops to 2.21. The model takes more 55% of a round duration to distribute messages than the mixed simulation, but as before that can be a consequence of using a more conservative approach. But, if a comparison is made with the corresponding 24% eager flat strategy the model is still not able to improve the distribution speed, taking more 14% of a round duration. Nevertheless, this model is one of the best models in their accuracy, almost achieving the best **one hop accuracy** yet but that did not translate into performance improvements besides reducing the overall hops average.

4.5.3 *Global Analysis of the Obtained Models*

In the first `ShouldEager` definition all models reduced their eager transmission rate to maximize the desired **one hop accuracy**. This prevents models to obtain significant improvements in the distribution time because lesser risks are taken and the randomness obtained from the message transmission times that can take between 40 to 60% of a round implies that the most optimal path is not the same every time and to obtain distribution time improvements closer to the full eager performance other good alternative paths need to be used. That

said, there is a need of more aggressive models that take more risks and increase their eager transmissions in a selective way.

Overall, the models from the second iteration obtained a better average **one hop accuracy**, so the variables removed did not hinder the training process but the best performing model is from the first iteration using the simple dataset.

Some tendencies can be noticed analysing the results obtained:

- Complex models obtain a slightly higher accuracy than simple models.
- Basing decisions on **one hop accuracy** translates to considerably reducing the hops average.
- One hop accuracy does not contribute directly to better distribution speeds since some models had higher distribution times than the corresponding flat strategy that used the same amount of resources while averaging less hops.
- This training process increases its accuracy with more selective eager decisions, taking less risks, and therefore it is hard to considerably improve the mixed simulation distribution time by using this **ShouldEager** definition.
- One hop accuracy is crucial to ensure the reliability of the protocol making it very rare to lose a message.

Since the results for the safe approach obtained satisfactory results, there should not be many variants able to achieve a considerable improvement from the first simple model that was able to use low resources and still improve the mixed simulation distribution speed and path. But with this approach no aggressive models were obtained where the distribution speed considerably improved and reached times close to the distribution speed of the full eager simulation by selectively increasing the eager rate and is the main focus of the next iteration.

4.6 Aggressive Models

In this iteration the goal is to see how the models that increase the chances of eager transmissions perform. A model with a more aggressive approach should lower the average time needed to distribute messages closer to the optimal time found in the full eager simulation while using less resources or obtain better distribution speed than the corresponding flat strategy. To simplify the process and since it is the version that obtained the better performance metrics in the simulator, the remaining evaluation will be made only using the simple dataset.

Metric	Flat 8% eager	1st Model	2nd Model
Message OverFlow	0.27%	2.83%	4.54%
Eager Percentage	8%	7%	7.48%
Payloads Attempts	399.81	290.74	312.31
Successful Payloads Sent	398.72	282.50	298.14
Failed Requested Payloads	0.87	0.59	1.10
Hops Average	4.57	2.04	2.12
Rounds to Distribute	4.17	3.36	3.52
Predicted effectiveness	67.05%	55.69%	55.59%
One Hop effectiveness	67.05%	72.06%	72.01%
One Hop Eager Correct	23.41%	53.19%	48.14%
One Hop Lazy Correct	71.64%	73.56%	74.11%
Network Percentage	99.97%	99.9986%	99.9974%

Table 9: Results of the two most accurate simple models using the aggressive hop accuracy.

4.6.1 *Aggressive One Hop Effectiveness*

In this version the goal is to see how the model behaves considering the first two unnecessary receptions of a payload as good tries. Those good tries are assigned with 1 in the **impact** label during simulations. Therefore, this requires a change in the **ShouldEager** definition of eager transmissions, to also consider an impact value of 1, as shown in Listing 4.3.

```
response <- "ShouldEager"

table(dataset$ShouldEager)

dataset$ShouldEager[dataset$decision == "TRUE" & dataset$impact >=1]<- 1
dataset$ShouldEager[dataset$decision == "TRUE" & dataset$impact < 1]<- 0
dataset$ShouldEager[dataset$decision == "FALSE" & dataset$impact == 1]<- 0
dataset$ShouldEager[dataset$decision == "FALSE" & dataset$impact != 1]<- 1
```

Listing 4.3: Defining the ShouldEager columns based on the one hop effectiveness considering good tries.

Table 9 shows the results from the two most accurate models. The **one hop accuracy** remains at the same level as the one hop accuracy of the second iteration but contrary to the intention, the models are still conservative with a low percentage of eager decisions, around 7%. Still, they improve the corresponding conservative strategy. The first model is significantly better achieving a lower average of hops and time to finish distribution. In fact, in this iteration a new optimal value is found for the hops average and when compared to the corresponding flat strategy the distribution time improves considerably, finishing distribution in less 81% of a round duration.

In general the model obtained is an option for a conservative protocol, but the goal of this iteration is not achieved.

Metric	Flat 30% Eager	1st Model	2nd Model
Message OverFlow	3.24%	7.31%	7.26%
Eager Percentage	30%	29.43%	29.66%
Payloads Attempts	948.26	872.88	880.54
Successful Payloads Sent	917.55	809.04	816.59
Failed Requested Payloads	8.13	61.76	62.50
Hops Average	4.37	2.79	2.85
Rounds to Distribute	3.29	4.03	3.82
Predicted effectiveness	50.79%	80.33	80.22%
One Hop effectiveness	58.44%	70.09%	69.35%
One Hop Eager Correct	18.71%	23.85%	23.67%
One Hop Lazy Correct	77.84	89.96%	89.20%
Network Percentage	99.99%	99.99%	99.99%

Table 10: Results of the most accurate model using the predicted effectiveness training process.

4.6.2 *Predicted Effectiveness*

As another attempt to make a more aggressive use of eager transmissions, the next test bases the training process on the **Useful** column that contains the predicted effectiveness label of the decision.

```
response <- "ShouldEager"

table(dataset$ShouldEager)

dataset$ShouldEager[dataset$decision == "TRUE" & dataset$useful == "TRUE"]<- 1
dataset$ShouldEager[dataset$decision == "TRUE" & dataset$useful == "FALSE"]<- 0
dataset$ShouldEager[dataset$decision == "FALSE" & dataset$useful == "TRUE"]<- 0
dataset$ShouldEager[dataset$decision == "FALSE" & dataset$useful == "FALSE"]<- 1
dataset$ShouldEager <- as.factor(dataset$ShouldEager)
```

Listing 4.4: Defining the ShouldEager columns based on the predicted effectiveness in R.

With the process shown in Listing 4.4 the rows assign the **ShouldEager** with **1** when:

- The eager transmission is chosen and at the moment the message is sent the receiver still didn't have the message in their published messages.
- The lazy transmission is chosen and at the moment the message is sent the receiver already had the message published.

In the remaining situations the **ShouldEager** is 0, meaning that transmissions should be lazy.

As intended models in this iteration have a more aggressive approach than the previous ones but still are more conservative than the mixed simulation. Results obtained in this iteration with the simple dataset are shown in Table 10.

```
> for (i in 1:search_criteria$max_models) {
+   gbm <- h2o.getModel(sortedGrid@model_ids[[i]])
+   print(h2o.auc(h2o.performance(gbm, valid = TRUE)))
+ }
[1] 0.7731966
[1] 0.7702136
[1] 0.7690006
[1] 0.7650205
[1] 0.7641419
[1] 0.7597828
[1] 0.7576763
```

Figure 19: Accuracy on the test dataset using the combined approach.

The most accurate models obtained in this iteration had similar behaviours, even in the specific accuracy of eager and lazy decisions. Overall, the models performance metrics are not better than the corresponding flat strategy. Even though the models obtain an improvement in hops needed to distribute messages, the distribution gets slower.

Looking specifically into the accuracy of the `ShouldEager` definition, predicted effectiveness, both models obtain 80% accuracy, not far away from the 87% accuracy obtained during the training process, so they behave as intended. But once again, even though this definition is able to obtain high accuracies with a higher rate of eager transmissions it did not translate into faster distribution times which makes the models worse than the ones obtained before. This cost in time is very likely a consequence from the overhead of a high number of failed requested payloads.

4.6.3 *Combined Approach*

In the attempt to get the desired aggressive model a new version is tested that combines the previous approaches. Here eager transmissions are considered as good attempts if the payload was received up to 2 times. Lazy transmission is preferred after the third reception of the same payload. A preference for eager is also obtained by using the predicted effectiveness metric on lazy transmissions as shown in Listing 4.5.

```
response <- "ShouldEager"

table(dataset$ShouldEager)

dataset$ShouldEager[dataset$decision == "TRUE" & dataset$impact >=1]<- 1
dataset$ShouldEager[dataset$decision == "TRUE" & dataset$impact < 1]<- 0
dataset$ShouldEager[dataset$decision == "FALSE" & dataset$useful == "TRUE"]<- 0
dataset$ShouldEager[dataset$decision == "FALSE" & dataset$useful == "FALSE"]<- 1
```

Listing 4.5: Defining the ShouldEager columns based on a combined approach.

Metric	Mixed 50% Simulation	1st Model	2nd Model
Message OverFlow	16.87%	40.72%	36.63%
Eager Percentage	50%	62.10%	45.99%
Payloads Attempts	1453.44	1662.83	1252.21
Successful Payloads Sent	1208.29	985.66	793.53
Failed Requested Payloads	22.65	0.36	0.82
Hops Average	3.79	2.88	2.80
Rounds to Distribute	3.00	2.32	2.75
Predicted effectiveness	50.00%	58.37%	58.98%
One Hop effectiveness	46.87%	44.87%	56.82%
One Hop Eager Correct	13.42%	11.74%	15.38%
One Hop Lazy Correct	82.38%	99.22%	92.35%
Network Percentage	99.99%	99.70%	99.91%

Table 11: Results of models from the combined approach using the simple dataset.

The two most accurate models listed in Figure 19 are tested and their results shown in Table 11.

Eager Percentage indicates that the first model leads to more eager transmissions, 62.1%, while the other slightly reduces the eager transmission rate from the 50% to 46%. Therefore, the first model has the desired aggressive approach and the second model a small conservative approach but still near the same usage of resources of the mixed simulation. As a consequence the number of payload attempts is higher in the first model by nearly 210 more payloads per message and nearly less 200 payloads in the second one. Even though the number of attempts is higher on the first model, the number of successful payloads sent is 223 less than in the common protocol. This means the channels in which the eager transmission is done is way more preferential and ends up costing less bandwidth of the network since failed messages don't waste bandwidth. The same can be said for the second model that only sends 793 payloads per message, which saves 415 payloads successful emissions per message when compared to the mixed version using more efficiently resources.

Comparing the **necessary hops** to distribute messages, both get similar improvements, being the second model better by 0.08 hops on average. On the other hand the first model takes a lot less time to finish distribution when compared to the second model, but both improve the mixed version. The first model has a lot more payloads attempted so there can be more diversified paths selected to send the payload. Having more optimal paths attempted can explain the improvement on the number of rounds to distribute the payloads because, under the configuration used, each message takes a random time to finish the delivery between 40% and 60% and one of these paths might on each occasion reach first to a node than the other one by a considerable amount of time. The first model is only 0.3 rounds away from the optimal distribution time obtained with full eager simulations.

The **effectiveness metrics** get an odd result on the first model, but that explains why previous models were not able to improve the distribution speed until now. While the second model gets an improvement of almost 10% of their decisions in the one hop effectiveness, the first one ended up having a lower hit rate of 2% compared to the mixed version. But with a closer look to the specific eager and lazy one hop effectiveness the behaviour can be analysed. While the first model has a lower hit rate on the eager transmission, which can also be expected since the eager transmission is chosen more frequently, whenever it chooses for a lazy transmission it is the correct decision almost every time, reaching a hit rate of 99.2%. This implies that rarely any notification is received of a message that a node does not have and therefore few requests are made, avoiding wasting time with this procedure. On the other hand, in the second model eager transmissions are more selective which means it has a higher correct rate, but a lot more notifications are sent lowering the lazy correct ratio and taking longer to finish distribution.

In general, both models improve the performance on the simple dataset. Both have a more preferential approach to assign links for eager transmissions with the first model having a wider range of links where it sends the payloads through. Both reduce the number of hops to deliver a message to the network by nearly one hop, having discovered more sub-optimal paths to send the payloads than on a normal mixed execution. The necessary time to distribute the messages is where there is a considerable disparity between the two models, where the more aggressive approach of the first model ends up having a faster distribution, at the cost of sending more payloads while the second model gets a smaller improvement while lowering the number of payloads sent maintaining the network more stable. Message overflow is higher in both models, but that is a consequence of choosing the same links for eager transmissions. The **network percentage** of the second version of the model gets a network reach of 99.91% but in the first model the network percentage is relatively lower, losing 0.3% of the expected destinations. Taking these facts into account both models could be chosen to improve the protocol depending on the use case. If having a reliable protocol is crucial the previous conservative models could be chosen, but if a node not receiving a message occasionally has no big impact a faster distribution can be obtained with one of these models.

4.7 Evaluation of Effectiveness in Different Environments

Considering that the environment changes, there are two main factors that impact how the model is able to improve the gossip protocol performance:

- Available information of what is happening in the peer-to-peer network.
- Existence of super-nodes that facilitate spreading the payload.

With the purpose of understanding if the ML model can still improve the protocol performance in different and challenging scenarios, a final evaluation is done when these two factors come into play.

The first test limits the availability of information to one hop range, their **neighbours**, direct neighbours. The goal is to understand how many nodes are needed to collect data for the training in order for the model to improve by a good margin the performance.

The second test turns the decision-making of spreading the messages harder since it uses the **WireKOut** topology generator where all nodes have the same fan-out. With this topology there are no super-nodes making the only factor for the distribution decision the link state, turning the optimal spreading path of messages less obvious and more challenging.

4.7.1 *Limited Information*

The goal is to find the minimum percentage of randomly chosen nodes needed to observe and generate a dataset for the training process, so models can be effective and capable of optimizing protocol performance. This test represents a scientific necessity because it allows quantifying the percentage of nodes as a precondition to applicability in real cases.

For that purpose, using the simple dataset version, a percentage of nodes are randomly selected during the training process and only rows from nodes of that subset are used in the training process.

```
randomizeNodes <-sample(200, size = floor(0.4*200))
dataset<- dataset[(is.element(dataset$nodeFrom,randomizeNodes)), ]
```

Listing 4.6: Obtaining rows from the subset selected, following a percentage of 40%.

In Listing 4.6 an example is shown of how the dataset is obtained using only 40% of the nodes. The **sample** function reorders the nodes randomly and then obtains from that re-ordered list the initial 40% of nodes, that in this case corresponds to the first 80 nodes that are saved into a list named **randomizeNodes**. From that a filter is used in the dataset data frame where only the rows that have a **nodeFrom** value contained in the selected nodes set remain for the training process.

For the first iteration three different models were built using progressive subsets of nodes containing 10, 20 and 40% of the entire network to compare how they perform against the model obtained from the full simple dataset. The dataset didn't have highly correlated variables similarly to the process on Section 4.5.1, but using the **ShouldEager** definition of the combined approach. All the models obtained have a small conservative approach which facilitates the comparison and allows to understand the point where performance declines.

As can be seen in Table 12, **one hop effectiveness** tends to reduce with the absence of information but not by a considerable amount. All models are able to perform in a very similar level so the percentage considered for the comparison is still too big to prevent the

Metric	100%	40%	20%	10%
Message OverFlow	34.91%	35.48%	36.69%	36.17%
Eager Percentage	33.10%	33.14%	33.71%	34.15%
Failed Requested Payloads	0.51	0.46	0.51	0.60
Hops Average	2.61	2.61	2.60	2.60
Rounds to Distribute	2.83	2.86	2.91	2.86
Predicted effectiveness	58.81%	57.47%	57.12%	57.45%
One Hop effectiveness	64.78%	63.59%	62.77%	63.30%
Network Percentage	99.98%	99.98%	99.95%	99.97%

Table 12: Results comparison between the model using the entire dataset to subsets that only contain a percentage of the selected nodes.

Percentage	Nodes Reached
10%	150
20%	189
40%	198

Table 13: Neighbours reached per percentage selected from the dataset.

model from optimizing the distribution. That is a good thing since it suggests that the percentage of nodes needed for observation is lower than 10% making it easier to deploy a similar solution on a real life context.

Analysing the number of nodes reached from the percentage considered we obtain the following results: Table 13 explains the reason why the models obtained are not significantly affected by the limited information used. With 10% of nodes observing, the dataset still contains information of 150 nodes, since each node selected has knowledge of **one hop** of their neighbourhood. This translates for having wide information of 75% of the network, even though the dataset was only built from the initial 10% nodes of the network.

A deeper analysis of Figure 20 shows how many neighbours are reached with smaller percentages. With only 1% of nodes used for the dataset, 60 unique neighbours are reached meanwhile with 3% of selected nodes the dataset contains information about half of the nodes present in the network. For this reason two more models are tested using only 1% and 5% respectively, as shown in Table 14.

In this case the break in performance is evident, where the one hop accuracy decreases 5.28% and 8.63% respectively. This means the model does not achieve its purpose. Specifically, the model that uses 5% of nodes distributes the payload slower and uses the eager transmission in 2.31% more cases which is considerably worse. The model that used only 1% of nodes for training, even though it still has a conservative approach uses a lot more eager transmissions, affecting the path in order to distribute faster, but at a cost of losing a lot more messages. The main conclusion obtained here is that the model performance declines considerably using 5% of the nodes, so the breakpoint is between the 10% and 5%.

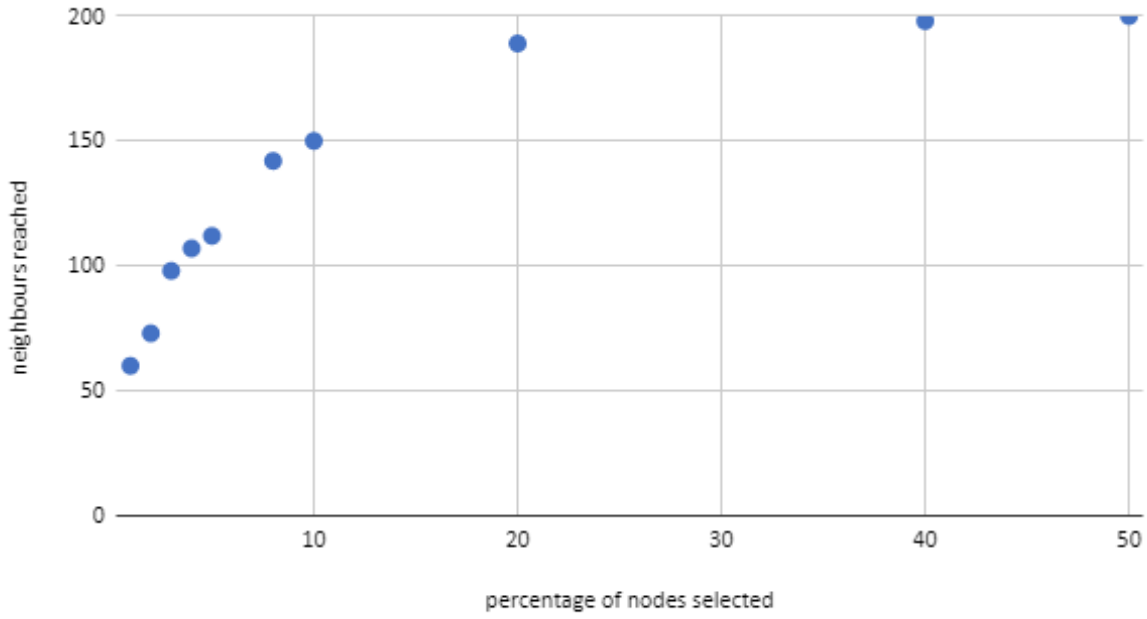


Figure 20: Neighbours reached per nodes percentage.

4.7.2 *Homogeneous Network*

This test evaluates the performance of the protocol with a homogeneous network, where every node has the same fan-out and, consequently, a quick message dissemination is harder to reach.

For this test the configuration changes from the previous one in the topology generator method. Instead of having a degree distribution like the one obtained in Figure 7, all the nodes have the same number of neighbours, 7. The dataset type used for the model is the simple version.

Since the topology is different, new protocol limits need to be obtained as baselines. In a topology with fewer links, the diameter is higher and the previous performance limits obtained before in Section 4.2 become unrealistic since payloads take longer to be disseminated.

Table 15 highlights differences from the limits obtained in Table 2. Comparing the full eager simulations, one expected behaviour is the slower distribution speed, taking almost more 88% of a round duration. The reason for this is the existence of fewer options where payloads can be sent, increasing the diameter of the network. This is reinforced by the **payload attempts** metric where the number of attempts is nearly half of the other topology. The number of nodes that don't receive messages in this topology is also way higher which means losing a message is more costly.

Metric	100%	5%	1%
Message OverFlow	34.91%	36.66%	34.99%
Eager Percentage	33.10%	35.41%	43.64%
Failed Requested Payloads	0.51	0.53	0.95
Hops Average	2.61	2.60	2.75
Rounds to Distribute	2.83	2.89	2.71
Predicted effectiveness	58.81%	55.09%	56.79%
One Hop effectiveness	64.78%	59.50%	56.15%
Network Percentage	99.98%	99.96%	99.92%

Table 14: Results comparison between the model using the entire dataset and subsets containing only 1 and 5 percent of nodes.

Metric	Full Eager	Full Lazy	Mixed 50%
Message OverFlow	49.59%	1.23%	21.06%
Payloads Attempts	1337.22	200.93	808.97
Successful Payloads Sent	674.11	198.46	638.60
Failed Requested Payloads	0	2.48	31.01
Hops Average	4.83	4.35	6.06
Rounds to Distribute	2.88	6.67	4.85
Predicted effectiveness	25.81%	50.25%	50.00%
One Hop effectiveness	14.21%	69.59%	44.67%
Network Percentage	95.93%	99.09%	98.33%

Table 15: New performance limits on a homogeneous network.

When comparing the full lazy simulations, the message overflow is considerably higher, from 0.04% to 1.23% and the main reason for that is that it sends the same amount of payloads, **Successful Payloads Sent**, while only having half the links. This means that connections have a higher message load to transmit and end up losing messages more often. It also takes considerably more time to spread the messages because of the higher diameter. Finally, the chance of a node not receiving a message is considerably higher, achieving the same **network percentage** of the previous full eager simulation, due to the homogeneous network style of the topology.

Finally, analysing the mixed gossip simulation the most curious fact is having almost the same average of payloads successfully sent as the full eager simulation, only sending less 35.5 payloads. As the other version, the mixed simulation has a distribution average time in between the distribution times of full eager and full lazy simulations. It also has a worse distribution path, needing in average more hops to finish distribution. The results of this column are the ones the model tries to improve from, in Table 16.

The model, obtained using the simple dataset version on a homogeneous network following the combined approach of Listing 4.5, considerably improves the performance. **One hop accuracy** is 5.4% higher. On average, the number of payloads successfully sent reduced

Metric	Model
Message OverFlow	35.62%
Eager Percentage	52.66%
Payloads Attempts	773.84
Successful Payloads Sent	498.18
Failed Requested Payloads	1.54
Hops Average	4.05
Rounds to Distribute	3.90
One Hop effectiveness	50.07%
Predicted effectiveness	51.10%
Network Percentage	98.71%

Table 16: Performance of the model on a homogeneous network.

from 638 to 498, overall leaving the network links on better conditions. It is also able to find a better distribution path reducing the necessary hops by 2 and distributes in less 95% of a round duration. One crucial factor for this improvement is the reduced number of **Failed Requested Payloads** that manages to be less than the value obtained in the full lazy simulation and is also far inferior to the mixed simulation. Finally, the protocol is slightly more aggressive than the mixed simulation, increasing the eager percentage by 2.66% but the **network percentage** is higher which indicates a better usage of the network, close to the value obtained from the full lazy simulation.

In general, the model improves all relevant metrics, achieving satisfactory results taking into account the information contained in the simple dataset version used and the network with the absence of super-nodes. It manages to get a considerable distribution speed optimization, reduced the number of hops average and has an interesting network percentage, close to the possible optimal value of the full lazy simulation. This means decisions can still be made based only on the current link state to optimize the protocol.

These results lead to the conclusion that the current state of connections is a determining factor in decisions to predict the correct strategy in each occasion, but the existence of super-nodes facilitates the message dissemination and makes decisions more obvious.

5

CONCLUSION AND FUTURE WORK

This work aims at discovering if ML technologies can improve epidemic multicast protocols. To this end we propose a proof of concept implementation that uses an ML model to make decisions regarding message dissemination in epidemic multicast protocols to achieve a better trade-off between the bandwidth used and required time to distribute messages.

The simulator implementation is modular and configurable, regarding: network topology, link capacity, payload size, frequency of message creation, and transmission strategy. The leaky bucket algorithm is used to simulate the bandwidth limits and dropped messages (7). The implementation also includes the observation of metrics and backtracking feedback for dataset generation. Metrics are collected by executing simulations of random realistic P2P topologies and used to train ML models. The goal is to get models to find protocol characteristics from the dataset to help nodes decide on transmission type in each occasion. Several models are produced in three different iterations by varying the information used and the definition of what is considered to be a correct decision.

The resulting models significantly optimize epidemic diffusion. Compared to the base flat protocol, all models obtain a lower average of hops, but the number of payloads sent and corresponding distribution time depend on the approach used and on the behaviour of the model. Message dissemination is controlled by ML models that choose paths in which messages should be sent eagerly, creating an emergent structure based on predictions factoring in the environment. One hop effectiveness is higher on conservative models where eager transmissions are used more carefully. In contrast, the best aggressive models have a higher predicted effectiveness. Conservative models produce several options with good performances where the best comes from the simple version in the first iteration, using less than 10% of eager transmissions but still able to reduce the distribution time of the mixed simulation while ensuring reliable dissemination. An aggressive model built with the simple dataset achieves a dissemination time of 2.32 rounds, just 0.3 rounds from the optimal full eager simulation speed but increasing the chance of a node not receiving a message compared to other models.

The main conclusions are the following: Our models obtained considerable improvements in relevant performance metrics, confirming that it is possible to improve the epidemic

multicast protocol using machine learning; a faster distribution speed is obtained from the models by creating an emergent eager structure basing the decisions on the actual state of links, as suggested in (30); **Should Eager** definition relates to the behavioural approach of the model where faster distributions have a higher **predicted accuracy** and conservative models tend to have a higher **one hop accuracy**, requiring less hops but the transmission being slower; to improve distribution speed, sub-optimal paths need to be taken due to the configuration used where messages take a random time from 40% to 60% of a round to be delivered; models using simple dataset perform better than the complex dataset meaning that less observation is required for optimization.

The main challenge encountered was how to avoid exceeding the memory limit and delaying execution time by using the inference model for simulations with some complexity. This was solved by assigning a global variable for the model and, on each occasion nodes would access the model for their prediction. Another challenge was defining a methodology where the correct decision assignment is able to improve the protocol performances, since that is very subjective depending on the goal and the circumstance the protocol is used and sometimes the model's behaviour was the opposite of the initial intention of the methodology chosen. Specifically, it was challenging to obtain an aggressive model where the distribution time considerably improved the mixed simulation performance. This was done by using different **ShouldEager** definitions on each iteration and the aggressive model came from a combined approach of definitions that benefit eager transmissions.

To understand the usability of this approach tests were done to determine the minimum necessary percentage of the network to build a dataset for models to be useful concluding that 5% to 10% of the nodes in the network are needed. Moreover, we evaluated the model effect on a homogeneous network, where only link state information is available. This allowed to conclude that the model is still able to improve the performance but at a smaller scale in the distribution speed and average hops.

As **future work**, there are some things that should be considered. First, even though the obtained models perform considerably better than the baseline, some better trade-offs are possible using different subsets of information or changing some parameters used in the training process. The main focus would be choosing a smaller subset of information of the complex dataset that could help to outperform the simple dataset. In line with this idea other definition of **ShouldEager** could be tested, to obtain different behaviours. An example for this could be similarly to to the **first N** payload receptions, consider payload receptions done **under a considerable time interval** from the first payload reception as a way to compensate eager transmissions that were really close to being a correct decision.

Since the simulator can have complete and perfect information of what is going on and the impact metric can serve as a reward tool for decisions, it is interesting to test how the

protocol does by using RL and evaluate if it can improve performance. On the same note other ML algorithms could be used.

BIBLIOGRAPHY

[1] A Comprehensive Guide to Convolutional Neural Networks. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5- Consulted in 10/03/2021.

[2] About Train, Validation and Test Sets in Machine Learning. https://towardsdatascience.com/train-validation-and-test-sets-72cb40cba9e7. Consulted in 10/12/2020.

[3] An introduction to Reinforcement Learning. https://www.freecodecamp.org/news/an-introduction-to-reinforcement-learning-4339519de419/. Consulted in 18/12/2019.

[4] Are you dropping too many correlated features? An analysis of current methods and a proposed solution. https://towardsdatascience.com/are-you-dropping-too-many-correlated-features-d1c96654abe6. Consulted in 15/12/2020.

[5] CIW: Trials, Warm-up Cool-down . https://ciw.readthedocs.io/en/latest/Tutorial-I/tutorial_iv.html. Consulted in 10/12/2020.

[6] Comparation of Learning Styles. https://medium.com/@chisoftware/supervised-vs-unsupervised-machine-learning-7f26118d5ee6.

[7] Computer Network | Leaky bucket algorithm. https://www.geeksforgeeks.org/leaky-bucket-algorithm/. Consulted in 30/01/2021.

[8] Deep Learning Vs Neural Networks - What's The Difference? https://bernardmarr.com/default.asp?contentID=1789. Consulted in 18/12/2019.

[9] Gradient Boosting Machine (GBM) - H2O documentation . http://docs.h2o.ai/h2o/latest-stable/h2o-docs/data-science/gbm.html. Consulted in 10/01/2021.

[10] H2O.ai - open source leader in AI. https://www.h2o.ai/. Consulted in 25/01/2020.

[11] How to build a machine learning model. https://searchenterpriseai.techtarget.com/feature/How-to-build-a-machine-learning-model-in-7-steps. Consulted in 30/01/2021.

[12] Hub-and-Spoke Topology. https://transportgeography.org/?page_id=653. Consulted in 10/12/2019.

[13] IBM - What is Supervised Learning? https://www.ibm.com/cloud/learn/supervised-learning. Consulted in 15/12/2020.

[14] Investopedia: Deep Learning. https://www.investopedia.com/terms/d/deep-learning.asp. Consulted in 15/01/2020.

[15] Machine Learning - Why is matters. https://www.sas.com/en_us/insights/analytics/machine-learning.html. Consulted in 28/01/2020.

[16] Machine Learning Concepts. https://www.expertsystem.com/machine-learning-definition/. Consulted in 10/10/2019.

[17] Machine learning explained: Understanding supervised, unsupervised, and reinforcement learning. https://bigdata-madesimple.com/machine-learning-explained-understanding-supervised-unsupervised-and-reinforcement-lear Consulted in 02/12/2019.

[18] NetLogo Models Library: Sample Models - Small Worlds. http://ccl.northwestern.edu/netlogo/models/SmallWorlds. Consulted in 04/01/2021.

[19] Peersim: A Peer-to-Peer Simulator. http://peersim.sourceforge.net/. Consulted em 29/09/2019.

[20] Real-Life and Business Applications of Neural Networks. https://www.smartsheet.com/neural-network-applications. Consulted in 13/01/2021.

[21] Simulation Practice - Ensuring Good practice. https://ciw.readthedocs.io/en/latest/Background/simulationpractice.html#simulation-practice. Consulted in 04/01/2021.

[22] The gossip epidemic and other issues in polite society. https://ayende.com/blog/169441/gossip-much-the-gossip-epidemic-and-other-issues-in-polite-society. Consulted in 10/12/2019.

[23] Understanding Gradient Boosting Machines . https://towardsdatascience.com/understanding-gradient-boosting-machines-9be756fe76ab. Consulted in 10/01/2021.

[24] Using R for Exploratory Data Analysis (EDA) . https://towardsdatascience.com/using-r-for-exploratory-data-analysis-eda-analyzing-golf-stats-812b5feb077a. Consulted in 15/12/2020.

[25] What is Deep Learning? Why is it Important? https://deepai.org/machine-learning-glossary-and-terms/deep-learning. Consulted in 20/10/2020.

[26] What is Gradient Boosting? . https://deepai.org/machine-learning-glossary-and-terms/gradient-boosting.

[27] What is Machine Learning? A Definition. . https://www.expert.ai/blog/machine-learning-definition/. Consulted em 29/10/2019.

[28] E. Michael Azoff. *Neural Network Time Series Forecasting of Financial Markets*. John Wiley Sons, Inc., USA, 1st edition, 1994.

[29] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.

[30] N. Carvalho, J. Pereira, R. Oliveira, and L. Rodrigues. Emergent structure in unstructured epidemic multicast. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 481–490, June 2007.

[31] Paulo Cortez and José Neves. Redes neuronais artificiais. 2000.

[32] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, USA, 1st edition, 1994.

[33] Geoffrey E Hinton. Deep belief networks. *Scholarpedia*, 4(5):5947, 2009.

[34] A. K. Jain, Jianchang Mao, and K. M. Mohiuddin. Artificial neural networks: a tutorial. *Computer*, 29(3):31–44, March 1996.

[35] J. Leitao, J. Pereira, and L. Rodrigues. Epidemic broadcast trees. In *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*, pages 301–310, Oct 2007.

[36] Ruowen Liu, Porter Beus, Steven Madler, and Bradley Bush. Analysis of watts-strogatz networks. *Arizona State University*, 2015.

[37] Dan W. Patterson. *Artificial Neural Networks: Theory and Applications*. Prentice Hall PTR, USA, 1st edition, 1998.

[38] Ruslan Salakhutdinov and Geoffrey Hinton. Deep boltzmann machines. In *Artificial intelligence and statistics*, pages 448–455. PMLR, 2009.

A

COMPLEX DATASET CALCULATIONS

For the fully distributed metrics, only the messages that stopped being spread in the network are considered. Whenever the controller NewRoundQueuer finds out a set of messages that were fully distributed, it communicates that set of messages by their identifiers for nodes to start processing and updating their values. With those sets, every node checks the corresponding events with the finalized impact and based on the impact they update the necessary counters. For each row, the node, depending on whether it is a lazy or eager transmission, increments the corresponding type of transmission impact counter by the number of times the row field **OverallImpact** indicates and increments by one the corresponding counter of occasions that type of transmission has occurred. Finally, to be able to obtain the connection impact averages, it also increments the corresponding connection impact and situations counter. Listing A.1 shows the necessary data structure to obtain the average impact of eager and lazy transmissions of each node.

```
public AtomicLong eagerFullyImpactCounter;
public AtomicLong eagerFullySituations;
public Map <Long, AtomicLong>  eagerFullyImpactPerNode;
public Map <Long, AtomicLong>  eagerFullySituationsPerNode;
public AtomicLong lazyFullyImpactCounter;
public AtomicLong lazyFullySituations;
public Map <Long, AtomicLong>  lazyFullyImpactPerNode;
public Map <Long, AtomicLong>  lazyFullySituationsPerNode;
```

Listing A.1: Necessary data structure to obtain impact averages of simulation.

To calculate the impact obtained from fully distributed messages it divides the corresponding total impact value by their respective type situations counter. For the connection impact, before making the calculation the node needs to obtain the corresponding counter matching the neighbour identifier as the key for the `HashMaps` shown in the previous data structures as shown in Listing A.2.

```
public double getFullyEagerConnectionImpact(long destinationID){
        if(this.eagerFullySituationsPerNode.containsKey(destinationID))
            if(!this.eagerFullyImpactPerNode.containsKey(destinationID))
                return 0;
            else return (double) this.eagerFullyImpactPerNode.get(destinationID).get() /
                (double) this.eagerFullySituationsPerNode.get(destinationID).get();
```

```
        else return 1;
    }
```

Listing A.2: Example method that obtains the Fully Distributed Eager Connection Impact metric.

The fully distributed metrics show the global simulation impact averages for the specific node to all their neighbours and specific to only the intended connection. Those metrics don't give recency feedback since it only updates after the messages stop circulating through the network. Those metrics are evaluated at the time the receiver gets the message and give feedback of recently sent messages. The logic behind the calculation this time is different, since it only considers the last **N** occasions and by default it is set for 20 messages. For these metrics there is a list of `Boolean` values for the node and another list for every connection on that node, always limited in size by **N**. Whenever feedback from a message transmission is received, it adds to the start of the list and the connection list the corresponding impact (true for positive impact and false for negative impact) and if the list has more than N values, removes the last value of the list that represents the oldest value. They use the data structure shown in Listing A.3.

```
public List<Boolean> eagerRecentImpact;
public List<Boolean> lazyRecentImpact;
public Map <Long, List<Boolean>> eagerRecentImpactPerNode;
public Map <Long, List<Boolean>> lazyRecentImpactPerNode;
```

Listing A.3: Data structure used for observation of the impact of recent messages.

With only those counters nodes can obtain the **lastMessagesEagerMean**, **lastMessages-LazyMean**, **lastMessagesEagerConnectionMean** and **lastMessagesLazyConnectionMean** metrics. They are calculated dividing the number of true values in the corresponding list by its size like the example of Listing A.4.

```
public double getLastMessagesEagerEffective(){
        if(this.eagerRecentImpact.size()>0){
            int aggregate=0;
            for(Boolean wasEffective: eagerRecentImpact)
                if(wasEffective)
                    aggregate++;
            return (double) aggregate / (double) eagerRecentImpact.size();
        }
        else return 0.5;
    }
```

Listing A.4: Example method that obtains the lastMessagesEagerMean metric.

To evaluate the connection availability and state of links during the whole simulation and in recent times four metrics are used: **connectionLostMessagesAverage**, **connection-LastNlostMessages totalLostMessagesAverage** and **lastNlostMessagesAverage**. Whenever a payload is sent, he can directly check if the message was sent or lost by overflow by reading the link state and the payload size. For the metrics that concern the whole simulation the

calculation is based on the number of lost messages and payload transmission attempts. Meanwhile, for the metrics that consider only the previous **N** transmissions, it is used once again a list of `Boolean` values to determine the percentage of recent payload transmissions that were lost, that being for the entire neighbourhood or specific to a connection. Listing A.5 shows the necessary data structure for those four metrics.

```
public AtomicLong personalMessagesCounter ;
public Map <Long, AtomicLong> messagesPerNodeCounters;
public AtomicLong personalMessagesLostCounter ;
public Map<Long, AtomicLong> messagesLostPerNodeCounters;
public List<Boolean> personalLastNWereLost;
public Map<Long, List<Boolean>> personalLastNWereLostPerNodeCounters;
public List<Boolean> personalLastNWereLost;
public Map<Long, List<Boolean>> personalLastNWereLostPerNodeCounters;
```

Listing A.5: Necessary data structure for the observation of lost payload transmissions.

The metrics that concern the whole simulation divides the total number of messages lost of that node by the number of payloads sent, as shown in Listing A.6. Equally, to the connection specific percentage, it also needs to use the neighbour identifier as a key for the `Hashmaps` to obtain the corresponding number of payloads sent and payloads lost in that link. The metrics that concern the previous **N** transmissions calculate the percentage dividing the number of true values in the list by its size.

```
public double getTotalLostMessagesAverage(){
    if(personalMessagesCounter.get()>0)
        return (double) this.personalMessagesLostCounter.get()/ (double)
            this.personalMessagesCounter.get();
    else return 0;
    }

public double getConnectionLastNLostMessagesAverage(long destinationId) {
    double result = 0;
    int aggregate = 0;
    ArrayList<Boolean> lost = null;
    if (this.personalLastNWereLostPerNodeCounters.
    containsKey(destinationId) ) {
        lost = (ArrayList<Boolean>)
            this.personalLastNWereLostPerNodeCounters.get(destinationId);
        if (lost.size() > 0) {
            for (Boolean wasLost : lost)
                if (wasLost)
                    aggregate++;
                result = (double) aggregate / (double) lost.size();
        }
    }
    return result;
}
```

Listing A.6: Example method that obtains the value for the totalLostMessagesAverage metric.

Finally, to obtain the metrics that give some insight in how obvious it is to make the right choice in that node some personal counter for that purpose, shown in Listing A.7.

```
public AtomicLong personalDecisionsCounter ;
public Map<Long, AtomicLong> decisionsPerNodeCounters;
public AtomicLong personalDecisionsCorrectCounter ;
public Map<Long, AtomicLong> decisionsCorrectPerNodeCounters;
public List<Boolean> personalLastNWereCorrectDecision;
public Map<Long, List<Boolean>> personalLastNwereCorrectPerNodeCounters;
```

Listing A.7: Necessary data structure for the observation of correct decisions.

The calculations for the percentage of correct decision remains the same. The node divides the number of times he was right by the times he made the decision, similarly to the other examples.

B

TRAINING PROCESS

B.1 DATASET SPLIT

```
1  smp_size <- floor(0.7 * nrow(dataset))
2  train_ind <- sample(seq_len(nrow(dataset)), size = smp_size)
3
4  train <- dataset[train_ind, ]
5  test_alpha <- dataset[-train_ind, ]
6
7  smp_size_test <- floor(0.50 * nrow(test_alpha))
8  test_ind <- sample(seq_len(nrow(test_alpha)), size = smp_size_test)
9
10 valid <- test_alpha[test_ind, ]
11 test <- test_alpha[-test_ind, ]
12
13 dim(train)
14 dim(valid)
15 dim(test)
16 dim(train)+dim(test)+dim(valid)
17 dim(dataset)
```

Listing B.1: Splitting the dataset into three different sets in R.

Listing B.1 shows how the split of the dataset is done in R, in which 70% of the dataset goes for the train variable and the remaining 30% equally divided to the variables valid and test from line 1 to 11. The remaining lines check if the valid and test sets have the same dimension and if all sets are exclusive by verifying if the total dimension of the sum of those three sets is equal to the dimension of the entire dataset. In Figure, 21 that property can be checked by comparing those variables dimension output.

B.2 ALGORITHM CONFIGURATION

```
gbm <- h2o.gbm(
  x = predictors,
  y = response,
  training_frame = train_h2o,
```

```
[1] 20367178       84
> dim(train)
[1] 14257024       28
> dim(valid)
[1] 3055077      28
> dim(test)
[1] 3055077      28
> dim(train)+dim(test)+dim(valid)
[1] 20367178       84
> dim(dataset)
[1] 20367178       28
```

Figure 21: Test, valid and train dimension verifications.

```
  validation_frame = valid_h2o,
  ntrees = 200,
  learn_rate = 0.05,
  stopping_rounds = 5, stopping_tolerance = 1e-4, stopping_metric = "AUC",
  sample_rate = 0.8,
  col_sample_rate = 0.8,
  seed = 1234,
  score_tree_interval = 10
)
```

Listing B.2: H2O gradient boost machine configuration.

Listing B.2 shows the configuration of the GBM algorithm, used for the training process. There we specify the predictors as the metric labels from the dataset version, the response label containing the desired **ShouldEager** value, the data frames of training and validation sets and the number of trees to build. This, paired with the configuration of Listing B.3 is able to obtain models with different hyper-parameters for training. Using a grid search that will obtain randomly generated hyperparameters between a set of possible parameters, stored in the `hyper_params` variable, 10 models are trained with different hyperparameters configured. They will train until a stopping criterion is met and all models are then stored in the `grid_1` variable.

```
hyper_params = list(
  max_depth = seq(1,30,1),
  sample_rate = seq(0.2,1,0.01),
  col_sample_rate = seq(0.2,1,0.01),
  col_sample_rate_per_tree = seq(0.2,1,0.01),
  col_sample_rate_change_per_level = seq(0.9,1.1,0.01),
  min_rows = 2^seq(0,log2(nrow(train))-1,1),
  nbins = 2^seq(4,10,1),
  nbins_cats = 2^seq(4,12,1),
  min_split_improvement = c(0,1e-8,1e-6,1e-4),
  histogram_type = c("UniformAdaptive","QuantilesGlobal","RoundRobin")
)

search_criteria = list(
  strategy = "RandomDiscrete",
  max_runtime_secs = 6000,
```

```
  max_models = 10,
  seed = 123,
  stopping_rounds = 5,
  stopping_metric = "AUC",
  stopping_tolerance = 1e-3
)

grid <- h2o.grid(
  hyper_params = hyper_params,
  search_criteria = search_criteria,
  algorithm = "gbm",
  grid_id = "grid_1",
  x = predictors,
  y = response,
  training_frame = train_h2o,
  validation_frame = valid_h2o,
  ntrees = 30,
  learn_rate = 0.05,
  learn_rate_annealing = 0.99,
  max_runtime_secs = 6000,
  stopping_rounds = 5, stopping_tolerance = 1e-4, stopping_metric = "AUC",
  score_tree_interval = 10,
  seed = 1234
)
```

Listing B.3: Hyper-parameters and grid search following a random discrete search.

B.3 EXPORTING BEST MODELS

```
sortedGrid <- h2o.getGrid("grid_1", sort_by="AUC", decreasing = TRUE)

for (i in 1:search_criteria$max_models) {
  gbm <- h2o.getModel(sortedGrid@model_ids[[i]])
  print(h2o.auc(h2o.performance(gbm, valid = TRUE)))
}

exportModel <- h2o.getModel(sortedGrid@model_ids[[1]])
```

Listing B.4: Ordering the models by their accuracy and storing it on the sortedGrid variable.

As shown in Listing B.4, all models are ordered after finishing training by their performance on the test set to evaluate their accuracies and the best performing models are then exported to test in the simulator.