Universidade Federal de Pernambuco
Centro de Informática

Graduação em Engenharia da Computação

PRISMA: A Packet Routing Simulator for Multi-Agent Reinforcement Learning

Tiago da Silva Barros

Trabalho de Graduação
Orientador: José Augusto Suruagy Monteiro

Recife
May 1, 2023

Universidade Federal de Pernambuco
Centro de Informática

Tiago da Silva Barros

PRISMA: A Packet Routing Simulator for Multi-Agent Reinforcement Learning

Trabalho apresentado ao Programa de Graduação em Engenharia da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Engenharia da Computação.

Orientador: *José Augusto Suruagy Monteiro*

Recife
May 1, 2023

List of Figures

Contents

Resumo

Um problema desafiador no estudo de redes é o Roteamento Distribuído de Pacotes (DRP, em Inglês), onde os nós devem definir uma interface de roteamento para os pacotes que chegam. Os métodos de roteamento tradicionais apresentam algumas limitações quando não há uma visão completa sobre a topologia da rede ou sobre o tráfego (ex. Redes Multi-Domínio ou redes ad-hoc sem fio). Um bom candidato para o roteamento de pacotes é o uso de Multi-Agent Reinforcement Learning (MARL), uma abordagem distribuída onde os agentes estão localizados em cada nó e aprendem uma política de roteamento de maneira colaborativa. Alguns trabalhos têm sido realizados nessa direção. Entretanto, eles não usam uma plataforma de simulação realista e padronizada. Eles também não consideram mensagens de protocolo entre os agentes. Assim, propomos o PRISMA [5], um simulador de redes a nível de pacotes baseado na biblioteca de redes NS3 [6]. O PRISMA é capaz de integrar com agentes MARL. Também integramos o PRISMA com mecanismos de mensagens de protocolo e avaliamos o desempenho e o *overhead* na comunicação. Pelo nosso conhecimento, essa é a primeira ferramenta desenhada que objetiva resolver DPR usando MARL e esse é primeiro trabalho a avaliar o *overhead* de comunicação causados pelas mensagens de protocolo. Os resultados indicam um *tradeoff* entre o desempenho e o *overhead* de comunicação, onde o agente o qual performa melhor que o Algoritmo de Melhor Caminho e aproxima do *Roteamento Oráculo* apresenta uma razão de overhead de 150%.

Palavras-chave: *simulação de redes, Multi-Agent Reinforcement Learning, Reoteamento distribuído de pacotes, mensagens de protocolo*

Abstract

One challenging problem in networking is the Distributed Packet Routing (DPR), where the nodes have to define a routing interface for an incoming packet. The traditional routing methods present some limitations when there is no complete view about the network topology or about the traffic (e.g. multi domain systems or wireless ad-hoc networks). One good candidate for packet routing is the usage of Multi-Agent Reinforcement Learning (MARL), a distributed approach, where the agents are located at each node and learn a routing policy in a collaborative way. Some works were done in this direction. However, they don't use a realistic and standard simulation framework. Also, they don't consider the control signalling messages between the agents. Then, we propose PRISMA [5], a packet level network simulator based on the network library NS3 [6]. PRISMA is capable of integrating with a MARL algorithm. We also integrated to PRISMA the control signalling mechanisms and evaluated the performance and the communication overhead. For our knowledge, this is the first tool designed for solving DPR using MARL, and this is the first work that evaluates the communication overhead caused by control signalling. The results indicate a tradeoff between the performance and the communication overhead, where the agent, which outperforms Shortest Path and approximates to the Oracle Routing, presented an overhead ratio of 150%.

Keywords: *Network simulation, Multi-Agent Reinforcement Learning, Distributed packet routing, control signalling*

Acknowledgements

During the undergraduate degree, the experiences I lived helped me to be who I am and the people I had the opportunity of meeting are very important to me in this journey. Then, I thank everyone who helped and supported me during the undergraduate course.

I thank my family, for all the love and support I received. They were essential for this under graduation study. I thank my supervisor, prof. José Augusto Suruagy for helping and advising. His support was fundamental for this work. I would like to thank also my colleagues in the lab INRIA and I3S, where I had the opportunity of doing my internship. I am thankful, specially, to prof. Ramon Aparicio-Pardo and Redha Alliche, which collaborated in this work. Their support were essential.

I also thank my friends and colleagues in the labs ROBOCIN and ESTUFA. There, I had the opportunity of learning new concepts and developing new skills. Their support were fundamental during the undergraduate course.

I thank also my dear friends from UBINET course, from Polytech Sophia-Antipolis. You are very special to me. Also, I thank CAPES for making possible the exchange in Nice, France.

Chapter 1: Introduction

In this chapter, we first present a general context and the motivation for our work in section 1.1. Then, in section 1.2, we present its main objectives.

1.1 Context and motivation

Recently, applications and network management mechanisms in the network domain have attracted a lot of attention in computer science research, such as video streaming [7], scheduling [8] and control congestion [9]. The many types of networks and protocols may lead to complex and dynamic systems. Moreover, nowadays, network applications generate a large amount of data, creating a challenge to the researchers.

In this context, Machine Learning (ML) techniques are good candidates for managing network protocols. These techniques can provide models with good accuracy and precision, for example in avoiding congestion or predicting traffic patterns [10].

One complex task in networking is the packet routing. In this problem, the nodes exchange packets, which must be forwarded towards their final destinations. When a packet arrives at a node, the node has to make a local forwarding decision. One traditional method for deciding the forwarding actions is through the Distributed Packet Routing (DPR). In this approach, as can be seen in Figure 1.1, each node contains its own forwarding algorithm, which decides the forwarding actions and updates the local forwarding tables.

Recently, new network architectures paradigms have appeared, such as the Software Defined Network (SDN). SDN working is described in Figure 1.1. In this paradigm, the switches are connected to the same controller, which is responsible for defining the routing algorithms and updating each switch's forwarding table. In SDN, thus, the decisions are logically centralized and the controller requires access to the complete network. On the other hand, DPR uses a distributed architecture and the nodes can define its routing algorithm locally. Our work is based on the Distributed Packet Routing, since it does not require a centralized deployment.

The packet routing, even in the distributed approach, can lead to a very challenging problem when there is no complete view of the topology or the nodes are not aware about the traffic information (e.g. multi-domain wired and ad-hoc wireless networks). Then, one alternative method is the usage of Multi-Agent Reinforcement Learning (MARL). In this approach, each node (agent) works in a collaborative way applying a Reinforcement Learning (RL) algorithm [11]. The agent aims to build a model capable of learning a good routing policy behavior through several interactions with a given environment. This policy, based on a local network state, performs an action which minimizes the end-to-end delay of the packets.

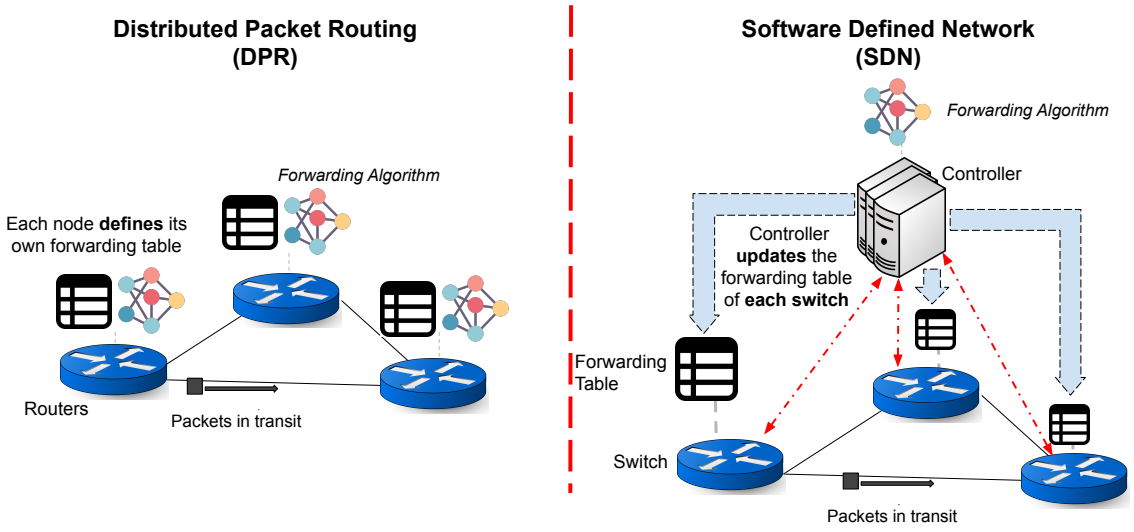

Figure 1.1 Distributed Packet Routing (DPR) vs. Softwared Defined Network (SDN) environments

In order to learn, during the training phase, the nodes have to exchange some information with its neighbors in order to adapt their policies to the different traffic conditions. This extra information, which is shared between the nodes, is called *control signalling*. This information can be seen as the protocol routing messages the node exchanges and it may pose an expressive communication overhead.

Some works [12, 13, 14, 15] presented some methods for the usage of MARL for DPR. They focused in building model designs and training mechanisms in order to improve the agents performance. However, they use simplified ad-hoc simulation tools, which are not standard, neither realistic. Then, due to the lack of reproducibility, it becomes hard to compare the works. Moreover, these works don't implement or evaluate the overhead of control signalling.

For performing the simulations, some important tools are available. The NS3 library [6] may provide useful simulation tools. It generates discrete events and provide several protocol implementations (e.g. TCP, UDP, IP). Furthermore, it also provides some statistical tools, which are useful to measure the network performance.

In order to make the communication between the network simulation and the agent, one important tool is the OpenAI Gym [16]. It creates a generic and well-structured interface, which provides a toolkit allowing the agents to perform actions and receive observations and rewards in a simple and standardized way, using numerical values.

Some works (e.g. Ns3-gym and ns3-ai, [2, 3]) were proposed for acting as an interface between the network simulation using NS3 and the Open AI Gym environment. However, these tools are not adapted for multi-agent approaches and for the Distributed Packet Routing problem.

1.2 Objectives

In this context, we propose the PRISMA (Packet Routing Simulator for Multi-Agent Reinforcement Learning) [5]. The PRISMA [1] is an open-source simulation framework based on an extension of the NS3-gym approach [2] in order to manage DPR problems using MARL. In this work, we aim at the following objectives:

1. **To implement a MARL framework for packet routing allowing realistic and reproducible simulations**. This is the PRISMA framework.

2. **To evaluate the control overhead introduced when we use MARL for packet routing.** A tool like PRISMA is required to achieve this objective.

This is the first tool, from our knowledge, to implement in a reproducible and realistic way network simulations for MARL-DPR. This is also the first work to implement and analyse the control signalling overhead and measure its impact to the models performance.

This report is organized in the following way. Firstly, in chapter 2, a background of some concepts is provided. Then, in chapter 3, the works related to our implementation are presented. In chapter 4, the PRISMA framework is described. Then, the main results of our experiments are discussed in chapter 5. Finally, the main conclusions are drawn in chapter 6.

[1] Repository link available in: https://github.com/tsb4/prisma-v2

Chapter 2: Background

This chapter describes some background concepts for this work. Firstly, a background of distributed packet routing is provided. Then, we discuss the Reinforcement Learning (RL) process for networking. Finally, we present the main methods for introducing the control signalling messages between the agents.

2.1 Packet Routing

Let G(V,E) be a connected graph representing a network with V as the vertices set and $E \subseteq V^2$ as the edges set. Each edge $e \in E$ has a value indicating its cost. These costs can be associated to the latency, link size, link speed or the associated monetary cost, for example [17].

Then, inside a network, the routing task is responsible for finding the neighbor node to which the current node should transmit the packet to get as faster as possible to its final destination [18]. In other words, it aims to find good paths from a source to a destination.

The optimization aimed by the routing can be expressed in the following equation, as seen in [18].

$$n = \underset{y \in neighbors(x)}{\arg\min} \; Q_x(d,y) \tag{2.1}$$

Here, n represents the next hop the package will be delivered to, x is the current node, and $Q_x(d,y)$ is the cost of sending the package from node x to a final destination node d through a node y. Thus, based on this idea, each node must have a forwarding table linking each destination to a neighbor, which will be the next hop for a given packet.

There are several methods of calculating the forwarding table in graphs theory. One of the most known method is the Dijkstra's algorithm [19]. This method, which is used in routing protocols, such as Open Shortest Path First (OSPF), aims to find the shortest path from a specific node to every other node in a connected graph. However, it fails when the nodes don't have a complete view of the network topology (e.g. in Multi Domain Networks).

2.2 Reinforcement Learning for networking

Reinforcement Learning (RL) is a subject which has been studied in some domains, such as robotics and games [20] and it is an interesting approach for networking. Sutton et al. [11] describes how Reinforcement Learning works. The RL approach claims that an agent is capable of learning a behavior through multiple iterations with an environment.

The RL techniques are suitable for solving DPR. In this process, the agents learn a good routing policy capable of managing the packets forwarding. The approach is described in fig. 2.1. Each node n contains a policy π, which represents how desirable an action is. Then, based on the policy, the agent takes an action $a_n \in A_n$, forwarding the packet to the node n'. The set of actions A_n represent the different possible forwarding interfaces for an agent n.

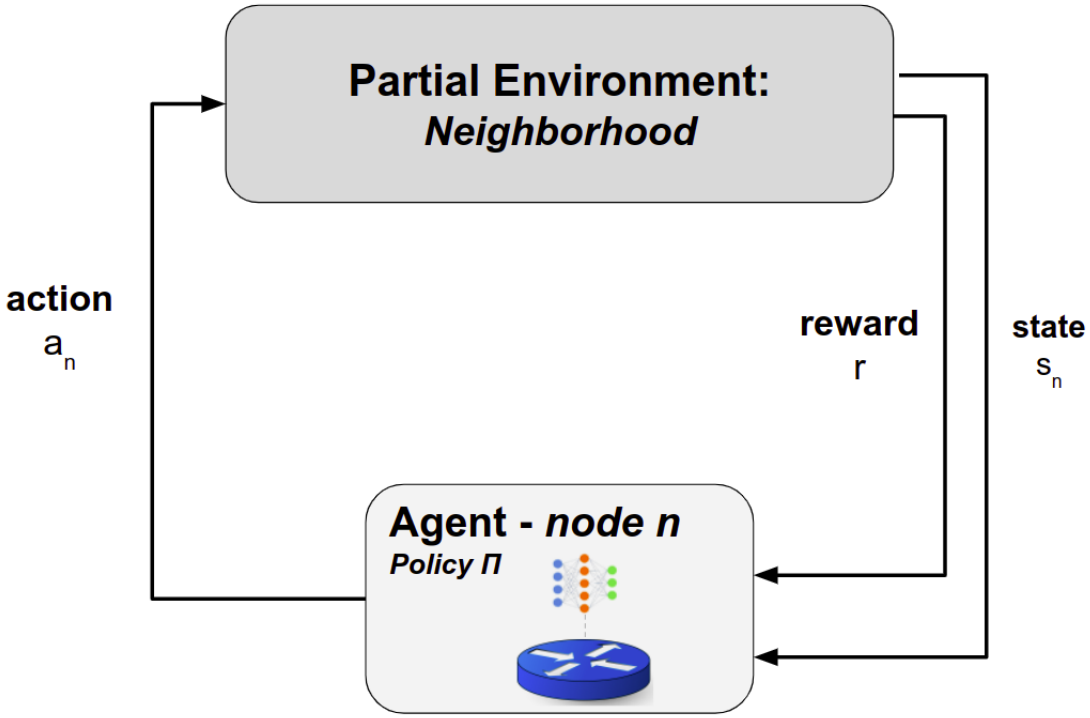

Figure 2.1 Reinforcement Learning architecture

When a packet arrives at the node n', the agent receives network state $s_{n'} \in S_{n'}$. The set of states S'_n represent a set of the possible observations containing the network state for the node n'. The agent also receives from the environment a reward r_n. It defines how good was the state transition from node n to node n' through action a_n. The reward, in our approach, represents the delay in this transition. This process repeats until the packet reaches its final destination.

The accumulated reward until the packet's destination in the several steps may be calculated by the following equation

$$G_n = \sum_{k=0}^{M} \gamma^k r_{node(k)} \tag{2.2}$$

Here, $node(k)$ indicates the node where the packet is at instant k, n is the initial node where the packet was located (i.e. $node(0) = n$) and γ is a discount factor, such that $0 \leq \gamma \leq 1$, that adjusts how much the agent must privilege the closest future rewards. Then, if $\gamma = 1$, all the hops delay are counted equally for the end-to-end delay computation.

The Reinforcement Learning Equation will define the Q value or the state-action value function, $Q^n_\pi(s,a)$, which defines how good it is to take an action a, given a policy π in node n being in state s. This value represents the estimated delay until the packet reaches the final destination depending on the action taken. Its value can be computed as the expected value for the accumulated delay until the packet's destination, as expressed in the following equation:

$$Q_\pi(s,a) = E_\pi[G_n|S_n = s, A_n = a] \tag{2.3}$$

With the Bellman equation, described below, it is possible to calculate the action-state values function recursively, summing the next reward and the state-action value for the next hop. If we consider the packet is forwarded from the node n to node n', the value Q for a node n can be computed using the following equation:

$$Q_\pi(s_n,a_n) = r_n + \gamma * \min_{a_{n'} \in A_{n'}} Q_\pi(s_{n'},a_{n'}) \tag{2.4}$$

Considering the best Q-value (minimal value between the actions) for the node n' equivalent to τ, we can rewrite the above equation in the following way:

$$Q_\pi(s_n,a_n) = r_n + \gamma * \tau \tag{2.5}$$

In the above equation, if we consider $\gamma = 1$, we have that the estimated delay from a given node n until the packet destination is the sum of the next hop delay (i.e., the delay from node n to node n') and the estimated delay from the next hop until the destination, represented by τ. The best Q-value (or the estimated delay until the packet's destination) is called as *target value*.

Thus, the Reinforcement Learning algorithm aims to find an optimal policy which minimizes the target value, i.e., minimizes the end to end delay, according to the following equation:

$$\pi^*(s_n) = argmin_a Q_\pi(s_n,a) \tag{2.6}$$

2.2.1 Deep Q-Network Algorithm

For systems with a large observation space, analysing each state-action value becomes impossible. Then, a possible solution is the usage of Reinforcement Learning combined with techniques of Deep Learning (DL). The Deep Q-Network (DQN) [1] is one algorithm which proposes this combination. It is widely used for environments with a large state, as the case of DPR. The algorithm implements an Artificial Neural Network (ANN), which predicts the Q-Value for each action based on the state, which is passed as input for the neural network. Figure 2.2 describes one example of a neural network architecture in DQN. Here, the state space is composed of 4 values. The hidden layer is composed of 5 neurons and the action apace has a size of 3 elements. Each output $a_i \in A$ in the DQN neural network implementation (which contains weights θ) predicts the Q-Value for the state input and the correspondent action, i.e. $Q_\theta(s,a_i)$.

The algorithm pseudocode can be observed in figure 2.3. For this, during the training period, the agent collects a set of experiences samples which is formed by the tuple $(s_n, a_n, r_n, s_{n'})$, containing the current state, the action performed, the reward obtained and the next state,

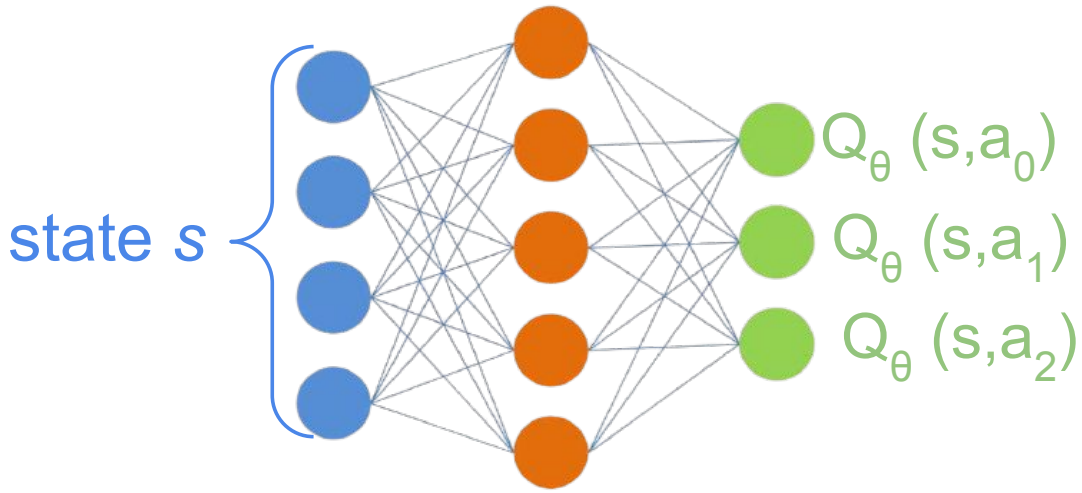

Figure 2.2 DQN Overview

respectively. The experiences samples are stored in a replay buffer and when updating the weights, the agent randomly picks a batch of experience samples for updating the weights. In this process, equation 2.4 is used as the target value for performing the gradient step.

2.2.2 Control Signalling

In the previous section, we could observe that the usage of MARL and DQN introduces a distributed approach for the packet routing problem. For this, the agents need to share some policy information with their neighbors. This is used by the learning process and for the environment changes adaptation (e.g. for traffic changing). In the MARL-DPR problem, the agents need to access the *target value* of the neighborhood. This value is used for estimating the Q-value, according to the equation 2.5.

For accessing the target value, during the training phase, the agents have to continuously exchange some specific packets called control signalling packets. However, these packets introduce an overhead in the network, which may compromise the network performance. Then, it's necessary to find a good trade off between the overhead introduced and the quality of the control signalling mechanism. In this way, the control signalling information has to be large enough to guarantee that the models can learn in a proper way and, on the other hand, if the information is too large, its excessive overhead can disturb the data packets flow and affect the network performance.

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Figure 2.3 DQN Pseudocode [1]

In order to find this trade off, in our approach, we consider two main mechanisms for the control signalling: the estimated *target value* (value sharing) or the entire model responsible for estimating the *target value* (model sharing).

1. **Model Sharing:** In this approach, the nodes share the neural networks weights with the neighbors periodically. Here, a heavy information is exchanged between the nodes.

2. **Value Sharing:** In this approach, the nodes directly exchange the *target value* (estimated delay until the packet's final destination) with their neighbors. This information is, then, used for computing the node's *target value* when updating the policies, according to the equation 2.5. In this control signalling mechanism, the shared information is lighter when compared to the model sharing.

For each mechanism to be deployed, the simulation framework must implement some special packets, which must be properly exchanged by the nodes. These packets are described in section 4.

Chapter 3: Related Works

In this chapter, we will describe the related works. Firstly, we will see some approaches which use RL algorithms for packet routing and how they build the environment information. Then, we will analyse the main tools available for integrating the RL algorithms into a network simulation framework.

3.1 Reinforcement Learning for Packet Routing

3.1.1 Markov Decision Proccess formulation

In recent years, an extensive research in the packet routing field led to the development of new techniques to optimize the problem. One of the most interesting approaches involves the usage of Reinforcement Learning, such as in [12, 13, 14, 15].

In [12], the authors propose an environment, in which periodically, packets are generated from a set of nodes N_s to a set of nodes N_d. Each node contains a FIFO (First-In, First-Out) buffer, where the packets are enqueued before sending. Each link has also a maximal capacity for sending packets. In the Markov Decision Proccess (MDP) described by the authors, the observation space is composed by the packet destination and the additional information (which contains the queue for each node from N_d). The action refers to the next hop. The reward is calculated by the negative of the number of packets in the buffer queue.

The authors in [13] model a similar framework for routing application. Focusing in finding a solution for multi-agent application (when the agents cooperate between themselves to learn the behavior), they propose the following MDP. The state contains 3 components: the destination node, additional information and the information shared between the neighbors. The action, as well as in [12], corresponds to the next hop and the reward is the sum $r = q + l$, where q is the time spent in the queue and l represents the transmission latency to the next hop.

The authors in [14] proposes an algorithm, which is able to generalize for various environment situations. Their focus are the wireless systems and they propose as state a combination of different sources: packet (TTL and position in queue), node (distance to the final destination, queue length and node's degree) and neighbors. Moreover, here, the authors use as input of the model the network state and the action. The output is the Q-value corresponding for that pair state-action. In this way, they can deploy generalized models.

In [15], the authors use Deep Neural Network (DNN) for applying the Deep Q-Routing approach. Here, the observation is composed by the following elements: the current node, the packet's destination, the set of available neighbors and the network adjacency matrix.

3.1.2 Control signalling evaluation

In the literature, the MARL-DPR has not been deeply analysed in terms of signalling overhead. In some works, such as [14], the training procedure is not fully distributed. In this approach, a centralized entity collects the agents model and distribute for every other agent in the network. Thus, the nodes have access to any other agent's model.

Some other approaches [13, 15] use a distributed approach for sharing the target value, but using different mechanisms. In [15], the authors use the value sharing approach, where the agents share with the neighbors the already computed *target value*. On the other hand, in [13], the authors use the model sharing approach, where the neural networks are shared between the neighbors, at each training step, for computing the target value.

The control signalling overhead in previous works were not properly analysed. In this way, we propose evaluating the signalling overhead and the models performance using both approaches, model sharing and value sharing. For this, we use PRISMA, a simulation framework developed by us for the MARL-DPR.

3.2 Simulation Tools for integrating with RL

The NS3 [6] is an useful library which offers several tools for deploying network simulation frameworks. Moreover, the OpenAI Gym [16] serves as an important interface for RL applications. Then, some works were proposed for integrating the NS3 simulation framework to the RL applications.

In [2], the authors introduced and described the ns3-Gym, a platform used for network applications using reinforcement learning. The developed system can be divided into 2 main modules, as is described in figure 3.1, the network simulation (NetSim) and the OpenAI Gym. The module of network simulation is responsible for running the simulation events. It contains the network model and can use several functions and tools provided by the NS3 simulator.

The OpenAI Gym module creates an interface for the agent and provides for this the environment observations and receives the actions. To connect the framework and the environment, the authors developed a middleware, which can be divided into two parts. The gateway is a C++ implementation, located in the simulator. It takes the observation and standardizes it in a numerical format. The proxy is a Python class responsible for making the interface between the Gym environment and the gateway.

The authors created a general framework for network reinforcement problems and, in addition, created some specific environments. One of these is a flow and congestion control environment for the TCP protocol.

On the other hand, the authors in [3] bring another approach in order to implement a framework for Ns-3 simulation. Their solution is more general. It is not only restricted to RL OpenAI Gym API, but to use several python-based ML applications. Their approach, called Ns3-ai, uses a shared memory pool as the way of communication between the simulator and some Python framework for ML, differently from the Ns3-Gym approach which uses sockets to make the communication between the modules.

The choice for using a shared memory is due to the reduced time in transmitting packets.

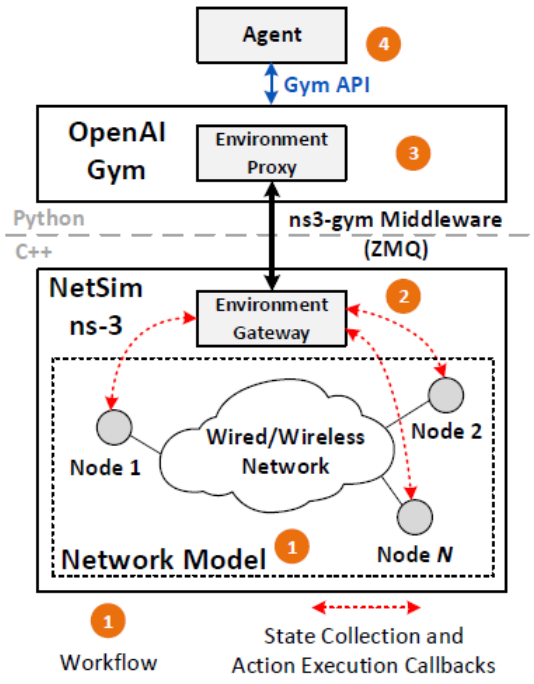

Figure 3.1 System architecture in ns3-gym [2]

Moreover, they implemented a high level interface in C++ and Python programs, so that it becomes easier to adapt for different applications or requirements. The shared memory pool proposed by the authors is described in figure 3.2 and is composed of three modules: the main control block, which controls the entire pool and is responsible for updating its version; the memory block and the control block, which includes the memory block information, such as its size and address.

To validate the proposed architecture, the authors conducted an experiment to predict CQI (Channel Quality Indication) using LSTM and the data generated by the ns-3 simulator. They observed that the trained model achieved an accuracy above 50%, indicating that the framework can provide good performance in learning. They also compared the transmission time between the approach proposed and the one which uses socket as communication method. They applied both approaches into a TCP congestion control using RL and observed that ns3-ai was 50 to 100 times faster than the other proposal.

For this project, we opted for using ns3-gym as base to implement the simulation framework environment. The reason for this is due to the fact that ns3-gym offers a more stable version and is better documented. However, in the future, we think of a migration to ns3-ai since it is based on ns3-gym.

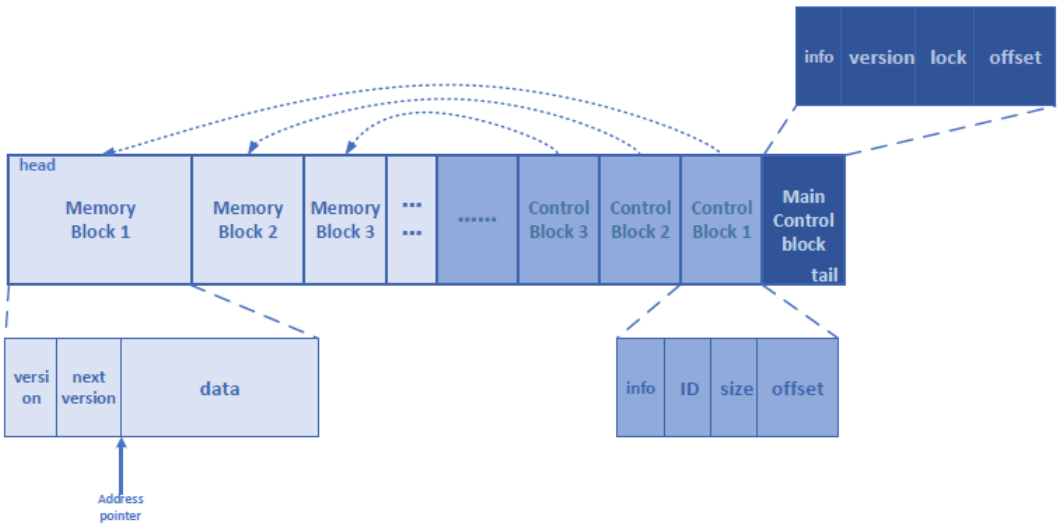

Figure 3.2 Shared memory pool approach [3]

Chapter 4: PRISMA Framework

In this chapter, we will describe the PRISMA framework. First, the PRISMA architecture and its design goals are presented. Then, we will describe the network simulation modelling. Here, we will describe the main NS3 tools used for building the network simulation addressed to packet routing. Finally, the node modules are described and the its logical behavior is presented, including the managing of the control signalling packets.

4.1 Framework Architecture

As mentioned before, we developed the PRISMA [5], a simulation framework for distributed packet routing in order to communicate with a MARL agent. The network simulation uses the NS3 library and it is capable of communicating with a RL agent, which uses the OpenAI Gym toolkit.

During the framework implementation, the following principles were used in the system modeling:

1. **Nodes isolation** In the network simulation, each node policy runs in a separated instance and does not disturb the other nodes functionalities.

2. **Modularity** The code is simple to reuse and change. The object-oriented design allows the reusing of the code, which makes it easier and faster to implement.

3. **Realistic simulation** The network is simulated close to real. The usage of NS3 tools, which uses random variables in order to generate packets, allows a more realistic simulation scenario.

4. **Completeness** The network simulation is complete, containing the simulation and statistical tools for evaluating, such as packet loss rate, end-to-end delay, size of the buffers and signalling overhead.

5. **Easy to configure** The framework provides a simple manner to configure, using parameters. Through them, it is possible to activate and deactivate the signalling mechanisms, as well as tuning the main network parameters.

The system overview is presented in fig. 4.1. Initially, it reads a text file containing the network's adjacency matrix of the network. Based on this information, it builds the network topology as a graph. The framework also reads a text file containing the traffic rate for each

pair of of nodes in the network. The traffic rate information is used in the Poisson Process for the packets generation in the network. More information will be discussed in section 4.3.1.

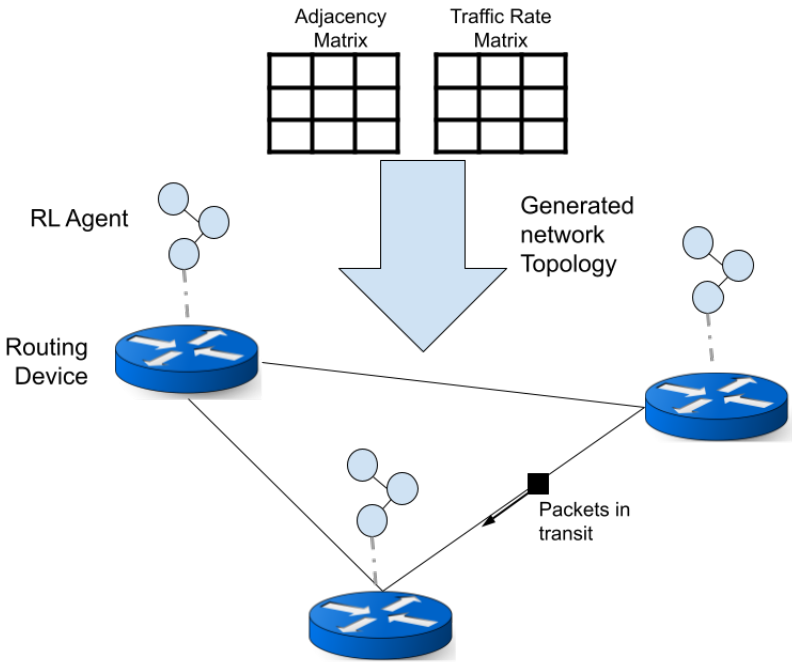

Figure 4.1 System overview

The system architecture design is described in figure 4.2. Inside the PRISMA framework, there are several nodes, which are connected in a wired simulated network. Each node is also connected to an agent, which is implemented in Python. For the communication between the node and the agent, we use the OpenAI Gym [16] toolkit as the communication protocol.

One contribution of this work is the network simulation framework design and implementation. We can divide our contributions into two main parts: our network simulation modelling, which implements, based on the NS3 library, the simulation mechanisms in a wired network; and the node implementation, which describes the logical behavior of the node for solving the DPR problem using MARL.

The node architecture is described in fig. 4.3. The node contains five modules. The data packets generator is the module responsible for generating the data packets, which are sent in the network according to the Traffic Rate Matrix. The control signalling packets generator module creates the packets which are sent as control signalling from each node for all its neighbors during the training phase.

The routing module is responsible for the logical decisions when a packet arrives. It performs the MARL agent action, forwarding the packet and collecting the network information used as state. Furthermore, the module determines the sending of signalling packets. The routing module, as well as the data packets generator and the control signalling packet generator, have access to the network channels, to which the node is connected.

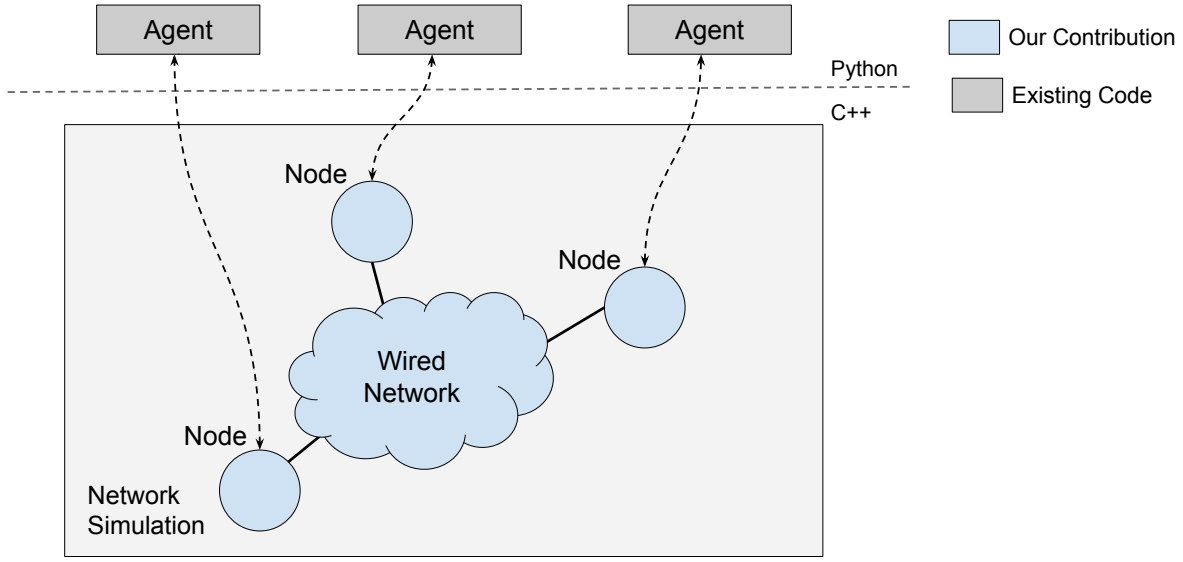

Figure 4.2 System architecture

The communication module is responsible for communicating with the agent, using the Google Protobuf communication protocol. It is directly connected to the routing module. The statistical module is responsible for measuring and providing the statistical measures of the network.

In the following subsection, we describe the proposed network simulator, which uses NS3. After that, we will deep in details for the node's modules implementation.

4.2 The proposed NS3 Network Simulator

Our network simulator was implemented in C++, using the NS3 library [6], which contains several useful tools for networking simulation. The main components used in the framework are described below.

4.2.1 Network

In our network simulator, we designed a wired network, which is described in a single simulation file. Here, the networking tools provided by NS3 [6] are defined, such as the nodes, the channels and the applications. Then, these tools are gathered in order to compose the simulated network. More details about these tools are described in the next sections.

4.2.2 Node

The node tool represents a node element in a network. It contains a unique IP address in the network and is connected to other nodes in a wired network. In addition, the node contains a

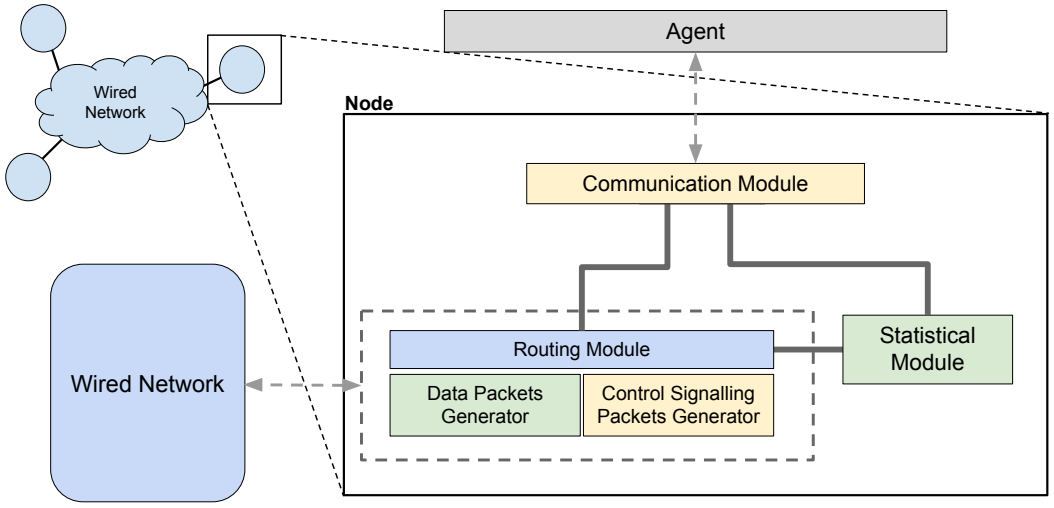

Figure 4.3 Node's architecture

set of running applications, responsible for the packet generation on it, and a set of network devices, which serves as interface between the node and the physical channels. More details about the node implementation are presented in section 4.3.

4.2.3 Net Device

The Net Device tool is responsible for being the interface between a node and a physical channel. A node representation with multiple net devices is indicated in figure 4.4.

The Net Device sends the packets through the channel and receives the packets arriving at the nodes.

When receiving a packet, the Net Device uncaps it until the network layer for accessing the packet information, such as source, destination, and payload size. Each net device has an output FIFO (First-In First-Out) queue for sending a packet, where the packets wait until the transmission. The maximum queue size is configurable and accessible by the nodes during the simulation. When the Net Device receives a packet, but $queueSize + packetSize > maxQueueSize$, the Net Device drops the packet. At this moment, one signal is emitted for the agent, advertising the loss.

4.2.4 Channel

The channel represents the physical link, through which the packets traverses. The channels used in our project are of the point-to-point type. It means that each channel connects exactly two Net Devices. Each channel has a constant associated delay, which indicate the time taken by the packets to propagate through the channel.

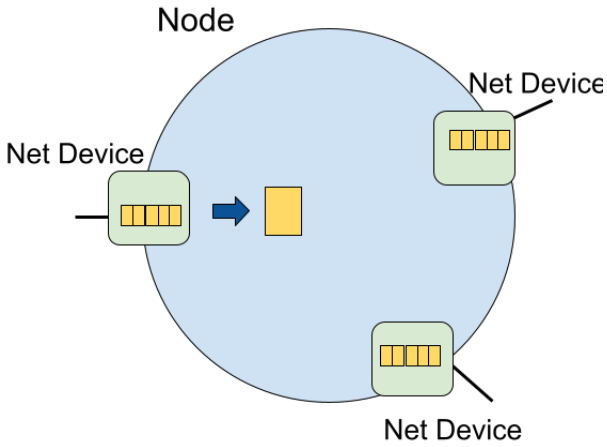

Figure 4.4 Node description in NS3

4.2.5 Packet

The packet class corresponds to the packets which are exchanged by the nodes. We can define two different kinds of packets: data packets and control signalling packets. Each packet contains a tag (identification), indicating its type.

4.2.6 Application

The application tool provides a way for controlling a program running in a node. The applications are responsible for the packets generated inside the nodes. Each node may have several applications running in it. Each application is associated with one socket. Then, after a packet is generated, the socket sends it and the packet is forwarded to the correspondent sending Net Device.

In our project, each node has two applications running: one responsible for generating the data packets and one responsible for sharing the signalling packets. Both applications use UDP sockets for sending. We opted for this protocol since there is no network traffic control mechanisms in UDP. More details about the packets generator implementation are provided in the next sections.

4.3 Node Implementation

The node element implementation can be divided into five modules: The data packets generator, the control signalling generator, the routing module, the communication module and the statistical module. Here, we describe in details each module implementation.

4.3.1 Data Packet Generator

The data packets generator module is responsible for generating packets that are going to be sent to other nodes. In our design, each node generates data packets to all the other nodes present in the network, with a rate according to the value corresponding to the pair indicated in the Traffic Rate Matrix.

In order to approximate the simulation to a real-world process, the simulator can generate traffic information in a non-deterministic way. When each node generates the traffic, it creates the packets with a rate following an exponential random variable.

In this way, building a more complex and realistic simulation, with dynamic data generation, is possible. For this purpose, an application was developed to be a Poisson Traffic Generator. The probability of the exponential distribution can be seen in the equation below, as discussed in [6],

$$P(x)dx = \alpha e^{-\alpha x}dx, \quad x \in [0,\infty) \tag{4.1}$$

where $\alpha = \frac{1}{\mu}$ and μ is the mean of the traffic rate for the pair. The value of the distribution can be found by the following equation

$$x = -1/\alpha \log(\beta) \tag{4.2}$$

where β is a uniform variable between 0 and 1.

4.3.2 Control Signalling Packets Generator

During the training phase, each node shares with its neighbors signalling packets with information that are used for policy updating in the learning process. In our work, we deployed and analysed two main mechanisms for this purpose, as demonstrated in [4].

In the model sharing approach, the nodes exchange with its neighbors the neural network weights in a periodic interval. These weights are used then to calculate the *target value* according to the neighbor's policy. On the other hand, in the value sharing approach, the nodes exchange the already estimated *target value*. This information is, therefore, used for updating the policy according to the equation 2.5.

In order to implement the signalling mechanisms, the nodes exchange specific packets containing the control information. Two types of packets can be used for this purpose: the target update packet and the replay memory update packet.

1. **Target update packet:** The target update packet contains the node's neural network weights. They are generated using a particular application, which runs inside each node. Then, according to a constant *Target Update Period U*, the node retrieves the neural network weights, splits them into segments of *512 bytes* and sends them to each neighbor.

   The weights shared by this packet are equivalent to the policy π in node n' for calculating the *target value* (τ) in equation 2.4.

2. **Replay memory update packet:** This packet contains the information, which is going to be used by the nodes for updating the target value. This packet serves as an answer to

the data packet's arrival. When a node n forwards a data packet p to the node n', this last node creates a replay memory update packet p', which is sent back to node n.

The composition of the replay memory update packet can vary depending on the control signalling mechanism. In model sharing, the replay memory update packet is composed of the reward (indicating the delay from the last hop) and the network state of the node n'. This information is correspondent to the values of r_n and $s_{n'}$, respectively, in equation 2.4, during the process of updating the policy.

In the value sharing approach, the replay memory update packet contains the reward and the target value. These values are correspondent to r_n and τ, respectively, which are going to be used in equation 2.5 for updating policy, as described in section 2.

The control signalling information is summarized in table 4.1. The table indicates that the model sharing approach uses both types of packets, whereas the value sharing uses only the replay memory update packet. However, the replay memory update packet changes according to the signalling mechanism.

Table 4.1 Control signalling mechanisms

	Model Sharing	Value Sharing
Target update packet	Neural network weights	–
Replay memory update packet	Reward and state	Reward and target value

4.3.3 Routing Module

The routing module is responsible for the logical decisions of the node. When a packet arrives at node n, the routing module processes the packet and collects the network information transmitted to the agent. Furthermore, the routing module triggers the transmission of the replay memory update packet. Then, when the agent performs the action, the module in node n forwards the packet to the next hop n'. Table 4.2 gathers the notation used for describing the routing module.

4.3.3.1 Routing module step behavior

The routing module behavior during a simulation step is described in fig. 4.5. When a Net Device receives a data packet, it processes the packet, uncapping it until the network layer. Then, some information, such as source, destination and the payload size are extracted. Also, the tag containing the packet type is read.

Then, the routing module evaluates depending on the type of packet received. If the receiving packet is a control signalling packet (target update packet or replay memory update packet), the node advises the agent, sending the packet information.

If the receiving packet is a data packet, firstly, the packet information is extracted, such as source, destination and payload size. Moreover, the local node information, such as the queues size, is collected. Then, the environment information (node and packet) are transmitted to the

Table 4.2 Notation used in Routing module

Notation	Meaning
N	Total number of nodes
n	current node
n'	Next hop
D	Packet's destination
q_i	Queue's length for the the i^{th} interface
r	Reward
QT	Queueing Time
TT	Transmission Time
PT	Propagation Time
w	Worst Case Possible Delay
tr	Transmission Rate (in *bps*)
ps	Packet Size (in *bits*)
M	Maximal queue's size (in *bits*)

agent. Also, the node creates a *replay memory update packet* and sends it to the node that sent the last data packet received.

If the receiving node is the packet's final destination, the step is ended and the process repeats. Otherwise, the node waits for the action to be performed by the agent. Then, the node performs the action, forwarding the packet through the corresponding interface and the entire process repeats.

4.3.3.2 Environment Information

During a step, when receiving a data packet, the information collected by the node is transmitted to the agent. This information corresponds to the local node information and the packet information, such as destination, source and packet size.

The environment information can be divided into four groups: network state observation, reward, done flag and the extra information. The environment information is described below.

- Network state observation

  The network state observation space is composed by the following vector:

  $$[D, [q_0, q_1, q_2, ..., q_k]] \tag{4.3}$$

  In which D is the final destination of the packet and q_i is the queue length of the i^{th} interface, in bytes. k is the total number of interface links the node has.

- Reward

  The reward is the delay taken by the packet for traversing the last hop. There are two possibilities for calculating this value. If the packet p arrives at the next node n', the reward can be calculated using the following equation:

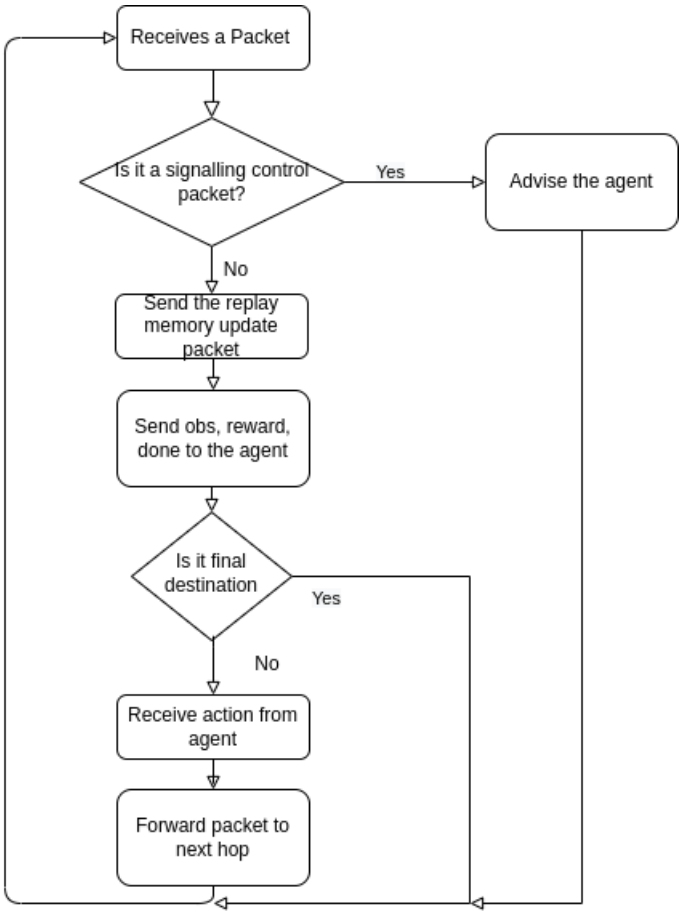

Figure 4.5 Routing module behavior in NS3

$$r = QT + TT + PT \tag{4.4}$$

Here, the QT is the queueing time, i.e., the time where the packet is buffered in the queue waiting to be transmitted; the TT is transmission time, given by equation 4.5, where tr is transmission rate, *in bps*, and ps is the packet size, in bits. Furthermore, the PT is the propagation time of the channel, in seconds.

$$TT = \frac{ps}{tr} \tag{4.5}$$

On the other hand, if the packet is dropped by the node, the reward is calculated by the equation 4.6. This value indicates the worst possible case delay in the network. In this case, the reward considers that all the buffers are full and the packet transits at every node, as can be seen in the equation below.

$$r = w = \left(\frac{(M + ps)}{tr} + PT \right) * N \tag{4.6}$$

Where M is the maximal capacity of the buffer, in bits; ps is the packet size, in bits; tr is the transmission rate, in bps, PT is the channel's propagation delay, in seconds; and N is the total number of nodes in the network. Assuming a high delay for the lost packets, we enforce the policy for avoiding losing packets.

- Done flag

  When a packet arrives at its final destination, the flag "done" is set to true. Otherwise, it is false.

- Extra Info

  The extra information contains further information: packet size, starting simulation time of the packet, current simulation time, and a packet unique ID, as well as the statistical measures, which are described in section 4.3.5.

4.3.4 Communication Module

The communication module is responsible for the communication between the simulated node and the agent. For this, the Google Protobuf Protocol is used. It is an open-source protocol used for communication of structured data, which uses the UDP protocol.

In fig. 4.6, the communication protocol during an episode step is shown. There are three main entities participating in the process. The agent is implemented in Python. The agent is responsible for determining the policy. The communication module, as well as the routing module, are implemented in C++. The communication module serves as interface between the agent and the routing module in the node, which forwards the packets and collects the environment information during the simulation.

When the agent, based on the policy, takes a forwarding action, it transmits the sending command via the Protobuf for the communication module. Then, the communication module calls a *execute action* function and the routing module forwards the packet to the next hop.

When a packet incomes to the node, the routing module gets the environment information described in section 4.3.3.2. The communication module encapsulates all the information onto a single message and the information is transmitted to the agent, which will receive it, and based on the network state, will take the next action.

4.3.5 Statistical module

The PRISMA framework is capable of, in real time, providing statistical measures about the network, which are transmitted to the agent. In this way, it becomes easier to code and deploy the models. The statistical measures provided by the PRISMA, can be divided into three main groups: number of packets, packets delay and signalling overhead. The first group of the measures is composed by the following information:

- Number of Injected Packets in network
- Number of Lost packets

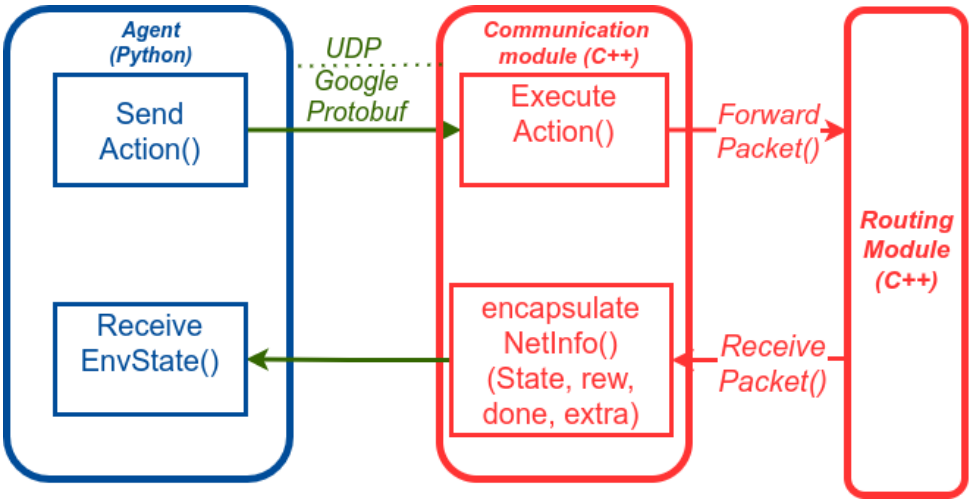

Figure 4.6 Communication module protocol

- Number of packets which arrive at its final destination.
- Number of Packets buffered in the queues

The group of statistics related to the packets delay is described below.

- Average End to End Delay

  Average delay of packets which arrive at its final destination.

- Average cost

  The cost measures the average end-to-end delay with a penalty for the packets which are dropped in the path. For this penalty, we considered the worst possible delay, which is expressed in equation 4.6. The average cost can be computed by the following equation,

$$c = \frac{d + w \cdot l}{l + a} \tag{4.7}$$

  where d represents the sum of the end to end delay, l and a are, respectively, the number of lost and arrived packets; and w is the worst possible delay, as described in equation 4.6.

The statistics about the overhead in signalling computes the Overhead Ratio (OR). It indicates how much signalling is being injected in the network compared to the data traffic. It is computed by the following equation:

$$OR = \frac{\text{CSA}}{\text{DA}} \tag{4.8}$$

where, *CSA* and *DA* are, respectively, Control Signalling and Data packets Amounts, in bytes.

Chapter 5: Simulation Results

In this chapter, we analyse the experimental results for the validation and evaluation of PRISMA framework and the control signalling mechanisms. First, the MARL agent structure using DQN is described. Then, the experiments settings are presented. After, the results evaluating the control signalling overhead are provided Finally, we present the results involving different inference intervals during the test phase. Some of results provided in this section can be observed in [5] and [4].

5.1 Agent

The agent used for the experiments implements a Multi-Layer Perceptron (MLP) neural network, as described in [4]. The general overview of the neural network used can be observed in figure 5.1. The neural network takes as inputs the packet's destination and the buffer occupancy of each channel. The destination information is converted in a one-hot encoded vector. In this way, the neural network presents an input layer with size $N + out\ degree$, in which N and *out degree* correspond to the number of Nodes and the node's number of interfaces, respectively. The neural network contains three hidden layers. The first hidden layer is splitted in two equal parts containing 32 neurons each. The layer then process the inputs separately. After, the layer outputs are concatenated. The second and the third hidden layer contain 64 neurons each and the output layer has size *out degree*, containing the Q-value for each interface. The layers use ELU (Exponential Linear Units) as activation functions.

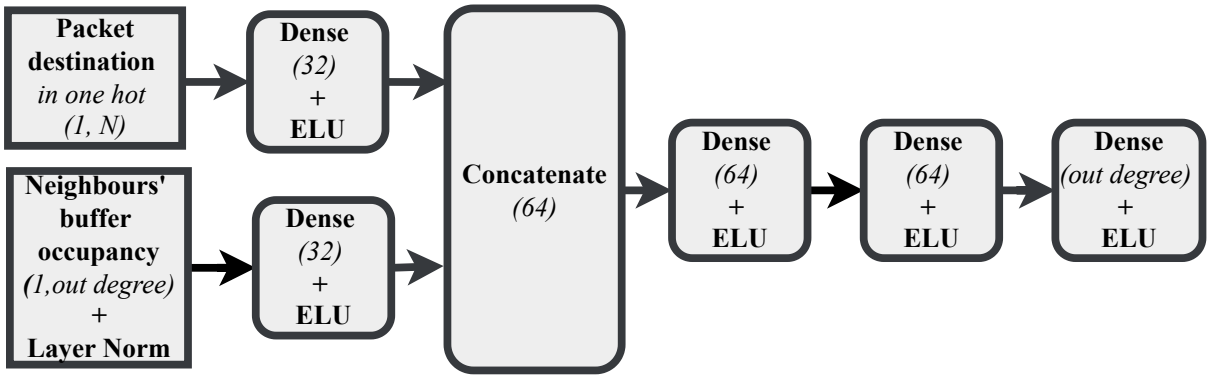

Figure 5.1 Neural Network Overview [4]

5.2 Experiments settings

5.2.1 Hardware Settings

The experiments were ran in a Dell Precision 7920 workstation equipped with an Intel Xeon Gold 6230R Dual CPU (26 Cores, 2.1-4.0 GHz Turbo, 128 GB RAM) with 2 NVIDIA RTX A5000 GPUs. Also we used Intel Core i9-12900H (14 cores HT, 2.5-5.0 GHz Turbo, 32 GB RAM) with NVIDIA RTX A2000.

5.2.2 Network Parameters

For the simulation experiments, we used the Abilene topology [21], which is composed of 11 nodes. The topology representation is described in figure 5.2. For the network parameters, we fixed the channel propagation delay to *1 ms* and the transmission rate to *500 Kbps* for all the simulated links. These values values were chosen in order to allow the simulation experiments in a suitable time. All the channels and Net Devices contain the same delay and rate, respectively.

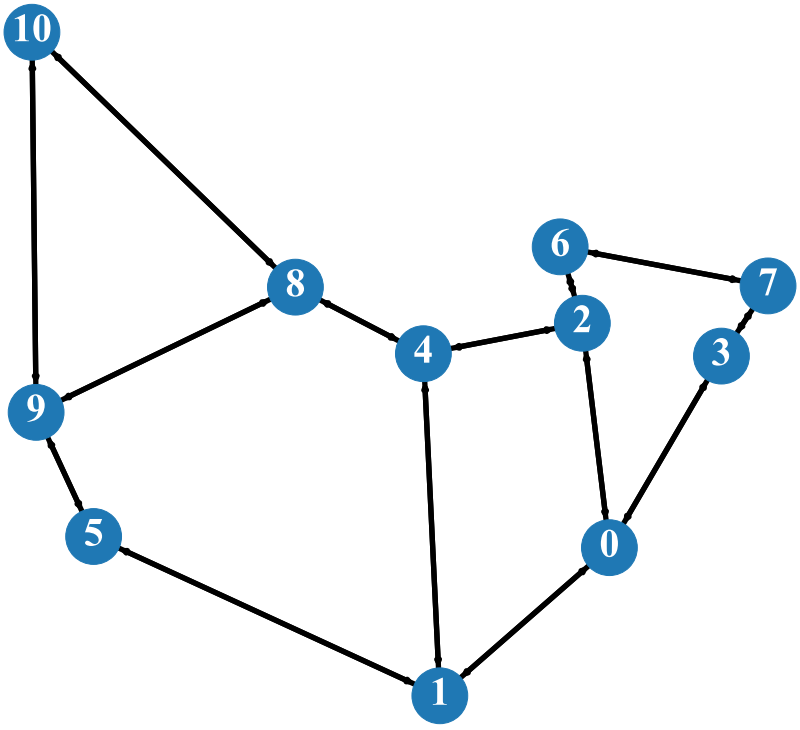

Figure 5.2 Abilene Topology

In order to generate the traffic rate matrix H, we sampled each element h_{sd} from the matrix, where s is the source and d, the destination, from an Uniform Distribution $U(0,1)$. We scale these matrices by multiplying by a coefficient α. We increased α until the largest value α_{max} for which there is no packet loss using the Oracle LP routing. The matrix $\alpha_{max} \cdot H$ is associated

to a load factor $\rho = 1$. In total, we generated four traffic matrices with different coefficients α.

The data packets generated use transport protocol UDP over IP and have a constant payload of *512 B*. In addition, the UDP header has *8 Bytes* and IP header contains *20 Bytes*. Then, summing, the packets have size of *540 B*. The queues located in the Net Devices are set with a maximal capacity of *16260 B*. It corresponds to *30 packets* of *540 B*.

For the control signalling packets, the packets size varies, depending of the signalling mechanism used. Using the model sharing approach, the model which are shared has total size of *36000 B*. The weights are split into segments with *512 B* of payload. For the replay memory update packet, the float and the integers values are encoded in *8 B* slots. Then, for model sharing, the replay memory update packets contains the values of r_n and $s_{n'}$. Then, it is $8 + 8 \cdot KB$ long, where K is the number of node's interfaces.

In value sharing approach, the replay memory update packets have the payload 16 B long, since it contains a pair of values $(r_n, \tau_{n'})$, representing the reward and the next hop's target value, respectively.

For the experiments, we tested two different agents: One using the model sharing and another using the value sharing approach. For the benchmarks, we used the Shortest Path Algorithm, which implements the Dijkstra algorithm [19]. The other benchmark used is the Oracle LP. It is an optimization algorithm using Linear Programming.

We used three different metrics in order to evaluate the framework performance: the packet loss rate, the average end-to-end delay and the average cost, which combines the two other metrics. More details about the cost are described in section 4.3.5. We also evaluated the overhead signalling ratio in the system.

5.2.3 Training parameters

The agents were initially pre-trained using a supervised learning approach in order to have a behavior close to the Shortest Path. In this approach, a dataset of tuples $(d, L_{sp}(d,n'))$ was created as training samples, where d is the final destination and $L_{sp}(d,n')$ is the Shortest path length until d using the node n' as next hop. Then, we minimized the loss and saved the weights for the training phase.

The main hyperparameters used for the training are described in table 5.1. We used ADAM as the optimizer. We also used the ε-greedy as a strategy for the tradeoff between exploration and exploitation, starting with a value $\varepsilon = 1.0$ and ending with $\varepsilon = 0.01$.

Table 5.1 Training hyperparameters

Parameter	value
Simulation time	60s
Learning rate	0.001
Batch size	512
Load factor	0.4
Replay buffer size	5000 (VS), 15000 (MS)
Target update time	[1,2,3,4,5,6,7,8] ms

During the experiments, for the replay buffer size, we used two different values, depending

on the agent used. For the model sharing approach, the replay buffer size was 15000 and for the value sharing, we used 5000. When the replay buffer size is too small, it does not store enough variety of samples and, when the size is too large, the samples can become outdated. Then, the values used look ideal for the models performance. During the experiments, we also varied the target update time used for synchronizing the models during the model sharing agent training. After each training session, the model weights were saved.

5.2.4 Testing parameters

For the testing phase, we evaluated each model using the same traffic matrix we used for training. We tested using different load factors, from 0.6 until 1.4, with interval of 0.1. The idea is testing the network in different traffic conditions, from a low traffic scenario until a very high load traffic configuration. For each traffic matrix simulated, we tested under 5 different seeds.

The testing experiments had the simulation time of 20s. This time is sufficient for the network to reach a stationary phase. During the testing phase, there is no need of generating control signaling packets in the network, since there is no policy improvement.

5.3 Experiments analysis

5.3.1 Control signalling mechanisms analysis

In this subsection, we evaluate the control signalling and its impact in the network.

In Figure 5.3, we can observe the average cost over the target update period. Each point consists of the average between the 9 load factors tested and the 4 traffic matrices. In these experiments, we used the replay buffer size of 15000 for the model sharing. The target update period does not affect the value sharing performance, since it does not use the target update packet during the training.

We can observe that, from 1s to 5s as target update period, the model sharing performance was close to the Oracle LP routing and from target update period of 5s, the average cost for the model sharing approach increases significantly. This suggests that, when using a large update period, the model gets outdated, which affects the model performance.

Comparing the model sharing and the value sharing approaches, we can also observe that the model sharing presents a better performance, with cost close to the Oracle LP, whereas the value sharing presented a performance slightly better when compared to the shortest path. This occurs because, in value sharing, the models continuously update their target value. Then, the model can be using outdated values. On the other hand, in model sharing, the models are updated synchronously. Then, it indicates that the model sharing approach guarantees more stability than the value sharing.

Fig. 5.4 shows the average cost over the control overhead ratio during the training phase. Here, we used the experience replay memory size 5000 and 15000 for value sharing and model sharing, respectively. Each point corresponds to a different value for the target update period. We can observe that the value sharing approach presented a small overhead ratio (0.1) and its performance was slightly better than Shortest Path, presenting as average cost 0.43s.

On the other hand, we can observe that the model sharing approach presented a higher overhead (1.5 for target update period of 5s), since, in this approach, the agents share the entire model. However, it presented a better performance reaching close to Oracle LP routing, with an average cost of 0.33s. We can verify, then, that in order to improve the model performance, communication overhead is a price to pay.

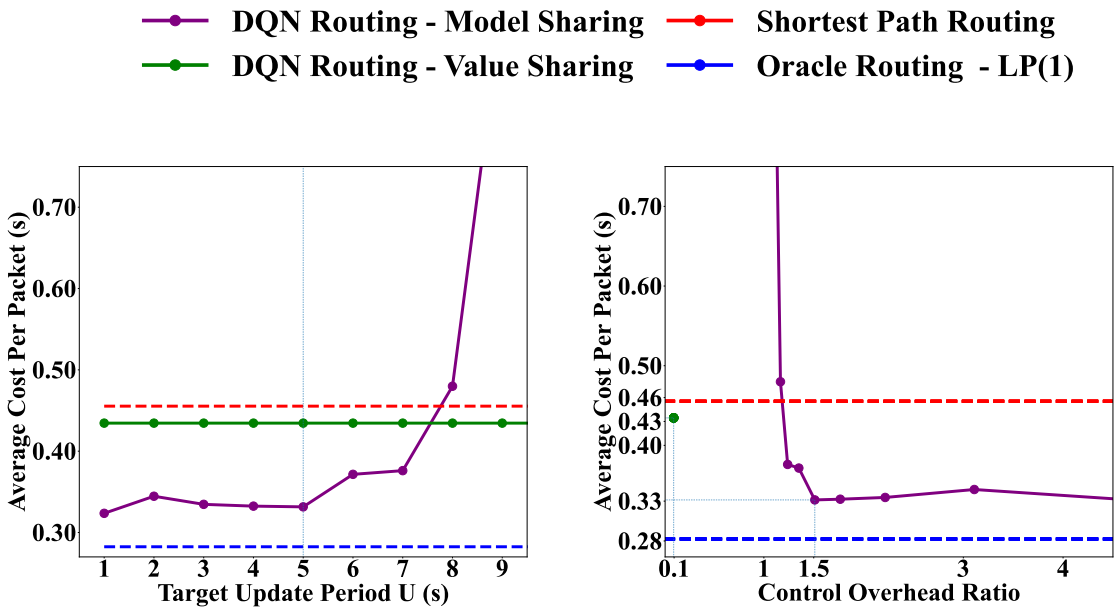

Figure 5.3 Avg cost vs. target update period [4]

Figure 5.4 Average cost vs. overhead ratio [4]

5.3.2 Performance over different loads

We also analysed, during the testing phase, the system performance over different loads. In this scenario, we tested the models on the load factors 0.6 to 1.4, with interval of 0.1. The results are presented in figs. 5.5 - 5.7. In the y axis, there is the evaluated metric: the average loss ratio, average end-to-end delay and the average cost, respectively; and in the x axis the load factor used in the experiment. For each competitor, the maximum, minimum and the average value of the experiments within the four traffic matrices. The average value is represented by the solid line whereas the maximum and minimum value are indicated by the limits of the shaded area.

In figure 5.5, we can observe that the Model Sharing approach presents a lower average loss ratio compared to the Shortest Path and the Value Sharing for all the loads tested. The model sharing presents a value close to the Oracle routing between the loads 0.6 to 0.9. For higher loads, the oracle LP routing presents a lower loss ratio than the model sharing. The value sharing approach provides a loss ratio slightly better than the Shortest Path for all the tested loads.

A similar behavior can be observed in figure 5.6. Here, the results for the average end-to-end delay over the loads are presented. We can observe that for the loads between 0.7 to 1.1, the Model Sharing approach presents a smaller delay when compared to the Oracle Routing.

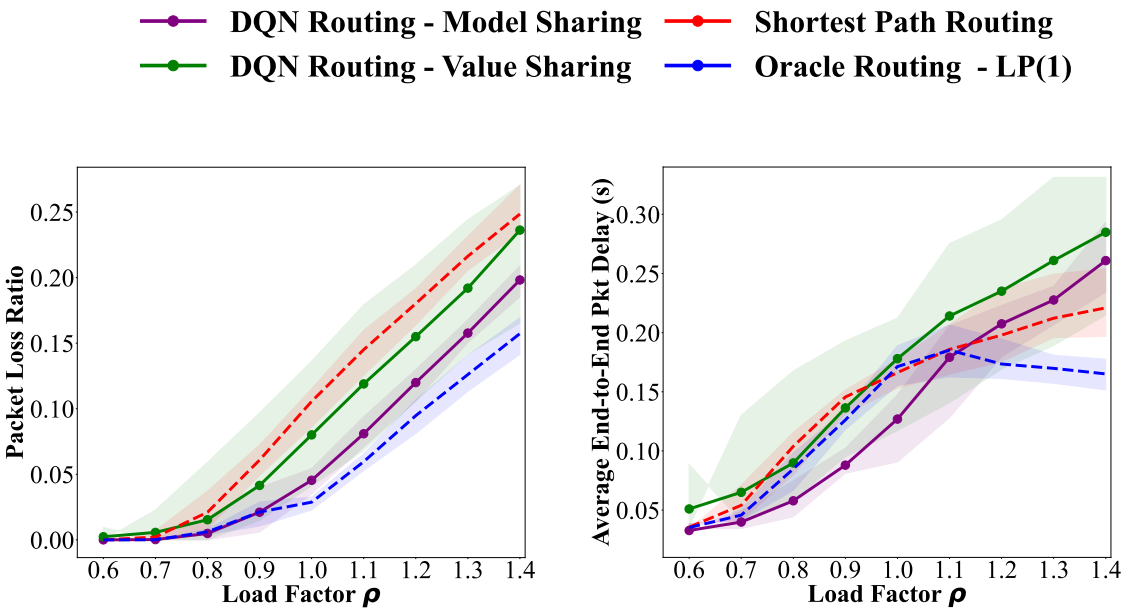

Figure 5.5 Packet Loss Ratio [4]

Figure 5.6 Avg. End-to-End Delay [4]

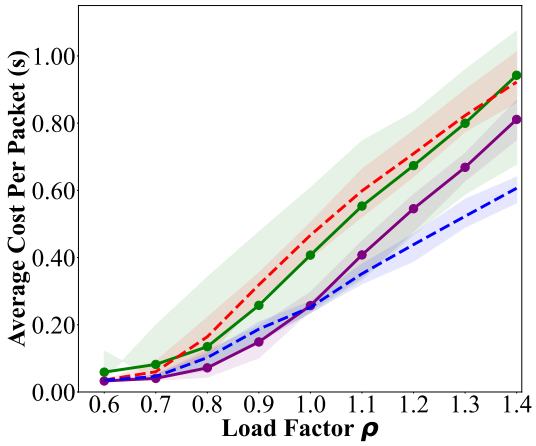

Figure 5.7 Average Cost Per Packet [4]

We can also verify that, from load 1.0, the Shortest Path method presents a smaller average end-to-end delay when compared to the the model sharing approach. This can be explained by the high loss rate verified for the shortest path routing in high traffic loads. Then, the Shortest Path can not manage the traffic load control. Thus, it presents a high loss rate and a not so high end-to-end delay (since the packets always take the same path).

Figure 5.7 represents the average cost of the system over the load factor. We recall that the cost is a combination of the end-to-end delay and the packet loss ratio. We can observe that the model sharing approach outperforms the Shortest Path and the value sharing. The model

sharing outperforms also the Oracle routing between the load 0.7 to 0.9. This behavior can be explained for a smaller end-to-end delay observed for the model sharing during this interval.

5.3.3 Analysing different intervals over testing

We also analysed the PRISMA working latency and the algorithms performance under different *intervals* during the testing phase. The usage of neural networks for deciding the forwarding interface for each incoming packet can be a difficult and slow task. The high execution time during the neural network forwarding can pose a high inference latency, which affects the network working and the MARL approach viability.

The PRISMA's execution time is related to the hardware capabilities of the router. In this way, more powerful routers processors lead to a faster inference of the routing decision. Then, in order to facilitate and allow more possibilities for the practical implementation, the PRISMA simulation framework was adapted and evaluated for different *intervals* during the testing phase.

In the experimented scenarios, we define an interval μ for running the DQN algorithm. This means that at each μ packets forwarded, the node simulates a local forwarding for every node in the network using the current buffer information, saves the forwarding interface corresponding to each destination and use this information for the next μ packets.

We used one traffic matrix for this experiment and the other testing parameters were used as described in section 5.2.4. In this experiment, we trained the model using the *model sharing* signalling mechanism. We tested for the intervals 50, 100 and the original scenario, where at each incoming packet, a neural network forwarding is ran for deciding the output interface.

Figure 5.8 indicate the average cost over the testing load factor for different interval candidates. We can observe that the *original* configuration is the one which presents the smallest cost. This behavior is expected since there is no gap between the neural network forwarding (which decides the output interface) and the packet forwarding moment, which makes the action more accurate. The results with interval $\mu = 50$ and $\mu = 100$ present a slightly worse performance compared to the *original* candidate.

Table 5.2 present the average execution time of packet forwarding for the the different interval candidates, in seconds. We also displayed the results compared to the original time in order to being able to compare. We can observe that the intervals 100 and 50 present a smaller execution time, which are, respectively, 0.1 and 0.19 times smaller when compared to the *original*.

Table 5.2 Training hyperparameters

Interval candidate	Average execution time (standard deviation) (ms)	Ratio average execution time over *original's*
original	3.2 (0.67)	1x
50	0.61(0.02)	0.19x
100	0.32 (0.02)	0.1x

This way, we can observe that the usage of different intervals during the testing phase can be an essential parameter for practical implementation. Using a high frequency considerably

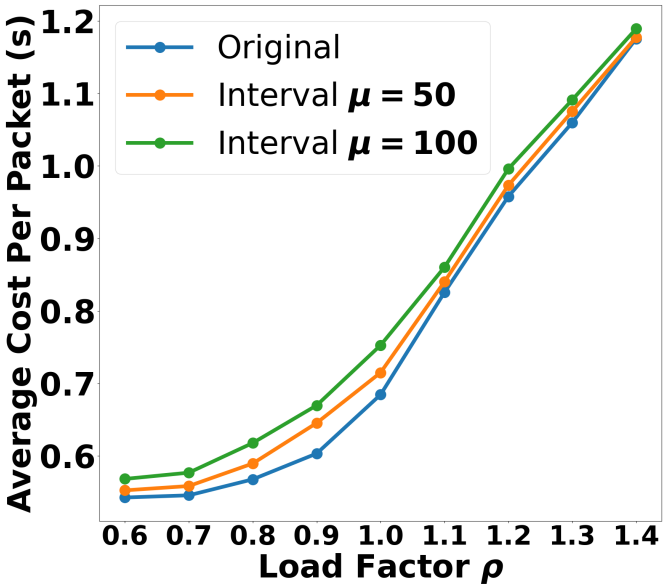

Figure 5.8 Avg. cost over load factor for *interval* candidates.

decreases the average execution time and slightly increases the cost. We intend to deeply analyze this possibility in future works.

Chapter 6: Conclusion and Future Works

In this work, we proposed the PRISMA, a simulation framework for Distributed Packet Routing (DPR) in wired networks. In this problem, when receiving a packet, the node has to decide for which interface it should route the packet. The PRISMA is capable of communicating with a Multi Agent Reinforcement Learning approach. In this approach, each node contains an autonomous agent, which is responsible for the routing policy in the node. We designed the simulation environment using the NS3 library [6] The nodes exchange UDP packets between themselves and we connected each simulated node with an OpenAI Gym [16] environment instance, in order to communicate with the agent.

In our project, we added the control signalling mechanisms, which allows the agents to exchange the models information through the network infrastructure in order to update the policies. We evaluated two control signalling approaches: Model sharing (where the agents exchange the neural network weights) and the value sharing (where the nodes exchange the already computed *target value*).

The experiments show that Value Sharing presents a low communication overhead (with ratio 0.1). However, its performance is just slightly better than Shortest Path. On the other hand, the Model Sharing approach presented a good performance, close to the Oracle LP Routing. However, it presented a high communication overhead (with ratio 1.5). Then, we can see that for reaching a good performance, there is need of more overhead in communication. We also observed that the usage of *interval times* in testing phase can lead to a smaller execution time, with a small performance degradation. This can be essential to the system deployment.

For future works, we plan to analyse the framework when scaling for other topologies (e.g. overlay networks) and link rates. We also want to investigate other control signalling mechanisms, which may provide a smaller overhead and a similar performance when compared to the model sharing.

Bibliography

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[2] Piotr Gawłowicz and Anatolij Zubow. ns-3 meets OpenAI Gym: The Playground for Machine Learning in Networking Research. In *Proceedings of the 22nd International ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, MSWIM '19, pages 113–120, New York, NY, USA, November 2019. Association for Computing Machinery.

[3] Hao Yin, Pengyu Liu, Keshu Liu, Liu Cao, Lytianyang Zhang, Yayu Gao, and Xiaojun Hei. ns3-ai: Fostering Artificial Intelligence Algorithms for Networking Research. In *Proceedings of the 2020 Workshop on ns-3*, WNS3 2020, pages 57–64, New York, NY, USA, June 2020. Association for Computing Machinery.

[4] Redha Abderrahmane Alliche, Tiago Da Silva Barros, Ramon Aparicio-Pardo, and Lucile Sassatelli. Impact evaluation of control signalling onto distributed learning-based packet routing. In *2022 34th International Teletraffic Congress (ITC-34)*, Shenzhen, China, September 2022.

[5] Redha Abderrahmane Alliche, Tiago Da Silva Barros, Ramon Aparicio-Pardo, and Lucile Sassatelli. PRISMA: a packet routing simulator for Multi-Agent reinforcement learning. In *2022 IFIP Networking WKSHPS Network Intelligence*, Catania, Italy, June 2022.

[6] Ns-3 consortium. `https://www.nsnam.org/`. Accessed: 2021-11-25.

[7] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural Adaptive Video Streaming with Pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 197–210, New York, NY, USA, August 2017. Association for Computing Machinery.

[8] Sandeep Chinchali, Pan Hu, Tianshu Chu, Manu Sharma, Manu Bansal, Rakesh Misra, Marco Pavone, and Sachin Katti. Cellular Network Traffic Scheduling With Deep Reinforcement Learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), April 2018. Number: 1.

[9] Wei Li, Fan Zhou, Kaushik Roy Chowdhury, and Waleed Meleis. QTCP: Adaptive Congestion Control with Reinforcement Learning. *IEEE Transactions on Network Science*

and Engineering, 6(3):445–458, July 2019. Conference Name: IEEE Transactions on Network Science and Engineering.

[10] Mowei Wang, Yong Cui, Xin Wang, Shihan Xiao, and Junchen Jiang. Machine Learning for Networking: Workflow, Advances and Opportunities. *IEEE Network*, 32(2):92–99, March 2018. Conference Name: IEEE Network.

[11] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[12] Long Chen, Bin Hu, Zhi-Hong Guan, Lian Zhao, and Xuemin Shen. Multiagent meta-reinforcement learning for adaptive multipath routing optimization. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–13, 2021.

[13] Xinyu You, Xuanjie Li, Yuedong Xu, Hui Feng, Jin Zhao, and Huaicheng Yan. Toward packet routing with fully distributed multiagent deep reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, pages 1–14, 2020.

[14] Victoria Manfredi, Alicia P Wolfe, Bing Wang, and Xiaolan Zhang. Relational deep reinforcement learning for routing in wireless networks. In *2021 IEEE 22nd International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, pages 159–168, 2021-06.

[15] Dmitry Mukhutdinov, Andrey Filchenkov, Anatoly Shalyto, and Valeriy Vyatkin. Multi-agent deep learning for simultaneous optimization for time and energy in distributed routing system. *Future Gen. Computer Systems*, 94:587–600, 2019.

[16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *arXiv:1606.01540 [cs]*, June 2016. arXiv: 1606.01540.

[17] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach*. Pearson, Boston, MA, 7 edition, 2016.

[18] T. Leighton, B. Maggs, and S. Rao. Universal packet routing algorithms. In *[Proceedings 1988] 29th Annual Symposium on Foundations of Computer Science*, pages 256–269, 1988.

[19] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1(1):269–271, dec 1959.

[20] Yifeng Zhu, Devin Schwab, and Manuela Veloso. Learning Primitive Skills for Mobile Robots. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 7597–7603, May 2019. ISSN: 2577-087X.

[21] Sndlib: Library of test instances for survivable fixed telecommunication network design. `http://sndlib.zib.de`.

Appendix A - PRISMA's code structure

The PRISMA framework is available in the link https://github.com/tsb4/prisma-v2. In the framework, the components are modularized. This increases robustness and scalability allowing the user to easily understand and modify the main code. Figure A.1 describes the code structure. The file *sim.cc* plays the main role of the code serving as aggregator of the several components of the system. The *sim.cc* also is the responsible for starting and stopping the simulation.

The *big-signaling-application.cc* and *poisson-application.cc* implement the control signaling packet generator and the Data packets Generator, respectively. They are implemented in the core simulation through the files *big-signaling-app-helper.cc* and *poisson-app-helper.cc*. These files serve as interface for the the generator modules. The *compute-stats.cc* implements the statistical module. It calculates and stores the simulation statistical measures The *data-packet-manager.cc*, *small-signaling-packet-manager.cc* and *big-signaling-packet-manager.cc* implement the routing module. They manage the incoming packets for deciding the action to be performed based on the agent's policy. This module is aggregated in the file *packet-routing-gym.cc*, which communicates with the *agent.py*, which contains the agent's implementation.

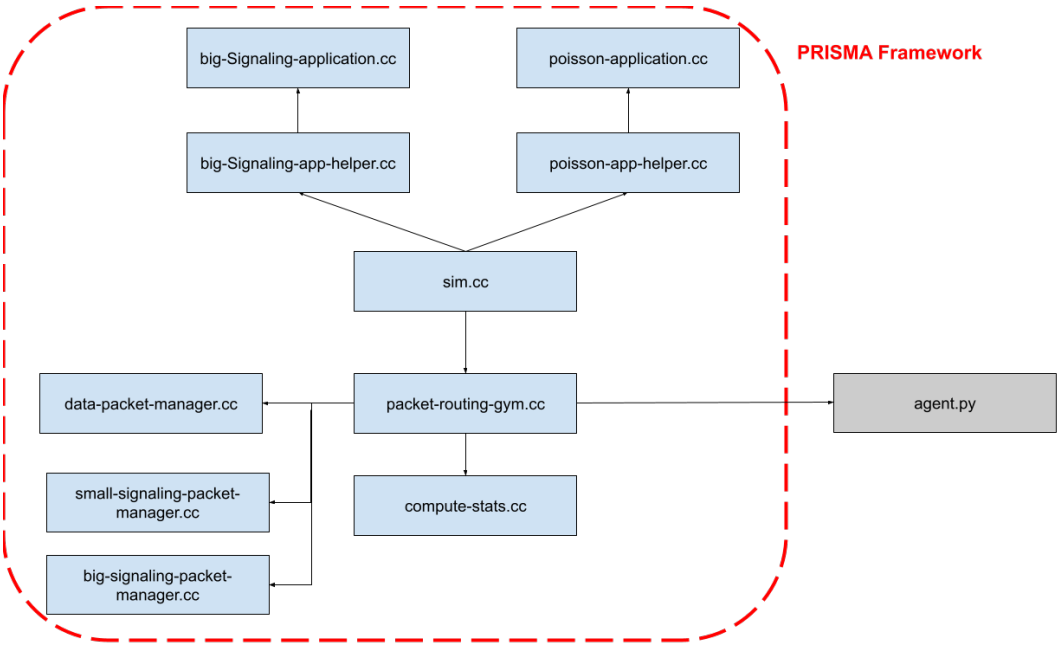

Figure A.1 PRISMA's code structure