DQN-Routing: a Novel Adaptive Routing Algorithm for Torus Networks Based on Deep Reinforcement Learning

Alessandro Lonardo

A thesis submitted
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Information and Communications Engineering

at

SAPIENZA UNIVERSITY OF ROME

DEPARTMENT OF INFORMATION ENGINEERING, ELECTRONICS AND TELECOMMUNICATIONS

Tutor: Prof. Luca De Nardis

Oct 2019

Cycle XXXII

This thesis was evaluated by the two following external referees:

NECTARIOS KOZIRIS
Professor and Dean of the School of Electrical and Computer Engineering at National Technical University of Athens

JAVIER NAVARIDAS
Lecturer in Computer Architecture at the University of Manchester

The time and effort of the external referees in evaluating this thesis, as well as their valuable and constructive suggestions, are very much appreciated and greatly acknowledged.

PhD Advisory Board

Contents

1 Introduction 1

1.1 Exascale Computing: the New Frontier of HPC Architectures 3

1.2 The path towards Exascale systems and beyond: hurdles and challenges 4

1.2.1 Energy efficiency 5

1.2.2 Reliability 7

1.2.3 Storage system 7

1.2.4 Programming models 8

1.2.5 Interconnection Networks 9

1.3 Modern Techniques in Reinforcement Learning 12

1.3.1 Enhancements to DQN 15

2 Interconnection Networks for High-Performance Computing 17

2.1 Networks classification 18

2.2 Network topologies 20

2.2.1 n-Dimensional torus/mesh 21

2.2.2 Fat-tree and Folded-Clos 22

2.2.3 Dragonfly 23

2.3 Router model 25

2.4 Packets, Flits, Phits 26

2.5 Packet Switching Techniques 27

2.5.1 Store And Forward (SAF) 27

2.5.2 Wormhole 29

2.5.3 Virtual Cut-Through (VCT) 30

2.6 Routing algorithms 30

2.6.1 Virtual channels 31

2.6.2 Deadlock 32

2.6.3 Livelock 38

2.7 Selection of Routing Algorithms 39
2.7.1 Dimension-Order Routing (DOR) 39
2.7.2 Star-channel [1] 41
2.7.3 Min-routing [2] 42
2.8 The APEnet Interconnection Network and the ExaNest Project 44
2.8.1 APEnet 44
2.8.2 The ExaNeSt project 48

3 DQN-Routing: a Novel Adaptive Routing Algorithm for Torus Networks Based on Deep Reinforcement Learning 51
3.1 Related Work 52
3.2 The DQN-Routing Algorithm 55
3.3 Considerations on the Implementation of the Algorithm . . . 58

4 Experimental Setup and Measurements 65
4.1 The Experimental Setup 65
4.1.1 The Reinforcement Learning Distributed Execution Framework 66
4.1.2 The Network Simulation Framework 68
4.1.3 Interfacing the Frameworks 69
4.1.4 The Simulated Network Architecture 70
4.2 Traffic Models 71
4.2.1 Uniform Random Traffic 72
4.2.2 DPSNN Simulator 72
4.3 Simulation Results 74
4.3.1 Metrics and Methods 74
4.3.2 Uniform Random Traffic 76
4.3.3 DPSNN Traffic 78

5 Conclusions and Future Work 81

Appendices **83**

A **85**

A.1 Configuration of the Agent 85

List of Tables

2.1 The APEnet roadmap to Exascale 45
2.2 The ExaNeSt multi-tiered network. 49

4.1 Measurements obtained simulating the execution of 400 ms of the DPSNN application distributed on 4x4 processes at different mean neuron firing rates. 79

List of Figures

1.1 Energy efficiency improvement potentials 5
1.2 Bandwidth vs. system distance 9
1.3 Energy vs. system distance 9
1.4 Scheme of a Reinforcement Learning scenario (from Wikipedia) 13

2.1 Classification of interconnection networks, original image from [3] 19
2.2 Example of a 4×4 2D torus (4-ary 2-cube). 21
2.3 Fat-tree topology 22
2.4 Folded-Clos (Fat-tree) network 22
2.5 Fully connected topology example for a dragonfly network. . 23
2.6 Internal structure of a generic router model (LC = link controller). Original image from [3] 25
2.7 Fragmentation of a message into packets, flits and phits. . . . 27
2.8 An example of messages travelling through a wormhole network. Message A is blocked by message B generating a contention. Original image from [3] 28
2.9 (a) Unidirectional ring (left) with its channel dependency graph (right). (b) Modified deadlock-free unidirectional ring (left) with its channel dependency graph (right). Original image from [4] 35
2.10 Path of the DOR algorithm on a 4x4 torus between nodes (3,0) and (0,1). 40

2.11 Subsets of the total dependency graph for a fully connected dragonfly network. Graph 2.11a depicts a standard configuration with no additional virtual channels. Graph 2.11b depicts a deadlock-free configuration with extra virtual channels (identified by a V letter) for local routing of packets coming from other groups. 42
2.12 The layered architecture of APEnet 47
2.13 A block diagram of APEnet architecture 47
2.14 APEnet packet header 48
2.15 APEnet packet footer 48

3.1 Reinforcement Learning scenario for DQN-Routing. 52
3.2 DQN-Routing table entry. 56
3.3 Header of a DQN-Routing regular traffic packet in the APEnet+ interconnect. 58
3.4 Header of a DQN-Routing acknowledge traffic packet in the APEnet+ interconnect. 60

4.1 The integrated simulation framework used as experimental setup in this work. 66
4.2 A stacked modular view of the RLlib library. Original image from [5] 66
4.3 RLlib components, their relations and customizability. Original image from [5] 68
4.4 Monitoring the training of 4 Agents during the simulation (mean Q value) 68
4.5 Monitoring the training of 4 Agents during the simulation (mean reward) 69
4.6 Average packet latency vs normalized applied load under uniform random traffic. 76

4.7 Normalized accepted traffic vs normalized applied load under uniform random traffic. 77

1 Introduction

Torus networks are widely adopted as custom interconnects in High-Performance Computing (HPC) systems because of a series of interesting features, such as their regular physical arrangement, short cabling at low dimensions, good path diversity and good performance for the rather wide class of workloads characterized by local communication patterns.

One of their main disadvantages is that they have a larger diameter compared to other network topologies, resulting in an increased communication latency at large sizes. Using a relatively small sized torus network as the lowest tier of a multi-tiered hybrid interconnect allows exploiting all the advantages of this class of networks while circumventing their inherent limitations, as demonstrated by recent works [6].

A large number of routing algorithms for this class of networks has been proposed throughout the years, ranging from deterministic to fully adaptive ones, with the aim of improving performance – especially under non uniform traffic conditions – and fault-tolerance.

This thesis describes DQN-Routing: a novel, distributed, unicast, fully adaptive non-minimal routing algorithm for torus networks. The idea behind the algorithm is to leverage the constantly ever-increasing availability of ubiquitous computing power to delegate the routing decision to an agent trained by reinforcement learning [7]. The agent is implemented, according to the Deep Reinforcement Learning approach [8, 9], with a convolutional neural network trained with a variant of Q-learning (DQN), having local and first-neighbour routers states and packet source and destination coordinates as inputs, and the value functions estimating future rewards for all possible routing actions as output. The agents calculate the reward corresponding to their routing action using the receive timestamp contained in the acknowledge message sent along the reverse path from destination to source node. These rewards are used to guide the training process toward better performance, *i.e.* to perform routing actions that try to minimize the communication latency of the routed packets given the experienced network state.

In our experimental setup, the routing problem is represented as an independent multi-agent reinforcement learning problem , where the environment is provided by the OMNeT++ [10] discrete event simulator framework modeling a torus network under different traffic conditions. The reference network architectures for our investigation have been APEnet [11] and its latest incarnation, the ExaNet [12] multi-tier hybrid network dedicated to HPC. In this context, we focused on the configuration characterized by a number of nodes in the sub-torus tiers equal to sixteen in a 4x4 bi-dimensional torus, which allowed to effectively simulate the network by means of a single, although powerful, GPU-accelerated workstation.

We compare the performance of DQN-Routing as measured on this experimental setup for different traffic conditions with those obtained by state-of-the-art routing algorithms, using traffic patterns generated both synthetically and by our reference application, the Distributed Polychronous Spiking Neural Network simulator - DPSNN [13].

1.1 Exascale Computing: the New Frontier of HPC Architectures

Next-generation computing systems are being designed to surpass the speed of today's most powerful supercomputers by five to ten times. We will then enter the domain of Exascale computing, when integrated computing system capable of performing at least a exaFLOPS, or 10^{18} floating point operations per second, will enable scientists to tackle problems that were not solvable in terms of their complexity and computation time (the so-called scientific HPC grand-challenges). A not exhaustive list of such problems includes: lattice QCD simulations, ab initio quantum chemistry, high-resolution climate simulations, extreme scale cosmology, and fluid dynamics.

A problem of particular interest is represented by the simulation of the activity of the human brain, as this would pave the way to the understanding of the inner mechanisms of its functioning, and hopefully to the cure of its diseases. On average, a neuron in the brain connects to about 10^4 other neurons. In the human brain, there are approximately 10^{11} neurons and so around 10^{15} connections. Understanding the way the brain works requires the simulation of each of these connections (called synapses). Even a cat brain, which is less than a few percent of a human one, requires a huge data set to be represented, at least a seventh of an exabyte. As of today, with top-tier HPC computing systems in the petaFLOPS range the simulation of the cat brain is just about attainable, but with significantly simplified neurophysiological features [14].

Besides these problems that are fundamental for the advancement of science, there are more application-oriented computing applications in the Exascale domain from which the whole society could have an immediate benefit. To give an idea of the potential disruptive Exascale aftermaths we will mention just a couple of them. Considering the impact on the environment, Exascale systems could enable the development of refined simulation models of advanced engines and gas turbines increasing by 25-50% the efficiency of combustion, and thus reducing the pollution caused by fossil fuels [15]. Focusing on health, the combination of Exascale computing and Deep Learning techniques is expected to enable precision medicine for treatment of cancer by understanding the molecular basis of key protein interactions, developing predictive models for drug response, and extraction of information from millions of cancer patient records to determine optimal cancer treatment strategies [16].

1.2 The path towards Exascale systems and beyond: hurdles and challenges

For decades, the introduction of each new semiconductor manufacturing process led to smaller and faster transistors, and to the doubling in the number of components per integrated circuit approximately every two years, as described by the famous Moore's law. Today, semiconductor process technology has been extended to below 10 nm. Arranging transistors at a smaller scale with the existing materials and technologies represents a big challenge, that involves leaving the classical physics domain and entering the quantum mechanics one. So current technology roadmap, based on the shrinking of the transistor size, may end after three or four generations, posing tremendous challenges to the design of post-Exascale systems. On the other hand, the naive expectation that an Exascale system can be obtained simply enlarging a current Petascale one is readily dispelled by a more thorough examination of the proposals for emerging computing architectures and the common traits of the challenges they are all faced with.

1.2.1 Energy efficiency

The primary constraint for Exascale and post-Exascale systems is their energy efficiency. The number one system in the June 2019 Top500 list [17] of the more powerful HPC deployments in the world is the IBM Summit supercomputer installed at the Oak Ridge National Laboratory, with a 148.6 petaFLOPS maximal achieved performance at a power budget of 10.1 MW. This corresponds to an energy efficiency of 14.7 gigaFLOPS/W, ranking the machine in the second position of the June 2019 Green500 list [18] of the most energy efficient supercomputers in the world, very close to the 15.1 gigaFLOPS/W of the best performing DGX SaturnV Volta. Trying to achieve an exaFLOPS machine simply scaling the IBM Summit supercomputer, assuming it is possible, would yield a $\sim$70 MW power requirement, with associated unsustainable environmental impact and cost of operation.

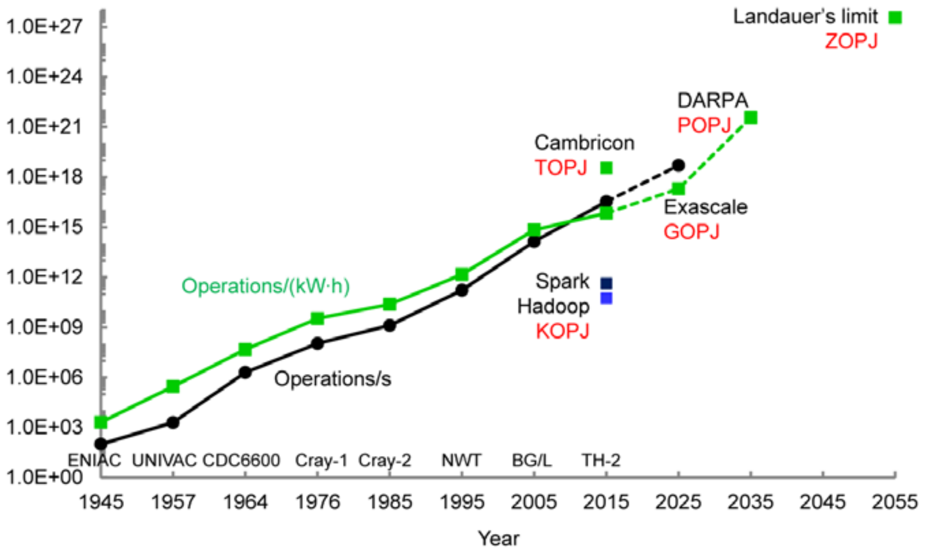

Figure 1.1: Energy efficiency improvement potentials [19]

This energy efficiency bottleneck is the key aspect to take into account in the design and development of Exascale, and post-Exascale, systems. The maximum attainable energy efficiency is dictated by the physical limit proposed by Landauer [20, 21] of $kT \ln 2$ for the energy needed to perform one-bit irreversible operation (e.g. reset to one), corresponding to an energy of 3×10^{-21} J at room temperature and a limit for the energy efficiency of about 0.3 zetta operations per joule (ZOPJ). For comparison, the US DoE's exascale research program has the goal of deploying a 1 exaFLOPS machine in the 20-40 MW range of power consumption, corresponding to 25-50 gigaFLOPS/W around year 2022, while the US DARPA's JUMP program targets a long-term goal of 3 peta operations per joule (POPJ) –not necessarily 64-bit IEEE floating point operations– by around 2035. So there is a gap, and a corresponding space of improvement, of roughly six order of magnitudes before reaching the Landauer's limit, as shown in Figure 1.1 [19].

New technologies are to be explored to progress towards this limit: low-power devices and components, energy-aware system scheduling and compilation chains, low-power systems and cooling technology, and low-power heterogeneous computer architectures characterized by the presence of application specific computing accelerators. For the development of the latter class of devices, a hardware and software co-design approach and the maturation of proper cross-layer design methodologies will be necessary.

One last point that has to be mentioned here, is the fact that thanks to the growth in parallelism at all system scales, even performance of today's systems is increasingly determined by how data is communicated among the numerous devices rather than by the total computation resources available. So, as we will discuss in section 1.2.5, perhaps the most critical aspect to consider for the realization of Exascale and post-Exascale systems is the fundamental challenge of energy efficient data movement. With the energy cost of data movement dominating the total energy consumption and representing a key constraint on the final performance, the interconnection architecture plays a fundamental role towards the design of sustainable future HPC systems.

1.2.2 Reliability

With the end of Dennard scaling around 2005, i.e. when processors stopped getting faster as thermal limitations put an end to frequency scaling, the only way to keep increasing their performance was to increment the number of processing units. As a result, the degree of parallelism for a post-Exascale architecture will likely be very high: a ballpark figure of hundreds of millions concurrency is to be expected. Reliability and resiliency will be critical at this scale of parallelism: errors caused by the failure of components and manufacturing variability will creep in and more drastically affect systems operation. It is highly probable that a post-Exascale system will have an even shorter mean time between failures than current HPC systems, which is only hours [22]. Besides software failures (e.g. file system and kernel related failures), hardware related errors (e.g., uncorrectable memory errors) will be equally or more dominant [23]. Progresses in the fields of systemic fault-awareness and fault-reaction coupled with smart checkpointing and task migration strategies will be mandatory to reach post-Exascale performance.

1.2.3 Storage system

Currently, there is a significant gap between performance of processors and memory systems. This is the consequence of their very different rate of increase during the last 20 years: 50% per year for processors performance vs. 10% per year for memory access performance. As a result, the latency of current processors is below 1 ns, while that of current memory systems is around 100 ns; this performance gap is surely one of the main limiting factors to the development of sustainable HPC systems. The traditional approach to improve storage bandwidth of memory systems by increasing both the clock frequency and the width of the storage bus is bounded by physical limits [24]. The emerging 3D stack memory technologies represent a promising opportunity to break the so-called memory wall, permitting faster clock rates – with suitable processor logic – or permitting multicore access to shared memory using a large number of vertical vias between tiers in the stack [25]. In post-Exascale systems hundreds of millions of processes will perform I/O operations concurrently, requiring hundreds of petabyte (PB) data and dozens of TB/s aggregated bandwidth. Furthermore, this impressive requirements will be combined with the emerging request to support a more complex and diverse set of application domains including, besides traditional HPC, Artificial Intelligence and High Performance Data Analytics. Novel hybrid hierarchy storage architectures will be required to enable high scalability of I/O clients, I/O bandwidth and storage capacity. The extensive use of Non-volatile storage media (NVM) and the integration of storage and Remote Data Memory Access (RDMA) interconnection, along with the development of more scalable and robust distributed filesystems, are promising strategies to significantly improve the efficiency of future storage systems [26].

1.2.4 Programming models

Programming models will have to deal with the very high degree of parallelism expected, allowing the development of software frameworks, possibly domain-specific ones, that will enhance users' productivity in programming, debugging and tuning the applications.

In the the June 2019 Top500 list, the top four systems, and eight in the top ten, integrate some kind of heterogeneous accelerator device in their computing node. This trend will most likely continue in the next years due to performance, energy efficiency, density and costs considerations, so it is essential for Exascale and post-Exascale programming models to fully support heterogeneity of computing devices. Besides this, support for the explicit control of hierarchical parallelism and data locality, to minimize data movement overhead, and the integration of specific functionalities for fault detec-

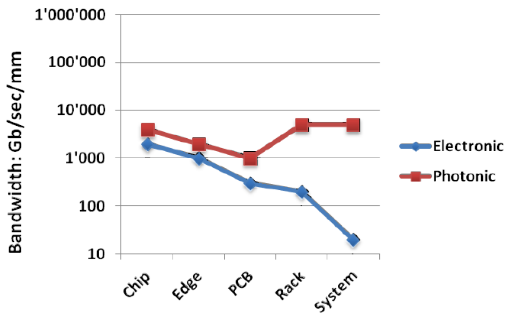

Figure 1.2: Bandwidth vs. system distance

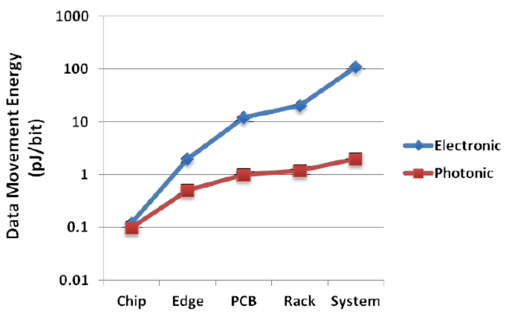

Figure 1.3: Energy vs. system distance

tion and recovery will be mandatory for the full exploitation of the computing resources [27].

1.2.5 Interconnection Networks

The increasing energy consumption and the bandwidth decrease associated with data movement as data propagates from on-chip, across the module (Edge), over the printed circuit boards (PCB), and onto the racks and the whole system is one of the main limiting factor to the scalability of current system architectures. Figure 1.2 shows the bandwidth taper (in blue) for conventional electronic interconnect technology (the units are Gbps/mm of horizontal cross-section), while figure 1.3 shows the relationship between the energy cost of data movement and system distance. For conventional electronic interconnect technology (in blue), off-chip and inter-node communications experience an order of magnitude energy wall and an associated shrinking in bandwidth, while optical photonic interconnect technology (in red) does not face this problems [28].

This spurs the research for interconnection architectures achieving adequate rates of data transfer, or bandwidth, and curtailing the delays, or latency, between the levels while keeping energy consumption under control.

A good design strategy for Exascale and post-Exascale systems interconnect would involve the implementation of a hierarchical network (e.g. intra-node, inter-node, intra-rack, and inter-rack) and the adoption of the technology (wireless, optical photonic and electronic), network topology, and routing algorithms most suited for each level, considering energy efficiency, latency and bandwidth.

The performance increase needed to reach the forthcoming Exascale systems will be determined by the enhancement of the computing node processing capability along with the increase in the number of computing nodes; considering that the single node processing power should be around 10 teraFLOPS, the focus for this systems is on the architecture of an interconnection network of about 100,000 nodes. The required single node communication bandwidth for this architecture can be easily calculated through the bandwidth to processing performance ratio of a current HPC system. Taking the Tiahne-2 [29] as an example of a well-balanced architecture, with its single-node computing performance of 3 teraFLOPS and node communication bandwidth of 112 Gbps, this ratio is around 0.04 bit/FLOP, so for the Exascale node a communication bandwidth of al least 400 Gbps will be needed to maintain the same ratio [30]. However, for some communication-intensive HPC application like for example computational fluid dynamics or graph processing, a higher ratio and so a even higher network bandwidth would be needed to reach optimal performance.

Silicon photonics interconnects technology offers the possibility of delivering the needed communication bandwidths with extremely scalable energy efficiency, although the lack of practical buffering and the fundamental circuit switched nature of optical data communications require a holistic approach to designing system-wide photonic interconnection networks. New network architectures are required and must include arbitration strategies that incorporate the characteristics of the optical physical layer [31]. For this reasons, they will be probably adopted in later post-Exascale systems, while forthcoming Exascale ones will still be based on conventional electronic interconnects.

Besides bandwidth, it's well known that the mean network communication latency is another critical parameters to be addressed in the design of a network interconnect for a HPC system: using a low-latency interconnect minimizes the impact on application's time-to-solution when scaling the system to a large number of computing nodes [32, 33, 34]. However, efficient communication does not come for free. Traditionally, the handling of network protocols deplete precious processor and memory cycles; more specifically, a common rule-of-thumb states that one CPU clock cycle is needed to process one bit of incoming data. Furthermore, by copying the message payload to intermediate buffers at send or receive side, memory bandwidth is needlessly consumed, the caching subsystem is stressed and the communication latency is increased. Another relevant aspect to be taken in consideration in the design of HPC interconnection networks is the fact that for communication-intensive applications, changes in performance are more highly correlated with changes of variation in network latency than with changes of mean network latency alone [34].

The implementation of host offloading functionalities, such as zero-copy [35] and Remote Direct Memory Access (RDMA) [36], in *smart network interfaces* has proven to be an effective approach to face these problems, and will be most likely further developed for next generations HPC interconnection networks. Dedicated or configurable computation engines embedded in the network devices have already started to appear: for example the Mellanox Scalable Hierarchical Aggregation and Reduction Protocol (SHARP) [37] is based on dedicated compute engines located on the Infiniband switch data path dedicated for performing data reduction and aggregation operations. These hardware enhancements are referred to as In-Network Computing.

The network topology defines the way in which the computing nodes are connected, and has also a great impact on the performance of the system. For HPC interconnection networks, besides the traditional torus and fat-tree topologies, more recent ones have been proposed, such as Dragonfly [2]. The properties of these topologies are well-known and they have been used in the design of supercomputers over the last decades. All these topologies are characterized by a rigid structure; recently Jellyfish, a network interconnect adopting a random graph topology and featuring better incremental expandability, shorter paths, and better resilience to failures, has been presented [38]. Again, a multi-tiered network design characterized by the best match between network topology (along with the corresponding routing algorithms) and the network tier (intra-node, inter-node, intra-rack, and inter-rack), will be needed to reach optimal performance for next generations HPC interconnection networks. For example, a two-tiered hybrid interconnect having a torus network as the lower tier and a fat-tree as the upper one can outperform state-of-the-art pure torus and fat-tree networks as long as the density of connections between the two tiers is high enough and the size of the subtori is limited to a few nodes per dimension [6].

1.3 Modern Techniques in Reinforcement Learning

Since this work is heavily based on the exploitation of recent achievement in the branch of Artificial Intelligence known as Deep Reinforcement Learning, we provide here a short overview on the topic [7, 39].

Reinforcement learning is about the problem of an *agent* learning to act, in the sense of mapping situations to actions, in an *environment* in order to maximize a scalar *reward* signal. This is a *closed-loop* problem, as the agent actions have an effect of its later inputs. No direct supervision is provided to the agent, for instance it is never directly told what is the best action in a given situation, instead it must find-out which actions will provide the highest rewards by *exploration*.

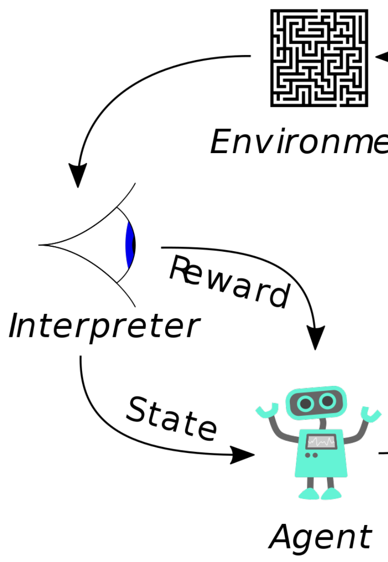

Figure 1.4: Scheme of a Reinforcement Learning scenario (from Wikipedia)

Agents and environments. At each discrete time step $t = 0, 1, 2 \ldots$, the environment provides the agent with an observation S_t, the agent responds by selecting an action A_t, and then the environment provides the next reward R_{t+1}, discount γ_{t+1}, and state S_{t+1}. This interaction is formalized as a *Markov Decision Process*, or MDP, which is a tuple $\langle \mathcal{S}, \mathcal{A}, T, r, \gamma \rangle$, where $\mathcal{S}$ is a finite set of states, $\mathcal{A}$ is a finite set of actions, $T(s, a, s') = P[S_{t+1} = s' \mid S_t = s, A_t = a]$ is the (stochastic) transition function, $r(s, a) = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a]$ is the reward function, and $\gamma \in [0, 1]$ is a discount factor. In our experiments MDPs will be *episodic* with a constant $\gamma_t = \gamma$, except on episode termination where $\gamma_t = 0$, but the algorithms are expressed in the general form.

On the agent side, action selection is given by a policy π that defines a probability distribution over actions for each state. From the state S_t encountered at time t, we define the discounted return $G_t = \sum_{k=0}^{\infty} \gamma_t^{(k)} R_{t+k+1}$ as the discounted sum of future rewards collected by the agent, where the discount for a reward k steps in the future is given by the product of discounts before that time, $\gamma_t^{(k)} = \prod_{i=1}^{k} \gamma_{t+i}$. An agent aims to maximize the expected discounted return by finding a good policy.

The policy may be learned directly, or it may be constructed as a function of some other learned quantities. In value-based reinforcement learning, the agent learns an estimate of the expected discounted return, or value, when following a policy π starting from a given state, $v^\pi(s) = E_\pi[G_t|S_t = s]$, or state-action pair, $q^\pi(s,a) = E_\pi[G_t|S_t = s, A_t = a]$. A common way of deriving a new policy from a state-action value function is to act ϵ-greedily with respect to the action values. This corresponds to taking the action with the highest value (the *greedy* action) with probability $(1-\epsilon)$, and to otherwise act uniformly at random with probability ϵ. Policies of this kind are used to introduce a form of *exploration*: by randomly selecting actions that are sub-optimal according to its current estimates, the agent can discover and correct its estimates when appropriate. The main limitation is that it is difficult to discover alternative courses of action that extend far into the future; this has motivated research on more directed forms of exploration.

Deep reinforcement learning and DQN. Large state and/or action spaces make it intractable to learn Q value estimates for each state and action pair independently. In deep reinforcement learning, we represent the various components of agents, such as policies $\pi(s,a)$ or values $q(s,a)$, with deep (i.e., multi-layer) neural networks. The parameters of these networks are trained by gradient descent to minimize some suitable loss function.

In DQN [8, 9] deep networks and reinforcement learning were successfully combined by using a convolutional neural net to approximate the action values for a given state S_t (which is fed as input to the network in the form of a stack of raw pixel frames). At each step, based on the current state, the agent selects an action ϵ-greedily with respect to the action values, and adds a transition $(S_t, A_t, R_{t+1}, \gamma_{t+1}, S_{t+1})$ to a replay memory buffer, that holds the last million transitions. The parameters of the neural network are optimized by using stochastic gradient descent to minimize the loss

$$(R_{t+1} + \gamma_{t+1} \max_{a'} q_{\overline{\theta}}(S_{t+1}, a') - q_\theta(S_t, A_t))^2 \ , \tag{1.1}$$

where t is a time step randomly picked from the replay memory. The gradient of the loss is back-propagated only into the parameters θ of the *online network* (which is also used to select actions); the term $\overline{\theta}$ represents the parameters of a *target network*; a periodic copy of the online network which is not directly optimized. The optimization is performed using RMSprop [40], a variant of stochastic gradient descent, on mini-batches sampled uniformly from the experience replay. This means that in the loss above, the time index t will be a random time index from the last million transitions, rather than the current time. The use of experience replay and target networks enables relatively stable learning of Q values, and led to super-human performance on several Atari games.

1.3.1 Enhancements to DQN

Several extensions to DQN have been proposed to overcome the limitations that have become evident with its wide adoption. Here we describe three extensions that have been used in this work.

Double Q-learning. Conventional Q-learning is affected by an overestimation bias, due to the maximization step in Equation 1.1, and this can harm learning. Double Q-learning [41], addresses this overestimation by decoupling, in the maximization performed for the bootstrap target, the selection of the action from its evaluation. It is possible to effectively combine this with DQN [42], using the loss

$$(R_{t+1} + \gamma_{t+1} q_{\overline{\theta}}(S_{t+1}, \operatorname*{argmax}_{a'} q_\theta(S_{t+1}, a')) - q_\theta(S_t, A_t))^2.$$

This change was shown to reduce harmful overestimations that were present for DQN, thereby improving performance.

Prioritized replay. DQN samples uniformly from the replay buffer. Ideally, we want to sample more frequently those transitions from which there is much to learn. As a proxy for learning potential, prioritized experience replay [43] samples transitions with probability p_t relative to the last encountered absolute *TD error*:

$$p_t \propto \left| R_{t+1} + \gamma_{t+1} \max_{a'} q_{\bar{\theta}}(S_{t+1}, a') - q_\theta(S_t, A_t) \right|^\omega ,$$

where ω is a hyper-parameter that determines the shape of the distribution. New transitions are inserted into the replay buffer with maximum priority, providing a bias towards recent transitions. Note that stochastic transitions might also be favoured, even when there is little left to learn about them.

Dueling networks. The dueling network is a neural network architecture designed for value based RL. It features two streams of computation, the value and advantage streams, sharing a convolutional encoder, and merged by a special aggregator [44]. This corresponds to the following factorization of action values:

$$q_\theta(s, a) = v_\eta(f_\xi(s)) + a_\psi(f_\xi(s), a) - \frac{\sum_{a'} a_\psi(f_\xi(s), a')}{N_{\text{actions}}},$$

where ξ, η, and ψ are, respectively, the parameters of the shared encoder f_ξ, of the value stream v_η, and of the advantage stream a_ψ; and $\theta = \{\xi, \eta, \psi\}$ is their concatenation.

2 Interconnection Networks for High-Performance Computing

The research in High-Performance Computing (HPC) architectures tries to give an answer to the ever-increasing demand for computing power. Today, the target for top-notch systems is in the range of the exascale computing, that means an integrated computing system capable of performing at least a exaFLOPS, or a billion billion floating-point operations per second (the prefix *exa-* denotes a factor of 10^{18}).

As we discussed in section 1.2.5, the design of a scalable and low-latency interconnection network is one of the main challenges in developing such distributed-memory multiprocessor systems: the communication subsystem turns out to be the bottleneck in most applications, with the *inter-node* communication latency, and its stability, being one of the most critical parameters of the overall parallel computing architecture.

After providing a summary of nomenclature and concepts used in the domain of interconnection networks according to the reference books from J. Duato et al. [3] and W. Dally et al. [45] we describe the architecture of the APEnet interconnection network [46] implementing, in its latest incarnation, the physical, data link and network layers of the ExaNeSt [12, 47] HPC architecture.

2.1 Networks classification

A classification of interconnection networks is shown in Figure 2.1; this scheme is not fully exhaustive but it is more than adequate for our purposes. According to this schema we can categorize networks in four main categories: *shared-medium* networks, *direct* network, *indirect* network and *hybrid* networks.

Shared-medium networks use a common medium to connect all the devices. Due to its shared nature the network experiences a severe performance degradation when the number of nodes increases. They usually provide good multicast/broadcast[1] performance and are used in small systems like multi-CPU nodes.

Direct networks uses a point-to-point node-to-node interconnection. Every node is a compute unit with its own processor, memory and peripherals. Each node has a router block which handles the communication with a subset of the nodes called neighbours and all the nodes are connected according to a given network topology. To establish communication between non-neighbour nodes intermediates steps are used according to a routing function. Direct networks offer good scalability and are used in many HPC systems.

Indirect networks are made of nodes interconnected through switches. Every node is a compute unit but it has no routing capability, the only network functionality of the node is carried out by the network adapter which connects the node to a switch. Every switch has a fixed number of ports and every port can be connected to: a node, another switch or not connected. The connection of the switches generates the network topology and the routing algorithm selects the path between the nodes. Having the compute node outside of the switch increases the distance across two nodes by two producing higher net-

[1]A multicast is a special send of a message from a node to many nodes, a broadcast is a particular case of multicast in which a message is sent to all the nodes

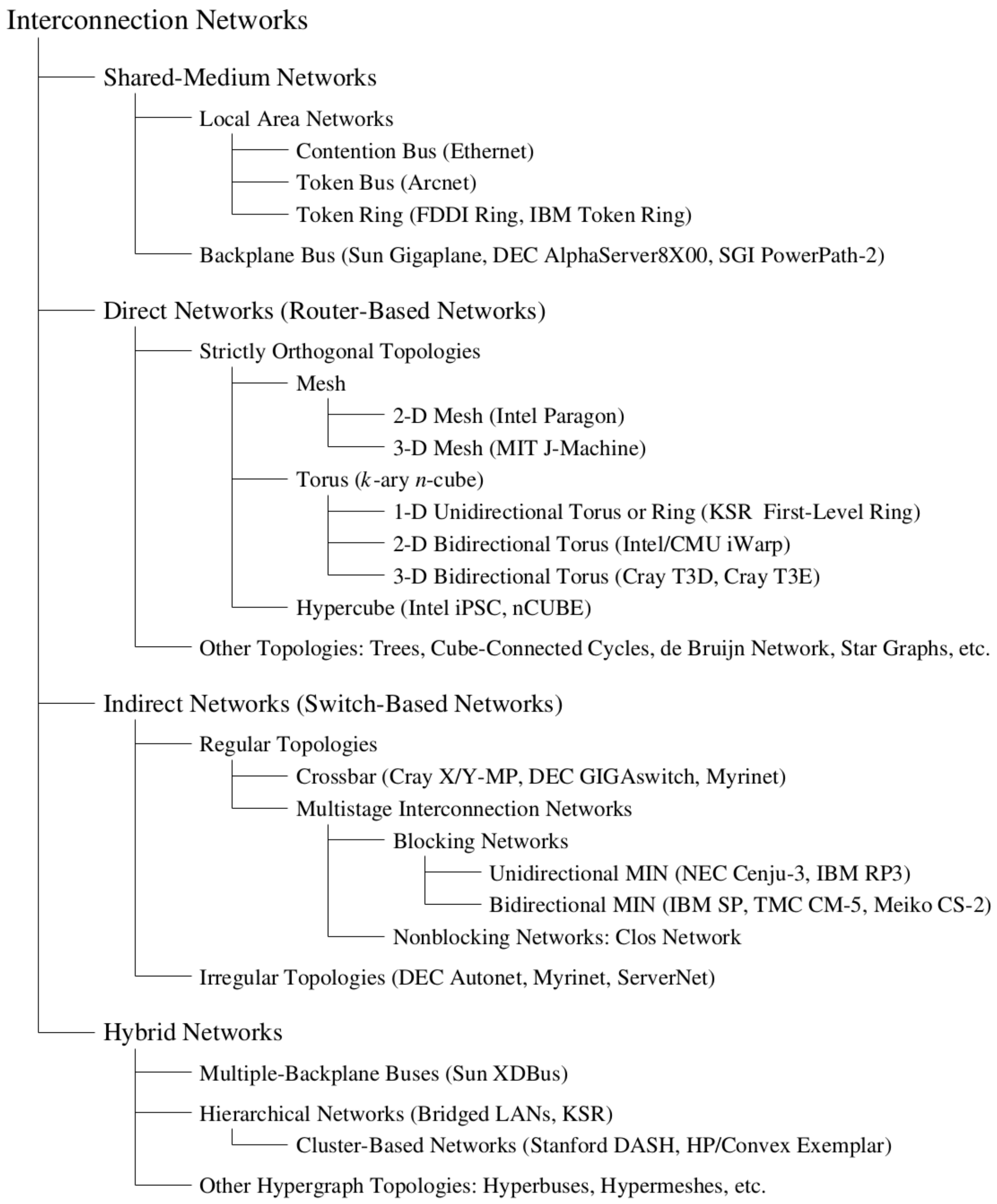
Interconnection Networks
Shared-Medium Networks
Local Area Networks
Contention Bus (Ethernet)
Token Bus (Arcnet)
Token Ring (FDDI Ring, IBM Token Ring)
Backplane Bus (Sun Gigaplane, DEC AlphaServer8X00, SGI PowerPath-2)
Direct Networks (Router-Based Networks)
Strictly Orthogonal Topologies
Mesh
2-D Mesh (Intel Paragon)
3-D Mesh (MIT J-Machine)
Torus (*k*-ary *n*-cube)
1-D Unidirectional Torus or Ring (KSR First-Level Ring)
2-D Bidirectional Torus (Intel/CMU iWarp)
3-D Bidirectional Torus (Cray T3D, Cray T3E)
Hypercube (Intel iPSC, nCUBE)
Other Topologies: Trees, Cube-Connected Cycles, de Bruijn Network, Star Graphs, etc.
Indirect Networks (Switch-Based Networks)
Regular Topologies
Crossbar (Cray X/Y-MP, DEC GIGAswitch, Myrinet)
Multistage Interconnection Networks
Blocking Networks
Unidirectional MIN (NEC Cenju-3, IBM RP3)
Bidirectional MIN (IBM SP, TMC CM-5, Meiko CS-2)
Nonblocking Networks: Clos Network
Irregular Topologies (DEC Autonet, Myrinet, ServerNet)
Hybrid Networks
Multiple-Backplane Buses (Sun XDBus)
Hierarchical Networks (Bridged LANs, KSR)
Cluster-Based Networks (Stanford DASH, HP/Convex Exemplar)
Other Hypergraph Topologies: Hyperbuses, Hypermeshes, etc.

Figure 2.1: Classification of interconnection networks, original image from [3]

work latency[2]. Indirect networks are commonly used in data centres because they are easier to maintain and do not require specialized network capabilities from the compute nodes.

Hybrid networks combine the use of direct and indirect network to achieve better performance. To mitigate the performance degradation of direct networks a hierarchical structure can be used, generating islands of direct networks interconnected using indirect networks. This approach is gaining acceptance into the HPC community.

2.2 Network topologies

In both direct and indirect networks the topology can be modelled by a directed graph $G(N, C)$, where the edges of the graph represent the C unidirectional communication channels[3] and the vertices the N switches or nodes for indirect and direct networks respectively. Using this simple model we can define some basic network properties from the graph representation:

- *Node degree/Switch radix:* Number of channels connecting a specific node/switch to its neighbours.
- *Distance:* The distance $d(a, b)$ between node a and node b of the network is defined as the minimum number of hops required for going from a to b. Since graph is directed the distance may not be commutative.
- *Diameter:* The maximum distance between two nodes in the network.
- *Regularity:* A network is *regular* when all the nodes have the same degree.
- *Symmetry:* A network is *symmetric* when it looks alike from every node.

[2]Network latency is defined as the time to wait before receiving the data over the network.
[3]Bidirectional channels can be represented as a couple of unidirectional ones.

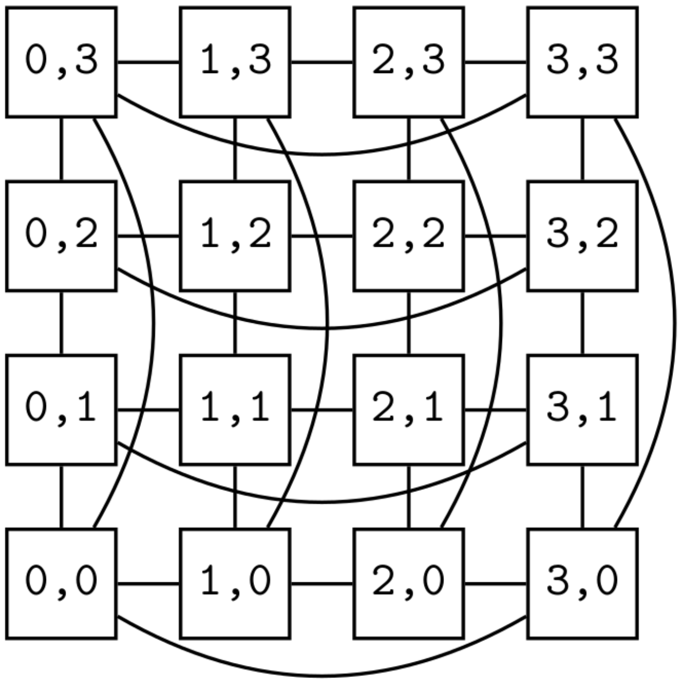

Figure 2.2: Example of a 4×4 2D torus (4-ary 2-cube).

We will now present some relevant network topologies and characterise their properties with a special focus on scalability.

2.2.1 n-Dimensional torus/mesh

In a n-dimensional torus having k nodes along each dimension, referred also as k-ary n-cube, there are $N = k^n$ nodes, each one connected to $2n$ neighbours, as depicted in Figure 2.2, therefore the nodes can be imagined as distributed on an n-dimensional grid[4]. The nodes at the boundaries of the grid are connected across the borders of the network providing: periodic boundary conditions, *symmetry* and *regularity*. The diameter of the network depends on the shape of the grid and on the torus dimensions, for an n-dimensional cubic grid of N nodes the diameter is:

$$d = n\frac{\sqrt[n]{N}}{2} = n\frac{k}{2} \tag{2.1}$$

[4]The network topology specifies only the connection between the nodes and not their position in space.

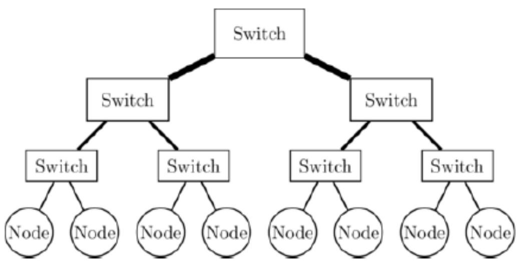

Figure 2.3: Fat-tree topology

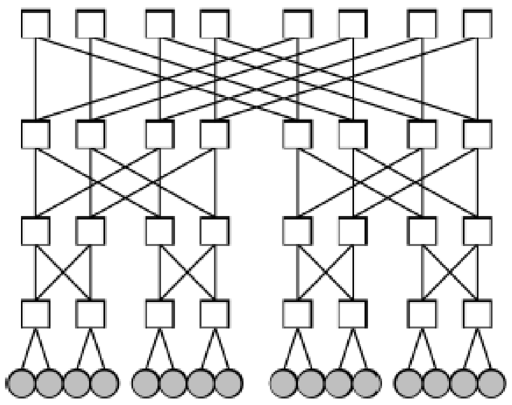

Figure 2.4: Folded-Clos (Fat-tree) network

From a scalability point of view this topology has a fixed node degree and a variable diameter, increasing the number of dimensions of the network provides a better scalability at the cost of more connectivity. Thanks to its fixed degree this topology is widely used in HPC's direct networks since adding more nodes does not change the structure of the node itself.

Meshes are similar to tori but they have open boundary conditions, so the network is *asymmetric* and *irregular*. The degree of the nodes is not constant, the diameter is $n(\sqrt[n]{N} - 1)$.

2.2.2 Fat-tree and Folded-Clos

Fat-tree topology was proposed by Leiserson in 1985 as an interconnection network for general-purpose parallel supercomputers with provably efficient communication [48]. In a tree topology, the traffic loads on the links increase moving towards the root: the main idea behind fat-tree network is to have higher capacity links as we approach the root, as shown in figure 2.3. To implement this concept, fat-tree networks requires switches with an increasing number of ports, to support more links, moving from leaf nodes to the root. This translates into a high cost of deployment when scaling the number of nodes, since high-radix switches are expensive devices.

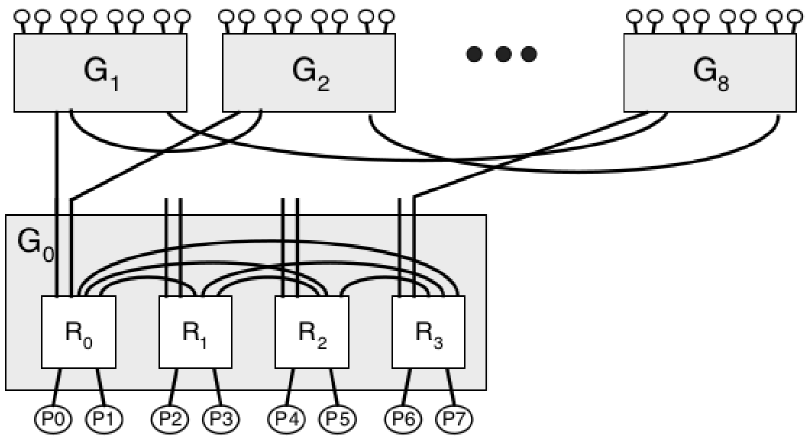

Figure 2.5: Fully connected topology example for a dragonfly network.

To overcome this limitation to the scalability of the network, a variation of the fat-tree can be build with a Folded-Clos [49] topology, using smaller radix switches, as shown in Figure 2.4. With this approach, it is possible to replace each high-radix switch close to the root with a set of low-radix switches, that logically behaves like a high-radix one.

2.2.3 Dragonfly

This topology aggregates routers in an efficient way to make them behave as higher radix ones [2]. The hierarchical structure of the network is composed by three levels: router, group and system. The *intra-group* and *inter-group* interconnection network topologies can be selected to achieve the system requirements.

In a dragonfly network we have:

- p Terminals connected to each router using *terminal channels*[5]
- N Network terminals
- a Routers in each group

[5]This connection can be done internally for a direct network or externally using a port of the switch for an indirect network.

- h Channels within each router used to connect to other groups
- g Groups in the system

Every router is connected to p terminals, $a - 1$ local channels and h global channels, therefore the radix of every router is $k = p + a + h - 1$. Because every group consists of a routers connected via intra-group channels it can be considered as a virtual router with an effective radix $k' = a(p + h)$. This very high radix ($k' >> k$) enables the system level network to be realized with very low global diameter, defined as the maximum number of expensive global channels on the minimum path between any two nodes. The parameters a, p and h can have any value; however, to get the optimal load balancing of the network traffic the parameters should respect the following relation:

$$a = 2p = 2h \tag{2.2}$$

This ratio is derived from the fact that each packet traverses two local channels along its route (one at each end of the global channel) for one global channel and one terminal channel. The basic topology is depicted in Figure 2.5 and uses a fully connected topology for both the inter-group and intra-group networks: if we make this particular choice the network is *regular* and *symmetric*. The *diameter* of the network is 3 for a direct configuration and 5 for an indirect one using the relation (2.2) for the parameters with $p = h = 2$, $a = 4$ the networks scales to $N = 72$ with $k = 7$ routers. By using virtual routers, the effective radix is increased from $k = 7$ to $k' = 16$. Different topologies can be used to reduce the radix of the nodes at the cost of an increase in network *diameter*.

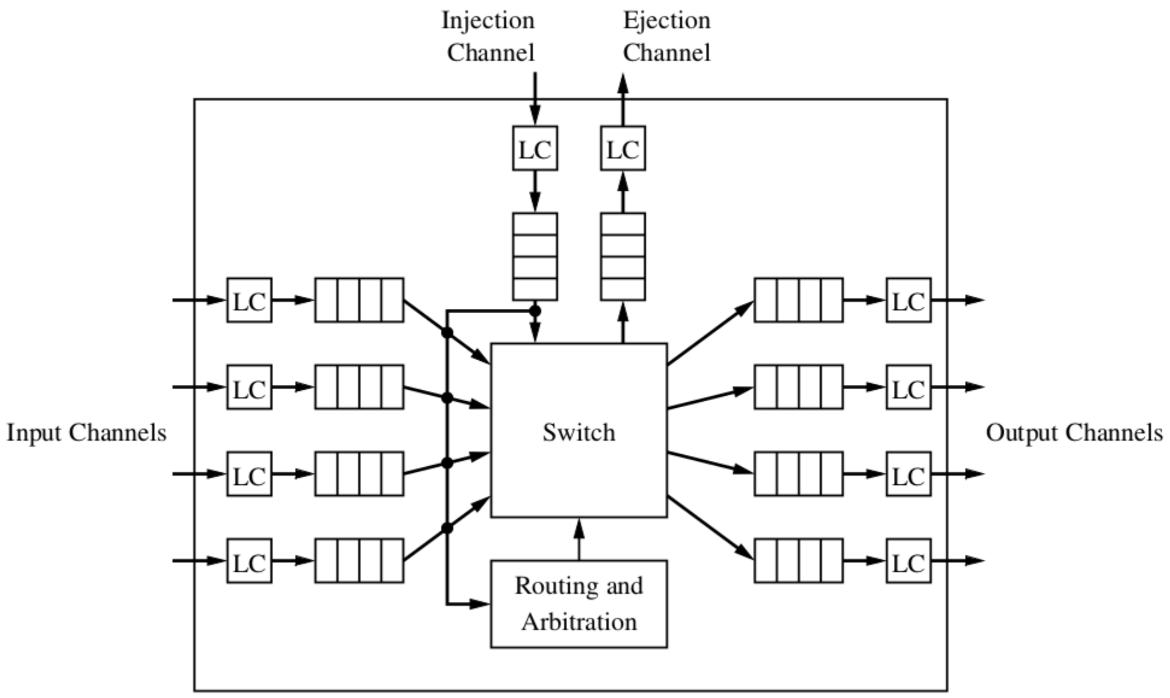

Figure 2.6: Internal structure of a generic router model (LC = link controller). Original image from [3]

2.3 Router model

Before going any further in this brief introduction to interconnection networks it is fundamental to define and describe a router model. A comprehensive and clear model is the one proposed in [3] which is a good representation of the real hardware architecture of a router. The internal architecture shown in Figure 2.6 is divided into the following main components:

- **Buffers:** *FIFO* buffers are used for storing messages in transit. The model in Figure 2.6 has buffers for input and output channels but alternative designs may have input or output buffers only.
- **Switch:** This component connects input buffers to output buffers.
- **Routing and arbitration unit:** Those components implement the routing algorithm, selecting the appropriate output channel for an incoming message and setting the switch accordingly. If the same output chan-

nel is requested by multiple messages at the same time the arbitration unit must resolve the contention. The arbitration policy can be a simple round robin or a complex priority algorithm.

- **LC (Link Controller):** It implements the control flow logic for the messages across the channel between two routers.
- **Processor interface:** This component is a channel interface to the local processor instead of an adjacent router. It consists of one or more injection and ejection channels.

This router model is designed for direct networks but can be used as a switch model for indirect networks if we remove the processor interface.

2.4 Packets, Flits, Phits

In order to transfer data over the network, they must be prepared and they may be split in chunks. This process is called packetization and it is done in the following way: the *message* to be transferred over the network is split in to chunks of a fixed size and all of the chunks are then encapsulated into *packets*. Packets are made of two parts a data one called *payload* and a protocol part called *header*.

The *packet* is the smallest unit of information that can be sent over the network, therefore the *header* must contain all the addressing data needed to deliver the payload to its destination; this information is crucial for the delivery, so the *header* is stored into the first part of the packet. Other useful protocol information, like error correction codes, can be stored into the optional *footer* which is stored into the last part of the packet.

The *packet* is then divided into smaller sub-units called *flits* (flow-control units), those units are the ones whose transfer is requested by the sender and acknowledge by the receiver. *Flits* may be divided into *phits* (physical units)

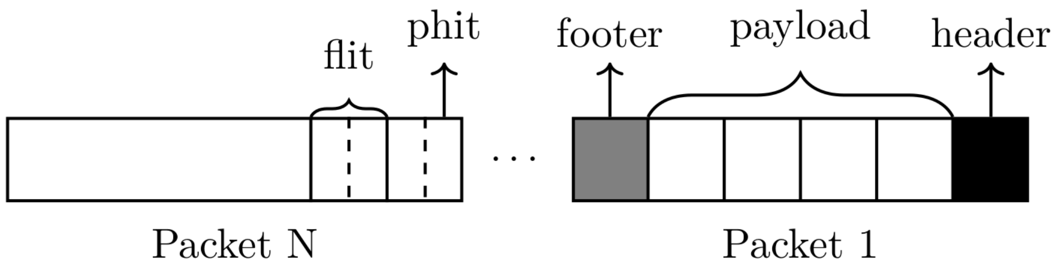

Figure 2.7: Fragmentation of a message into packets, flits and phits.

which are the units of information that can be physically transferred in a single cycle between two nodes. All the different sub-units of information are depicted in Figure 2.7.

2.5 Packet Switching Techniques

A crucial aspect of a network infrastructure is how the packets are switched and saved into intermediate nodes during their path. In both direct and indirect networks some switching action is required but we have not yet defined a policy for switching the packets. In this section three of the main switching techniques are presented: *store and forward*, *wormhole* and *virtual cut through*. Whatever policy we decide to adopt, we need to manage a critical physical resource: the receiving/sending buffers. Every switching technique has to decide how to manage the free space in the buffers and when to start and stop forwarding packets. When the network is in a congested state, a large fraction of the packets are waiting for resources.

2.5.1 Store And Forward (SAF)

This switching technique is very simple and works as follows:

- The node receives al the flits of the packet and stores them into the receiving buffer of the channel.

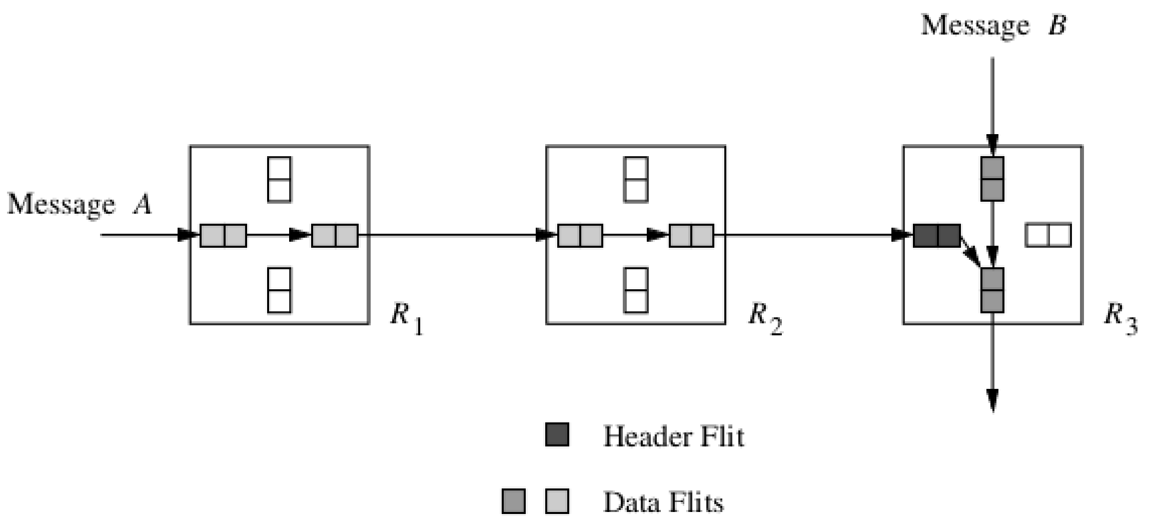

Figure 2.8: An example of messages travelling through a wormhole network. Message A is blocked by message B generating a contention. Original image from [3]

- The header is parsed by the router and the output port is calculated.
- If the receive buffer of the next node has enough free space to store the full packet and the channel is free the packet is forwarded, otherwise it waits.

This approach let the router deal only with complete packets. The main drawback of this switching technique is latency: at every hop the switch has to wait for the full transfer of the packet before forwarding the information. In absence of congestion the latency L of a packet is given by the (2.3) where: n_{flits} is the number of flits of the packet, n_{hops} is the number of hops between source and destination and t_{flit} is the time needed to transfer a single flit[6].

$$L = t_{flit} \cdot n_{flits} \cdot n_{hops} \tag{2.3}$$

2.5.2 Wormhole

This technique is the opposite of the SAF one. The switch forwards the packet a flit at the time as depicted in Figure 2.8 following this schema:

- The first flits of the packet containing the header are received and stored into the receiving buffer.
- The header is parsed by the router and the output port is calculated.
- If the receive buffer of the next node has enough free space to store a single flit and the channel is free the first flit is forwarded, otherwise it waits.
- Every new flit is forwarded as soon as available if the buffer and the channel are free.

In order to take the routing decision the router has to parse the entire header, this makes crucial keeping the header into the smallest possible number of flits, preferably one. The main advantage of this approach is the reduced latency by pipelining of the full packet transmission time across the hops. The latency L in absence of congestion can be calculated as follows:

$$L = t_{flit} \cdot (n_{flit} + n_{hops}) \tag{2.4}$$

Wormhole routing requires a more complex flow control system which has to be able to stop and resume the transfer of the packet if congestion occurs, and it is more prone to congestion because a packet uses resource of multiple nodes at the same time.

[6]We are neglecting the time spent by the router to make the routing decision, this time is common to all the switching techniques and therefore not interesting in this comparison.

2.5.3 Virtual Cut-Through (VCT)

In this case we want to take the best from both wormhole and SAF and combine them into a single technique. A VCT network behaves like a wormhole one apart from the forwarding requirements: to forward a packet the receiving buffer must have enough free space to store the full packet. In this way we have the same flow control simplicity and resources occupancy as in a SAF network but the same latency of a wormhole network in absence of congestion. The main drawback is an higher buffer capacity requirement than a wormhole network: in a wormhole network the buffers must provide at least enough free space to store a single flit; in VCT and SAF networks the buffers must provide at least enough free space for a full packet.

2.6 Routing algorithms

A *routing algorithm* defines which route a packet should take while travelling from its source to its destination. The selection of the algorithm can heavily affect network performance, therefore we have to select it carefully. In this section we will discuss the minimum requirements for a routing function and we will discuss different approaches to the problem of routing a packet.

It is useful to give some basic definitions. A *routing function* calculates the next step that a packet has to take in its path towards its destination: in principle we could use information gathered across the whole network to calculate the path but, if we want to achieve a good scalability of the system, this approach is not practical. In this work we will consider routing functions that use only local information, coming from the packets, the node, and its first-neighbour nodes.

The routing action is modelled as a two stage process implemented by two separated functions: routing and selection. The routing function provides a set of output channels based of the current node and the destination of the packet. The selection among the provided channels is made by the selection function based on the status of the output channels of the current node. To be selected a channel must be free according to the switching technique in use, if all the channels are busy the packet will wait for an available one. Because the routing function selects the subset of output channels it will be the only part responsible for granting the delivery of the packet, while the selection function will only affects performance. In this routing model the domain of the routing function is NxN for an N nodes network, because the function considers only the destination and the current nodes.

A routing function is *connected* if any node of the network can reach any other node, and because we want all the packets to be delivered on the network, every routing function must be *connected.* The routing function can select always the same path between two nodes or can choose between multiple ones according to the network traffic, the algorithms in the first case are called *deterministic*, the ones in the second case are called *adaptive*. An algorithm is *fully adaptive* if has the possibility to use all the possible paths. A routing function is *minimal* if it only supplies channels that bring the packet closer to its destination.

The minimum requirement for a routing algorithm is to deliver packets in a finite amount of time regardless of the network traffic, an in-depth study of this condition will be done especially for what concerns *deadlock* and *livelock*. Before introducing the problem we need to introduce one useful instrument to solve it: virtual channels.

2.6.1 Virtual channels

In section 2.5 we learned that the critical resources while sending packets over a network are the receiving buffers. A common strategy consists of optimizing the channel usage and reducing congestion using virtual channels [50]. Virtual channels consist on a set of duplicated buffers multiplexed and demultiplexed into the same physical channel by the link control logic. From the switch point of view there is no difference between a physical channel and a virtual one because the switch sees only the buffers, all the physical layer is handled by the link controller. By adding a virtual channel we can effectively increase the number of channels in the network without adding the extra cost and complexity of the real hardware link. All the virtual channels share the bandwidth of the physical link with a certain policy, for example round robin. If we select a non uniform policy it is possible to implement QoS into the network using several virtual channels with different priority levels. In this work virtual channels will be used only to implement deadlock-free routing.

2.6.2 Deadlock

A deadlock occurs when some packets cannot advance toward their destination because the buffer requested by them are full. A *configuration* is an assignment of a set of packets or flits to each buffer. A deadlocked configuration occurs when some packets are blocked for an infinite amount of time because they are waiting for resources that will never be granted because they are held by other packets. This condition must be avoided since it breaks the functionality of the network itself.

A deadlocked configuration is called *canonical* if all of the packets present into the network are blocked. We will study only canonical configurations because any other deadlocked configuration has a corresponding canonical one. To obtain a canonical deadlock configuration it is sufficient to stop injecting new traffic into the network and wait for the delivery of all the non blocked packets.

The presence or the absence of potentially deadlocking configurations is a property of the routing algorithm in use. The same topology with the same number of virtual channels may be deadlock-free or not depending on the selected routing algorithm. On the other hand this property is independent of the network traffic and must be verified without making any assumption on the traffic pattern.

To avoid deadlock we will focus on two opposite approaches: the detection of and the reaction to a deadlock configuration, and the impossibility of deadlock configurations to occur. The first method requires the network to be aware of being in a deadlocked state and to react breaking the deadlock condition. If we want to certainly detect a deadlock configuration we need to use non local information which is usually difficult to collect and analyse in large networks. If we want to use only local information we can guess if a packet is deadlocked by using a *time-out* system. This approach can lead to misidentification of congested packets into deadlocked ones if the network is heavily congested, leading to performance degradation. The reaction to a detected deadlock requires the capability of retransmitting packets that need to free resources. The network must detect and react to the deadlocked configurations faster than they occur, otherwise different deadlocked configurations can pileup generating network malfunctions. The second method consists of providing a routing algorithm that cannot generate deadlocked configurations. Because this property must be traffic-independent it is not trivial to proof that a routing algorithm is deadlock-free, but this problem can be approached using some graph theory and a bunch of theorems.

Definitions We introduce now the concept of *channel dependency* [4] and the *channel dependency graph*: there is a dependency between channels i and j, if the routing algorithm can forward a packet holding resources on channel i to channel j; the channel dependency graph is a directed graph $D = G(C, E)$ which vertices are all the unidirectional channels in the network I and arcs represents the pairs (c_i, c_j) such that there is a channel dependency from c_i to c_j.

We define the *routing subfunction* R_1 for a given routing function R as a routing function defined on the same domain as R that supplies a subset of the channels supplied by R. The channels supplied by R_1 for a given packet destination are indicated as *escape channels* for the packet; let's refer to the set of the escape channels with C_1. The *extended channel dependency graph* of R_1 is defined as the channel dependency graph in which all vertices belong to C_1, the graph is constructed taking into account a wider class of dependencies between channels (direct, indirect, direct cross and indirect cross) [51]. In the case of VCT and SAF networks, only direct and direct cross dependencies must be considered in building the extended channel dependency graph.

Necessary and Sufficient Condition The previous definitions finally allow us to enunciate the following important theorem[51] proposing a necessary and sufficient condition for a routing function to be deadlock-free:

Theorem 1. *A connected routing function R for an interconnection network I is deadlock-free if and only if there exists a routing subfunction R_1 that is connected and has no cycles in its extended channel dependency graph.*

If the routing algorithm is deterministic the only connected subfunction is R itself, so in this case the channel dependency graph and the extended channel dependency graph of R_1 are identical. Therefore the resulting condition for deadlock-free routing for non adaptive routing can be stated in the following corollary, originally proposed as a theorem [4]:

Corollary 1. *A deterministic routing function R for an interconnection network I is deadlock-free if and only if there are no cycles in its channel dependency graph D.*

This corollary can be used to generate deadlock-free deterministic routing functions, as in the following example. Consider a ring network of four nodes connected by unidirectional channels, as the one depicted in Figure 2.9(a), its

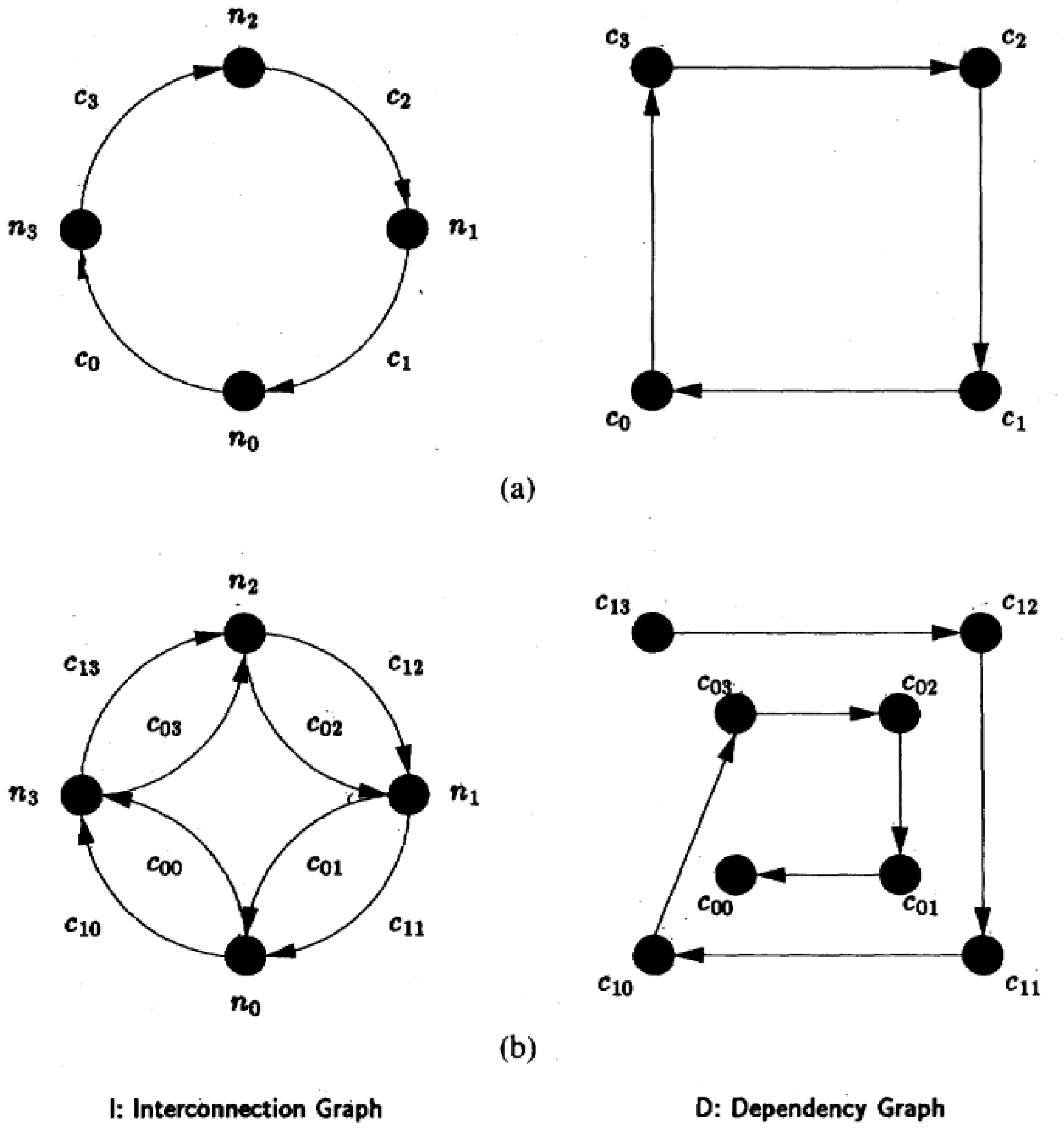

Figure 2.9: (a) Unidirectional ring (left) with its channel dependency graph (right). (b) Modified deadlock-free unidirectional ring (left) with its channel dependency graph (right). Original image from [4]

Algorithm 1 : Deadlock-free routing for unidirectional ring using high and low virtual channels

```
1: if $curr = dest$ then
2:     $Channel := Internal;$
3: else if $curr < dest$ then
4:     $Channel = c_{1curr}$
5: else
6:     $Channel = c_{0curr}$
7: end if
```

channel dependency graph D is shown on the right. Because of the simplicity of this network the only connected routing function is the one that forwards the packet through the only available channel if the destination is different from the current node. The channel dependency graph for this algorithm is not acyclic, therefore the network is not deadlock-free. In this trivial example it is very easy to find a deadlocked configuration: any configuration with all the buffers from all the nodes filled up with packets and zero packets arrived at destination is deadlocked.

We can make this network deadlock-free by splitting each physical channel into two groups of virtual channel: high virtual channels (c_{10}, ..., c_{13}) and low virtual channels (c_{00}, ..., c_{03}) as shown on the left of Figure 2.9(b).

The routing function 1 and the introduction of two virtual channels per physical channel modify the channel dependency graph into the one depicted on the right of Figure 2.9(b) which is acyclic, therefore the network is now deadlock-free.

Sufficient Condition for VCT and SAF Switching Networks If we focus on VCT and SAF switching networks, only direct and direct cross dependencies must be considered in building the extended channel dependency graph [51]. Direct cross-dependencies are due to the fact that some channels are used either as escape channels or as regular channels depending on packet destination. To avoid this situation, we can define the R_1 subfunction using the C_1 subset of channels as follows:

$$R_1(x,y) = R(x,y) \cap C_1 \forall x,y \in N \tag{2.5}$$

With this definition of R_1 a channel that belongs to C_1 will be used always as an escape channel for all possible destinations for which it can be supplied by R, so the cause of direct cross-dependencies is eliminated: the channel dependency graph and the extended channel dependency graph of the routing subfunction are identical. This allows to define the following corollary, that provides a direct manner to check whether a routing function is deadlock-free (sufficient condition):

Corollary 2. *A connected routing function R for an interconnection network I is deadlock-free if there exixts a channel subset $C_1 \subseteq C$ such that the routing subfunction $R_1(x,y) = R(x,y) \cap C_1 \forall x,y \in N$ is connected and has no cycles in its channel dependency graph D_1.*

Finally, the following theorem proposes a relevant property of routing subfunctions defined according to expression 2.5:

Theorem 2. *A connected routing function R for an interconnection network I is deadlock-free if there exists a channel subset $C_1 \subseteq C$ such that the routing subfunction $R_1(x,y) = R(x,y) \cap C_1 \forall x,y \in N$ is connected and deadlock-free.*

Duato's Protocol for VCT or SAF Networks Theorem 2 is at the basis of a methodology, referred as Duato's Protocol (DP), that provides fully adaptive minimal and nonminimal routing algorithms, starting from a deadlock-free connected routing algorithm. Given a VCT or SAF switching interconnection network I_1, the DP is accomplished in two stages:

1. Select one of the existing routing functions for I_1, let it be R_1. R_1 must

be deadlock-free and connected, it can be deterministic or adaptive. C_1 is the set of channels supplied by R_1.

2. Split each physical channel into a set of additional virtual channels, C is the set of all the virtual channels in the network. C_{xy} represents the set of output channels from node x that belongs to a path from node x to node y. The new routing function R can then be defined as:

$$R(x,y) = R_1(x,y) \cup (C_{xy} \cap (C - C_1)) \forall x, y \in N \tag{2.6}$$

So the new routing function can use any of the added virtual channels or, in alternative, the channels supplied from the starting routing function $R1$, but new routing options to C_1 are not allowed.

In this work we followed the DP methodology to guarantee that DQN-Routing is deadlock-free, as we will see in chapter 3.

2.6.3 Livelock

Livelock is a misbehaviour of the routing algorithm which forwards packets along a path that does not contain their destination, resulting in not delivered packets and wasted network bandwidth. Any minimal routing function is automatically livelock-free, the distance d between any couple of nodes into the network is a finite quantity, therefore every minimal path algorithm delivers any packets in d steps. For non minimal algorithm extra attention to avoid livelock must be paid, especially if the algorithm uses only local information to calculate the path.

A simple technique to avoid livelock, that has been used for this work, is to limit the number of non minimal hops that a packet can take; this limitation reduces the flexibility of the algorithm but avoids infinite looping.

2.7 Selection of Routing Algorithms

In this section several routing algorithms for tori and dragonfly will be presented. For each algorithm we provide proof of the absence of livelock and deadlock.

2.7.1 Dimension-Order Routing (DOR)

DOR [4] is a minimal non adaptive routing algorithm for n-dimensional tori and meshes. Because of the simplicity of this algorithm it is often used as a base for more complex ones.

The position of the nodes in the network is indicated by n integers giving the Cartesian coordinates of the node into the grid. The algorithm starts comparing the node coordinates against the packet destination coordinates from dimension 0 up to dimension $n-1$. If the coordinates are different the packet is forwarded following the minimal path to the destination, if the coordinates are equal no action is required: the packet has arrived to its destination. The critical point of the algorithm is to manipulate the n dimensions in a fixed order[7]. Figure 2.10 depicts an example of path selected by the dimension-order algorithm on 2D torus.

The algorithm uses minimal paths so it is livelock-free by definition. Before considering the deadlock freedom of the algorithm it is useful to divide the channels into classes:

- A channel belongs to the class d^+ if it connects two node in the dimension d in the increasing coordinates direction.
- A channel belongs to the class d^- if it connects two node in the dimension d in the decreasing coordinates direction.

[7]The order relation can be any total one and it can be selected only once. To change the order relation the network has to be completely empty.

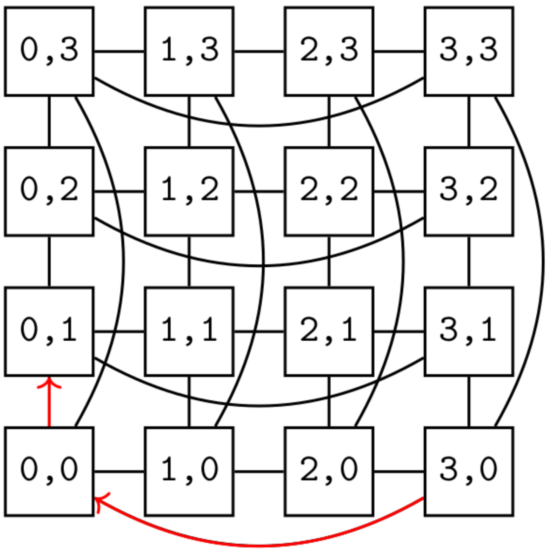

Figure 2.10: Path of the DOR algorithm on a 4x4 torus between nodes (3,0) and (0,1).

To proof the absence of deadlock configurations, we can use the corollary 1 and discuss the channel dependency graph. Thanks to the dimension order any channel of class $i^{\pm}$ cannot depend on any channel of the class $j^{\pm}$ where $j > i$, this removes all the possible loops between channels belonging to different dimensions. The algorithm selects the minimum path to the destination, for a given dimension d the shortest path can be either in the direction $+$ or in the direction $-$ but it cannot be in both directions. Therefore it is impossible to generate dependencies between channels d^{+} and d^{-} in the same dimension. The only possible source of cyclic dependency left is within channel of the same class. If we consider a mesh network, packets are not allowed to cross the boundaries of the network, preventing deadlock. The torus topology requires more attention because the algorithm is not deadlock-free in the form presented before. A single channel class of a torus is exactly the same topology as the circular network depicted on the left in Figure 2.9(a) , therefore adding one extra channel dedicated to the packets that have crossed the border is enough to remove any deadlocking situation. The extra virtual channels are only required for half of the nodes in the class. In the real implementation it is good practice to keep the nodes symmetric and add the extra virtual channel for all the nodes in the class.

2.7.2 Star-channel [1]

This is a minimal path, fully adaptive routing algorithm for n-dimensional tori and meshes based on the dimension-order routing. This algorithm adds an extra virtual channel to the ones used by the DOR to achieve full adaptivity. The idea behind this routing function is to use the DOR as an *exhaust valve* for any possible deadlock configuration originated by the adaptive behaviour.

The network is divided into two sub-networks: one is the *star* subnet made of all the channels used by the dimension-order algorithm, the other is the *nonstar* one which has one virtual channel on every link[8]. The routing algorithm uses the two subnetworks according to the following rules:

- A packet can use a free channel of the *nonstar* subnet to follow a minimal path to its destination in any dimension.
- A packet can use a *star* channel only if it respects the dimension order criterion.

Livelock can be excluded because the algorithm uses minimal paths. To exclude deadlock we can use the channel dependency graph. Packets can only use the *star* channels if they meet all the dimensional requirements imposed by the algorithm. Because the dimension-order routing function has an acyclic graph, the extra dependency added by the presence of the *nonstar* subnet cannot generate deadlock configurations. For VCT and SAF switching networks, we can use theorem 2, where C_1 is the subset provided by the

[8]Every algorithm can have more virtual channels to implement QoS or other traffic control features, here we are discussing the minimum requirements for deadlock and livelock free routing.

$$\begin{tikzcd}
& G_6G_3 \arrow[r] & G_3R_3R_0 \arrow[dr] & \\
G_6R_0R_1 \arrow[ur] & & & G_3G_0 \arrow[dl] \\
& G_0G_6 \arrow[ul] & G_0R_1R_2 \arrow[l] &
\end{tikzcd}$$

(a) Standard network

$$\begin{tikzcd}
G_0R_1R_2V & G_3G_0 \arrow[l] & G_3R_3R_0 \arrow[l] \\
G_0R_1R_2 \arrow[r] & G_0G_6 \arrow[r] & G_6R_0R_1V \\
G_6R_0R_1 \arrow[r] & G_6G_3 \arrow[r] & G_3R_3R_0V
\end{tikzcd}$$

(b) Network with extra virtual channels

Figure 2.11: Subsets of the total dependency graph for a fully connected dragonfly network. Graph 2.11a depicts a standard configuration with no additional virtual channels. Graph 2.11b depicts a deadlock-free configuration with extra virtual channels (identified by a V letter) for local routing of packets coming from other groups.

star channels and the associated routing subfunction R_1 is the connected and deadlock-free DOR.

2.7.3 Min-routing [2]

This is a near-minimal non adaptive routing algorithm for dragonflies and it can be considered as a base algorithm for adaptive or more complex ones.

Packets in a dragonfly network can be identified by three number:

- **Group index (G):** it indicates the group
- **Router index (R):** it indicates the router within the group
- **Node index (N):** it indicates the node within the router

The min-routing algorithm uses a three step process to forward the packet using the corresponding local or global channel, for better clarity the algorithm can be described using this pseudo code:

```
if(G_node != G_destination)
  if(is_connected(G_node, G_destination))
    forward_g(G_destination);
  else
    forward_l(global_to_local(G_destination));
else if( R_node != R_destination )
  forward_l(R_destination);
else
  forward_i(N_destination);
```

The function *is_connected* returns true, if the node has a direct global connection to the given group, false otherwise. The various *forward* functions forward the packet using global, local or internal port accordingly. The function *global_to_local* returns the index of the local node which as a global direct connection to the provided group. This method provides a fully connected near-minimal routing function.

The algorithm is distance-bounded and therefore livelock-free. If no virtual channels are used deadlock configuration can occur because there are cyclic dependencies. We can easily create a loop between ingoing and outgoing packets between different groups, as depicted in Figure 2.11a because the same local channels are used for all the packets. In order to break those loops we need to add one extra virtual channel to the local network dedicated to packets coming from a different group. If we consider now the channel dependency all the loops are broken as shown in Figure 2.11b. Note that the dependency graphs are partial to provide a better understanding of the critical path, the graph in Figure 2.11b is not connected but this is due to the partial nature of the graph itself and it does not imply a not connected routing function.

2.8 The APEnet Interconnection Network and the ExaNest Project

In this section we will introduce the APEnet interconnection network and ExaNet, its latest implementation in the context of the H2020 European Project ExaNeSt [47].

2.8.1 APEnet

The Array Processor Experiment (APE) is a custom design for HPC, started by the Istituto Nazionale di Fisica Nucleare (INFN) and partnered by a number of physics institutions all over the world; since its start in 1984, it has developed four generations of custom machines (APE [52], ape100 [53], APEmille [54] and apeNEXT [55]). Leveraging the acquired know-how in networking and re-employing the gained insights, a spin-off project called APEnet developed an interconnect board based on FPGA that allows to assemble a PC cluster *à la* APE with off-the-shelf components.

The design of APEnet interconnect is easily portable and can be configured for different environments: (i) the APEnet [56] was the first point-to-point, low-latency, high-throughput network interface card for Lattice Quantum ChromoDynamics (LQCD) simulations dedicated clusters; (ii) the Distributed Network Processor [57] (DNP) was one of the key elements of RDT (Risc+DSP+DNP) chip for the implementation of a tiled architecture in the framework of the EU FP6 SHAPES project [58]; (iii) the APEnet Network Interface Card, based on an Altera Stratix IV FPGA, was used in a hybrid, GPU-accelerated x86_64 cluster QUonG [59] with a 3D toroidal mesh topology, able to scale up to $10^4 - 10^5$ nodes in the framework of the EU FP7 EURETILE project. APEnet+ was the first device to directly access the memory of the NVIDIA GPU providing GPUDirect RDMA capabilities and experiencing a boost in GPU to GPU latency test; (iv) the APEnet network IP — *i.e.* routing logic and link controller — is responsible for data transmision at Tier 0/1/2 in the framework of H2020 ExaNeSt project, as we will see later in the section.

Table 2.1 summarizes the APEnet families comparing the main features.

	APEnet	**DNP**	**APEnet+**	**APEnet+ V5**	**ExaNet**
Year	2003	2007	2012	2014	2017
FPGA	Altera Stratix III	ASIC	Altera Stratix IV	Altera Stratix V	Xilinx Ultrascale+
BUS	PCI-X	AMBA-AHB	PCIe gen2	PCIe gen3	AXI
Computing	Intel CPU	RISC+DSP	NVIDIA GPU	NVIDIA GPU	ARM+FPGA
Link Bandwidth	6.4 Gbps		34 Gbps	45 Gbps	32 Gbps
Latency	6.5μs		4μs	5μs	1.1μs

Table 2.1: The APEnet roadmap to Exascale

The APEnet interconnect has at its core the DNP acting as an offloading network engine for the computing node, performing internode data transfers; the DNP has been developed as a parametric Intellectual Property library; there is a degree of freedom in choosing some fundamental architectural features, while others can be customized at design-time and new ones can be easily added. The APEnet architecture is based on a layer model, as shown in Figure 2.12, including physical, data-link, network, and transport layers of the OSI model.

The physical layer — **APEphy** — defines the data encoding scheme for the serialization of the messages over the cable and shapes the network topology. APEphy provides point-to-point bidirectional, full-duplex communication channels of each node with its neighbours along the available directions (*i.e.* the connectors composing the IO interface). APEphy is strictly dependent on the embedded transceiver system provided by the available FPGA. It is normally based on a customization of tools provided by the FPGA vendor — *i.e.* DC-balance encoding scheme, deskewing, alignment mechanism, byte ordering, equalization, channel bonding. In APEnet+ and APEnet+ V5, four bidirectional lanes, bonded into a single channel with usual 8b10b encoding, DC-balancing at transmitter side and byte ordering mechanisms at receiver side, allow to achieve the target link bandwidth (34 Gbps [60] and

45 Gbps [61] respectively).

The data-link layer — **APElink** — establishes the logical link between nodes and guarantees reliable communication, performing error detection and correction. APElink [62] is the INFN proprietary high-throughput, low-latency data transmission protocol for direct network interconnect based on word-stuffing technique, meaning that the data transmission needs submission of a magic word every time a control frame is dispatched to distinguish it from data frames. The APElink manages the frame flow by encapsulating the packets into a light, low-level protocol. Further, it manages the flow of control messages for the upper layers describing the status of the node (*i.e.* health status and buffer occupancy), and transmitted through the APElink protocol.

The network layer — **APErouter** — defines the switching technique and routing algorithm. The Routing and Arbitration Logic manages dynamic links between blocks connected to the switch. The APErouter applies a dimension order routing [4] (DOR) policy: it consists in reducing to zero the offset between current and destination node coordinates along one dimension before considering the offset in the next dimension. The employed switching technique — *i.e.* when and how messages are transferred along the paths established by the routing algorithm, *de facto* managing the data flow — is Virtual Cut-Through [63] (VCT): the router starts forwarding the packet as soon as the algorithm has picked a direction and the buffer used to store the packet has enough space. The deadlock-avoidance of DOR routing is guaranteed by the implementation of two virtual channels [50] for each physical channel.

The transport layer — **APE Network Interface** — defines end-to-end protocols and the APEpacket. The APE Network Interface block has basically two main tasks: on the transmit data path, it gathers data coming in from the bus interfacing the programming subsystem, fragmenting the data stream into packets — APEpacket— which are forwarded to the relevant

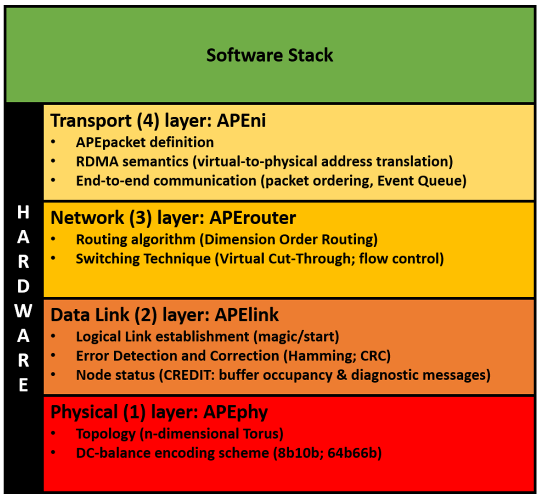
Software Stack
HARDWARE
Transport (4) layer: APEni
APEpacket definition
RDMA semantics (virtual-to-physical address translation)
End-to-end communication (packet ordering, Event Queue)
Network (3) layer: APErouter
Routing algorithm (Dimension Order Routing)
Switching Technique (Virtual Cut-Through; flow control)
Data Link (2) layer: APElink
Logical Link establishment (magic/start)
Error Detection and Correction (Hamming; CRC)
Node status (CREDIT: buffer occupancy & diagnostic messages)
Physical (1) layer: APEphy
Topology (n-dimensional Torus)
DC-balance encoding scheme (8b10b; 64b66b)

Figure 2.12: The layered architecture of APEnet

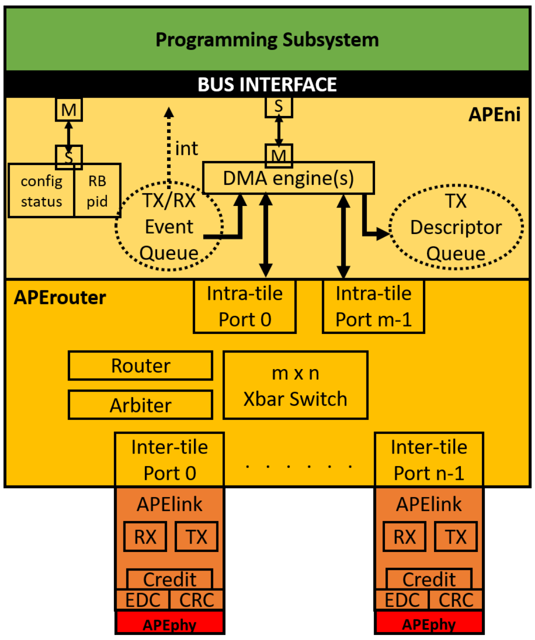

Figure 2.13: A block diagram of APEnet architecture

destination ports, depending on the requested operation; on the receive side, it implements PUT and GET semantics providing hardware support for the RDMA (Remote Direct Memory Access) protocol that allows to transfer data over the network without explicit support from the remote node's CPU.

An APEpacket is composed by a header (1 flit, 128 bit), a payload (up to 256 flits), and a footer (1 flit). For the format of the packet header and footer refer to Figure 2.14 and Figure 2.15.

The full APE Network Interface offers a register-based space for configuration and status signalling towards the host. Further, it offers a variable size region for defining a number of ring buffers, each one linked to an OS process accessing the device. These regions are typically accessed in slave mode by the host, which is master (read/write of single 32-bit based registers). Four DMA engines are used to move data to and from the device, plus other additional services: a TX descriptor queue (to issue buffer transfers from host to device) and an event queue (to notify different kind of completions to host).

Single or Multiple DMA engines could manage the same intra-tile port.

The block diagram of the APEnet interconnect architecture is shown in Figure 2.13.

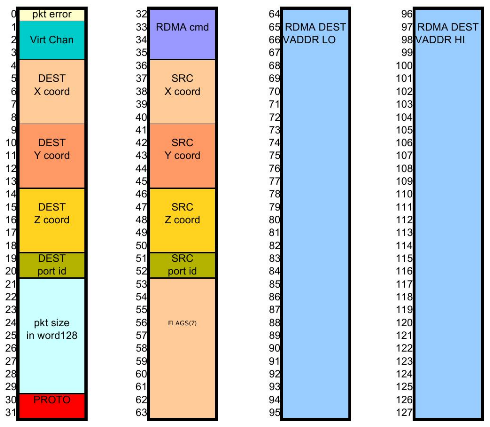

Figure 2.14: APEnet packet header

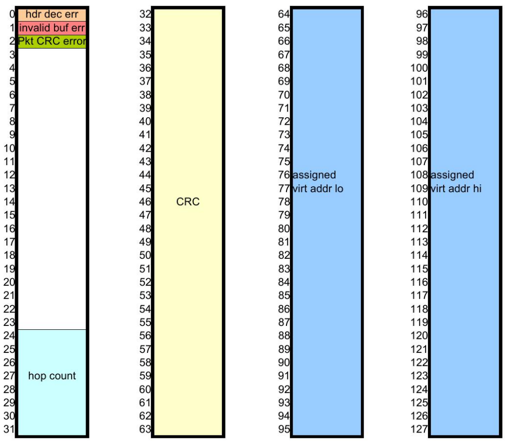

Figure 2.15: APEnet packet footer

2.8.2 The ExaNeSt project

The ExaNeSt project [64], started on December 2015 and funded by EU H2020 research framework (call H2020-FETHPC-2014, n. 671553), was a European initiative aiming at developing the system-level interconnect, a fully-distributed NVM (Non-Volatile Memory) storage and the cooling infrastructure for a Exascale-class supercomputer with heterogeneous computing nodes integrating multi-core ARM processors with FPGA reconfigurable devices.

One of the main goals of the project was the design and development of an interconnection network suitable for exascale-class supercomputers. The design envision a hierarchical infrastructure of interacting network layers. Design at the network level is configurable although topologies in the lowest layers (or tiers) are hardwired. An overview of the foreseen interconnects is in Table 2.2.

The *Unit* of the system is the Xilinx Zynq UltraScale+ FPGA, integrating four 64-bit ARMv8 Cortex-A53 hard-cores running at 1.5 GHz, and 2 ARM-

	Hierarchy	**Fanout**	**Switching**	**Topology**	**Bandwidth**	**Latency**
Tier 4	System	500 Racks	Optical			
Tier 3	Rack	3 chassis	10GbE (ExaNet)	Fat-Tree (Torus)	10 Gbps	
Tier 2	Chassis	9 mezzanines	ExaNet	2D/3D-Torus	4x10 Gbps	400 ns per hop
Tier 1	Mezzanine	4 nodes	ExaNet	Ring	2x10 Gbps	400 ns per hop
Tier 0	Node	4 FPGAs	ExaNet	All-to-All	16 Gbps	400 ns
FPGA	Unit	ZU9				
CORE		A53				

Table 2.2: The ExaNeSt multi-tiered network.

R5 cores.

The *Node* is the Quad-FPGA Daughter-Board (QFDB) containing four Zynq devices, 64 GB of DRAM and 250 GB SSD storage connected through the ExaNeSt Tier 0 network.

For inter-node communication, one FPGA provides a connector with ten bidirectional HSS links. Each external link has a maximum rate of 10 Gbps. Four out of ten links connect neighbouring QFDBs hosted on the *Mezzanine* (also known as Blade) (Tier 1). The Mezzanine board enables the mechanical housing of 4 QFDBs hardwired in a ring topology with two HSS links (2×16 Gb/s) per edge and per direction. The remaining six HSS links, routed through SFP+ connectors, are mainly used to interconnect mezzanines within the same Chassis (Tier 2).

3 DQN-Routing: a Novel Adaptive Routing Algorithm for Torus Networks Based on Deep Reinforcement Learning

We designed DQN-Routing, a novel, distributed, unicast, fully adaptive non-minimal routing algorithm for torus networks. Arithmetic routing is the traditional approach for torus networks: this is an efficient and scalable solution. Moving in the same direction, in the sense of relying on some kind of processing rather than on routing tables, we envisaged the possibility of delegating the adaptive part of the routing algorithm to a reinforcement learning Agent [7], using the deterministic dimension-order routing algorithm as escape routing subfunction (see theorem 2).

Let's refer to the reinforcement learning scenario shown in figure 3.1: the Router senses the network state, that makes up the Environment in RL terms, by observing a state S (that includes routing requests); it processes this state and provides an *interpreted* network state S' to the Agent, that in response propose an action (*i.e.* a routing decision) A' to the Router; depending on S, the Router decides to accept the Agent proposal ($A = A'$) or reject it ($A \neq A'$) and takes its routing decision A.

In an asynchronous manner, the Router receives, as part of S, feedback information for the actions $A = A'$ it has taken on Agent proposal. The Router uses this feedback information to provide a scalar *reward* R' to the

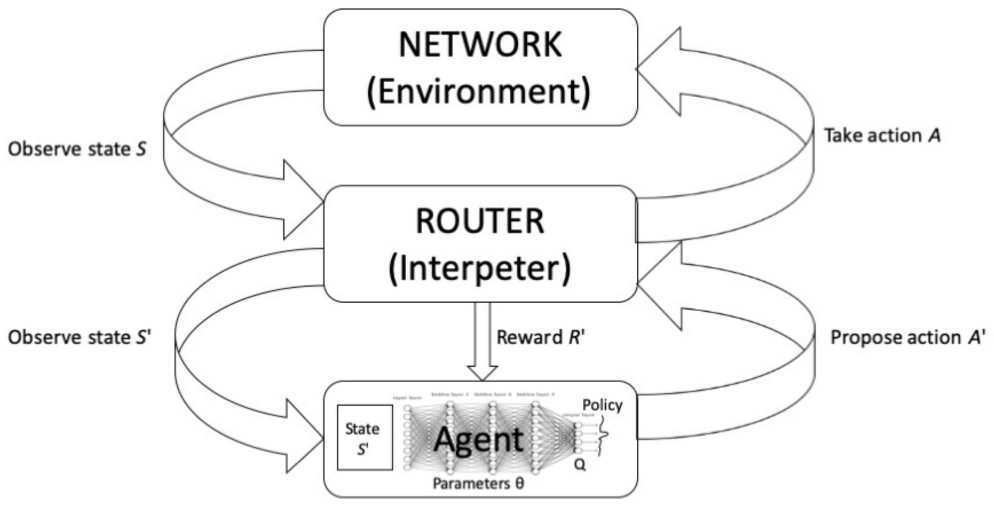

Figure 3.1: Reinforcement Learning scenario for DQN-Routing.

Agent for its proposed actions, the Agent uses R' to adjust its parameters in order to provide better proposals in future.

In DQN-Routing the Agent is implemented, according to the Deep Reinforcement Learning approach [8, 9], with a convolutional neural network trained with a variant of Q-learning (Deep Q-Network or DQN).

Although we focused on the view of a single Router plus Agent system, the whole routing problem can be represented as an independent multi-agent reinforcement learning problem, since the network interconnects many of such systems, that learn to act independently, without sharing experiences during the learning process.

3.1 Related Work

The seminal paper by Boyan&Littman (1993) [65] proposes *Q-Routing*, a distributed value-based RL algorithm for packet routing in networks with dynamically changing topology and traffic conditions. In *Q-Routing* each router does not consider global network information to take routing decisions or to learn and improve its policy, in this sense it is a *distributed* variant of the original Q-Learning value-based RL algorithm introduced by Watkins(1992) [66].

The main contribution of the paper from Peshkin&Savova [67] can be identified in applying a policy gradient ascent method for the problem of adaptive routing in packet-switched communication networks. This method is a type of reinforcement learning technique that relies upon optimizing parameterized policies with respect to the expected return (long-term aggregate reward) by gradient ascent. It extends previous results obtained by Williams [68] with the REINFORCE algorithm and by Baird and Moore [69] with the VAPS algorithm. The communication network (number and topology of nodes, status of the links, dynamic of packets) represents what, in reinforcement learning jargon, is called the environment. It is represented by a Partial Observable Markov Decision Process (POMDP) and its model is not known to the agents (the routers). The nodes (and the routers in the nodes – the agents) of the network are homogeneous. The agent performs a local observation of the destination of the handled packet, then performs an action forwarding the packet to link a according to a softmax rule (the policy) stored in a lookup table for any (destination, link) pair. This ensures that for a given destination, any of the links available will be used. After the packet has reached its final destination, a reward that depends on the total delivery time for the packet is assigned to the agent: this is the only global information that the agent has access to. Under these hypotheses, the presented algorithm is guaranteed to converge to a local optimum in policy parameters space.

Considering the inherent non-scalability of the tabular Q-learning approaches and the recent development of Deep RL techniques, the idea of using them in network routing was quite natural. To the best of our knowledge Valadarsky et al.(2017)[70] were the first to publish this idea, an algorithm based on Trust Region Policy Optimization (TRPO) [71], along with a research program to further investigate in this direction.

Mukhutdinov et el. (2018) [72] proposed an algorithm for packet routing in the logistics domain, such as material handling systems or automated traffic routing. Their approach is based on deep reinforcement learning networks combined with link-state protocol and preliminary supervised learning. The proposed algorithm is designed to run in a distributed fashion on a network on interconnected routers. The routing problem is schematized as a multi-agent reinforcement learning problem, with routers representing agents and modeled as deep neural networks. This choice allowed each router to account for heterogeneous data about its environment, and enabled the optimization of more complex cost functions, like in case of simultaneous optimization of bag delivery time and energy consumption in a baggage handling system. Application of the algorithm in simulated computer network and baggage handling system showed that it outperforms state-of-the-art routing algorithms.

You et al.[73] identify the curse of dimensionality of Q-routing as the limiting factor prohibiting a more comprehensive representation of dynamic network states, and thus limiting the potential benefit of reinforcement learning. They embedded deep neural networks in multi-agent Q-routing. Each router relies on its own neural network that is trained without communicating with its neighbors and makes its routing decision independently. They proposed two different multi-agent DRL-enabled routing algorithms: one replacing the Q-table of Q-routing by a deep neural network, the other employing additional information that include the past actions and the destinations of non-head of line packets. Results of simulations shows that the direct substitution of Q-table by a deep neural network may not yield minimal delivery delays since the neural network does not learn more from the same input data, while Adaptive routing policy can converge and significantly reduce the packet delivery time when a richer view on the network state is provided to the agents.

3.2 The DQN-Routing Algorithm

We designed DQN-Routing, a distributed, unicast, fully adaptive, non-minimal routing algorithm for n-dimensional torus networks.

Following to the Duato's protocol for deadlock-free adaptive routing algorithms introduced in paragraph 2.6.2, we selected the Dimension-Order Routing (DOR) for the torus interconnect having two virtual channels per physical channel, as our deadlock-free and connected routing subfunction R_1 and associated set of channels C_1. Then we added a new virtual channel per physical channel (*DqnChX-*, *DqnChX+*, ...): this new set of channels has been reserved for the fully adaptive, non-minimal routing actions proposed by the reinforcement learning Agent according to the scheme depicted in figure 3.1. The Agent cannot propose channels belonging to C_1.

DQN-Routing is based on two kind of traffic: *regular* packets, transporting data payload on the *forward path*, and *acknowledge* packets that are routed on the *reverse path* to provide feedback to all the routers involved in forward path. On acknowledge packet receive, the Router calculates the reward corresponding to its past routing decision proposed by the Agent using the timestamp contained in the acknowledge message travelling along the reverse path, and forwards it to the Agent. These rewards are used to guide the learning process of the Agent toward better performances, *i.e.* to improve its ability to propose routing actions that minimize on average the communication latency of the routed packets given the experienced network state.

Algorithm 2 describes DQN-Routing of regular packets; for the sake of clarity a 2-dimensional Torus interconnect is considered, but its extension to a highest dimension case is straightforward. DQN-Routing requires that regular packet must be uniquely identified with some kind of packet id; in section 3.3 we will discuss this requirement in more detail. The algorithm requires also that the data link layer of the network provides a mechanism for

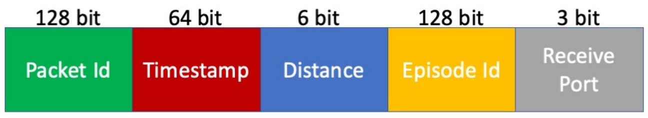

Figure 3.2: DQN-Routing table entry.

the router to have access to the virtual channel buffer occupancy of its first neighbour routers, as in the case of the APEnet interconnect. These quantities are indicated as *RemOccDqnChX-*, *RemOccDqnChX+*, *RemOccDqnChY-* and *RemOccDqnChY+* in algorithm 2.

When a packet header is received and packet must be forwarded since the local router is not its destination and it was not flagged as to be routed on a deterministic DOR path, the distance to the destination router is calculated and stored in a table entry in the Router memory along with the packet id, the timestamp, an episode id identifying the routing decision that will be proposed by the Agent for this packet, and the port from which the packet header was received.

Note that this last information does not take part in the routing decision, but is necessary to reconstruct the reverse path for the acknowledge packet, as it is described in algorithm 3. A suitable format of table entry for the APEnet+ interconnect is represented in figure 3.2. This table entry will be used during routing of acknowledge packet to produce a suitable reward for the Agent for the routing decision that it is going to propose for this packet, as shown in algorithm 3. The choice of a proper reward is of paramount importance in a reinforcement learning problem. In DQN-Routing we adopted the simple reward:

$$reward = distance/latency \tag{3.1}$$

Note also that as DQN-Routing is not minimal, so it will be possible to have multiple entries with the same packet id in the table.

The Router will then produce the observation of the network state S' for the Agent, consisting on the local router and destination router coordinates, the packet payload size and the level of occupancy of the local and remote buffers of the adaptive virtual channels, and will communicate it to the Agent which in response will propose a candidate channel to the Router. If the candidate output port is busy forwarding another packet and the remote buffer of the candidate virtual channel has not enough free space to store the entire packet payload, the candidate adaptive channel will be rejected and the packet flagged (*Pdor*) and escaped to the deterministic DOR routing, otherwise the channel proposed by the Agent will be used for routing.

A straightforward variation of algorithm 2 allows to limit the amount of traffic generated by acknowledge messages: it is possible to mark regular packets as triggering the generation of an acknowledge packet at destination router (*ReqAck* flag) and to start a new episode only every $AckRatio$ packets injected. An $AckRatio > 1$ parameter will correspond to a subsampling of the network feedback by the Router and will affect the time response of the learning process.

We can enforce the livelock-free property of the algorithm checking the number of *hops* of the packet being routed: if it is above a configurable threshold, that will be set considering the network diameter, the packet will be flagged (*Pdor*) and escaped to deterministic DOR routing.

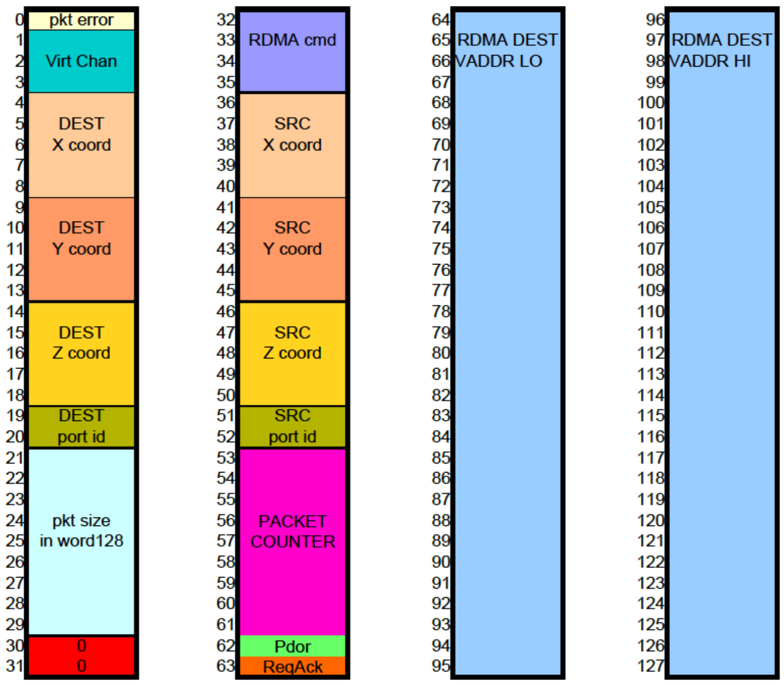

Figure 3.3: Header of a DQN-Routing regular traffic packet in the APEnet+ interconnect.

3.3 Considerations on the Implementation of the Algorithm

The actual implementation of the DQN-Routing algorithm poses some requirements both at the level of transport protocol as well as in the network infrastructure. The minimization of these additional requirements guided us during the design of the algorithm. Here we analyze them, discussing possible implementations having as a reference the APEnet+ interconnection network.

Transport Protocol The additional requirements in terms of the transport protocol are limited to the support in the regular packet header to carry the *Packet Id*, the *Pdor* (packet routed on escape DOR path) and *ReqAck* (regular traffic packet will trigger the send of an acknowledge packet at the destination router) flags, and the *Packet Type* (regular, acknowledge) information.

1. Ideally we would use a 128-bit UUID [74] for the *Packet Id* but this is an

impractical solution, as it would double the header size, from one to two 128-bit flits. Referring to the APEnet+ packet header (see Figure 2.14), a viable solution is to use the lower 9 unused bits in the *flag* field to store the value of the modulo 512 counting of injected packets. The whole header can then be used as *Packet Id*: in the very pathological situation when a node repeatedly send packets of the same size to the same node at the same remote virtual memory address and using the same virtual channel, this choice will at least guarantee a periodicity of 512 in packet ids. Furthermore the potential risk of mis-identification of the packet is mitigated by the possibility of choosing a $AckRatio > 1$. With this definition for *Packet Id*, lines 20-21 of algorithm 2 would be modified, and the 128-bit header would be used to set *Puid*.

2. The header must carry the *Pdor* and *ReqAck* flags: this information can be stored in otherwise unused higher two bits (62 and 63) of the *flags* field.

3. The *Packet Type* can be encoded in the *proto* field, currently encoding only the regular (RDMA) traffic, in bits 30 and 31 of the header.

The resulting DQN-Routing header format for the APEnet+ interconnect is represented in Figure 3.3.

Acknowledge packets must carry a timestamp and a packet id information. It is possible to implement this requirement with fixed size packets of 3 flits, a header carrying also the timestamp, as shown in figure 3.4, plus a payload flit containing the packet id and a footer.

Router Architecture During the design of DQN-Routing we strived to minimize the additional resources needed in the APEnet Router to implement it. The additional features are so limited to:

1. An additional virtual channel per physical channel.

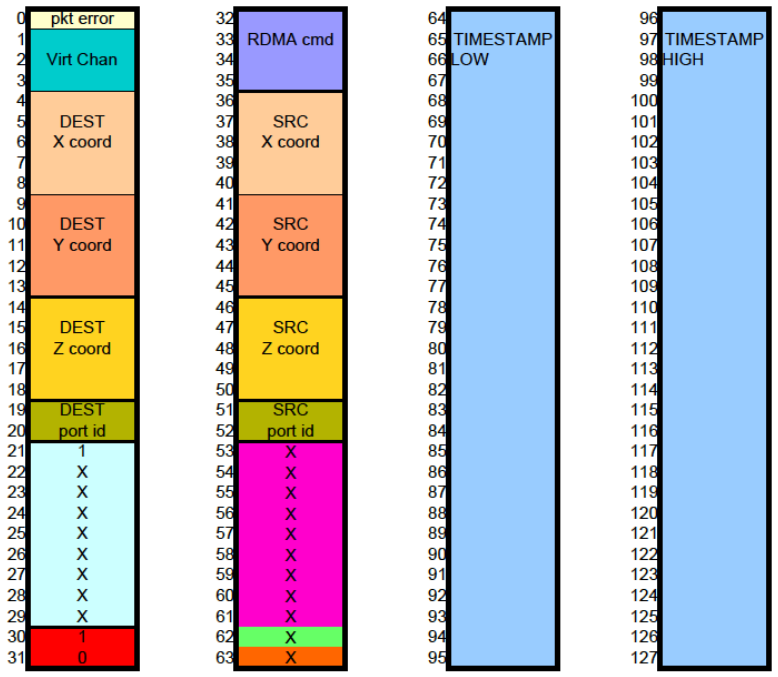

Figure 3.4: Header of a DQN-Routing acknowledge traffic packet in the APEnet+ interconnect.

2. Availability of internal, or tightly coupled, memory to store the table of routed packets waiting for an acknowledge, containing the information needed to calculate the reward (timestamp and distance of destination node), and the port from which the packet was received in order to route the acknowledge packet along the reverse path. In our simulator, for a 4×4 torus configuration and with one Agent serving four Routers, we set the maximum number of active episodes for the Agent to 100 without experiencing overruns during simulations. This parameter correspond also to the maximum allowed number of routed packets waiting for an acknowledge collectively for the four Routers. So, a preliminary estimate for the size of the table is in the order of some tens of entries for this topology, with a corresponding needed memory size in the order of few kilobytes. For a given bandwidth offered to the network, the number of pending entries is proportional to the torus diameter (see formula

2.1).

3. A low latency interface to communicate with the Agent. For example using the AMBA/AXI standard for On-Chip communication[75], also available for FPGA-based designs like APEnet and ExaNet, a communication latency in the order of few clock cycles is achievable.

4. Support for a distributed time synchronization infrastructure with an accuracy in the order of hundreds nanoseconds. We assume a 64-bit format for the timestamp.

Agent Architecture As we will see in Chapter 4, we adopted the *Rainbow* algorithm [39] for the Agent implementation in the simulation framework we used as experimental setup. A very raw estimate of memory and floating point performance needed to implement the Agent with the configuration used in this work (see appendix A.1) can be directly inferred by measuring the memory and processor occupancy for the GPU processes in the workstation we used to execute our simulations, neglecting the contribution of the CPU part. Each Agent utilizes 343 Mbytes of memory, and roughly the 0.5% of the processing resources, that considering the GPU model (NVIDIA GeForce RTX 2080 Ti) translates to $\sim$ 70 gigaFLOPS in single precision floating point arithmetic and $\sim$ 2 gigaFLOPS in double precision floating point arithmetic. These quantities are compatible also with an embedded processing device. Nevertheless a couple of consideration must be made. First, as we will see in the next chapter, we used the *Rainbow* Agent implementation provided by the RLlib library of the Ray framework [76] without making any modification on it. This choice was guided by the decision to give priority to the assessment of the potential of the DQN-Routing algorithm. Nevertheless the *Rainbow* Agent was designed, as the original DQN, to interact with the environment provided by the Atari 2600 benchmark. This environment requires the Agent to manage an higher dimensionality observation space, a $[210, 160, 3]$ floating point array corresponding to 210×160 RGB images, with respect to the one offered by the Router to the Agent in DQN-Routing, that is a unidimensional floating point array of 13 elements. This hints for a possible simplification of the Agent for our purposes, allowing to make an implementation that would require less memory and processing resources. Second, in this work we did not address the problem of designing or selecting a processing architecture optimized for low-latency inference in order to provide quickly a routing decision proposal, that represents a fundamental requirement for the Agent implementation.

Algorithm 2 : DQN-Routing for 2-D Tori (regular packets)

```
1: Global: Size of 2-D Torus ($Xmax$, $Ymax$)
2: Input: Packet Unique Id ($Puid$), Packet diverted to deterministic escape path flag ($Pdor$), Packet payload size ($Psize$), Receive Port for packet ($Rport$), Coordinates of current node ($Xcurr$, $Ycurr$) and destination node ($Xdest$, $Ydest$)
3: Output: Selected output $Channel$
4: Procedure:
5: $Timestamp$ := get_time();
6: $Xoffset$ := $Xdest$ - $Xcurr$;
7: $Yoffset$ := $Ydest$ - $Ycurr$;
8: if $Xoffset$ = 0 and $Yoffset$ = 0 then
9:     $ACKpacket$ := prepare_ack_packet($Puid$, $Timestamp$, $Xcurr$, $Ycurr$, $Rport$);
10:    inject_packet($ACKpacket$);
11:    $Channel$ := $Internal$;
12: end if
13: if $Pdor$ = TRUE then
14:    add_entry($Puid$, 0, 0, “”, $Rport$);
15:    $Channel$ := deterministic_order_routing($Xcurr$, $Ycurr$, $Xdest$, $Ydest$);
16: end if
17: $Distance$ := MIN(abs($Xdest$ - $Xcurr$), $Xmax$ - abs($Xdest$ - $Xcurr$)) + MIN(abs($Ydest$ - $Ycurr$), $Ymax$ - abs($Ydest$ - $Ycurr$));
18: $Eid$ := agent_start_episode();
19: if $Puid$ = “” then
20:    set_packet_uid($Eid$);
21:    $Puid$ := $Eid$
22: end if
23: add_entry($Puid$, $Timestamp$, $Distance$, $Eid$, $Rport$);
24: $Observation$ := prepare_observation($Xcurr$, $Ycurr$, $Xdest$, $YDest$, $Psize$, $LocOccDqnChX-$, $LocOccDqnChX+$, $LocOccDqnChY-$, $LocOccDqnChY+$, $RemOccDqnChX-$, $RemOccDqnChX+$, $RemOccDqnChY-$, $RemOccDqnChY+$);
25: $CandidateChannel$ := agent_get_action($Eid$, $Observation$);
26: if is_ch_available($Psize$, $CandidateChannel$) = FALSE then
27:    set_packet_dor(TRUE);
28:    $Channel$ := deterministic_order_routing($Xcurr$, $Ycurr$, $Xdest$, $Ydest$);
29: end if
30: $Channel$ := $CandidateChannel$;
```

Algorithm 3 : DQN-Routing for 2-D Tori (acknowledge packets)

```
1: Global: Size of 2-D Torus ($Xmax$, $Ymax$)
2: Input: Packet Unique Id ($Puid$), Packet Timestamp ($Ptimestamp$), Coordinates of
   current node ($Xcurr$, $Ycurr$) and destination node ($Xdest$, $Ydest$)
3: Output: Selected output $Channel$
4: Procedure:
5: $Xoffset := Xdest - Xcurr$;
6: $Yoffset := Ydest - Ycurr$;
7: if $Xoffset = 0$ and $Yoffset = 0$ then
8:     $Entry$ := find_entry($Puid$);
9:     if $Entry$ != $null$ then
10:        if $Entry.Eid$ != “” then ▷ calculate and log reward to the agent for this episode
11:            $Reward := Entry.Distance/(Ptimestamp - Entry.Timestamp)$;
12:            agent_log_reward($Entry.Eid$, $Reward$);
13:            agent_end_episode($Entry.Eid$);
14:        end if                          ▷ packet was escaped to a DOR path, no reward
15:        if $Entry.Rport = Internal$ then
16:            $Channel := Flush$;                    ▷ ack packet reached source node
17:        else if $Entry.Rport$ = X+ then
18:            $Xdest := (Xdest + 1) \mod Xmax$;
19:        else if $Entry.Rport$ = X- then
20:            $Xdest := (Xdest - 1) \mod Xmax$;
21:        else if $Entry.Rport$ = Y+ then
22:            $Ydest := (Ydest + 1) \mod Ymax$;
23:        else if $Entry.Rport$ = Y- then
24:            $Ydest := (Ydest - 1) \mod Ymax$;
25:        end if
26:        delete_entry($Entry$);
27:        patch_packet_destination($Xdest$, $Ydest$);
28:        $Channel$ := deterministic_order_routing($Xcurr$, $Ycurr$, $Xdest$, $Ydest$);
29:    end if
30: end if
31: $Channel := Flush$;                           ▷ flush mis-routed ack packet
```

4 Experimental Setup and Measurements

In this chapter we describe the simulation framework that we have integrated to be used as the experimental setup to test the DQN-Routing algorithm and to measure its performance.

For this purposes, both synthetic and application-generated traffic patterns were used. In particular, besides Uniform Random Traffic, we used network communication traces collected during the execution of the DPSNN spiking neural network simulator [13] on a parallel cluster.

Collected results are presented and discussed, comparing them with those obtained with deterministic dimension-order routing 2.7.1 and fully adaptive star-channel 2.7.2 algorithms.

4.1 The Experimental Setup

Our experimental setup can be represented as a multi-agent reinforcement learning problem, where the environment is provided by a simulator of the network. When we started this work, there were no available simulation frameworks suitable for our purposes. So we developed an experimental setup integrating the OMNeT++ [10] network simulation framework with the Ray [76] distributed execution framework that includes the reinforcement learning RLlib library [5]. In our setup, the OMNeT++ network simulator constitutes the *external* environment for the reinforcement learning agents

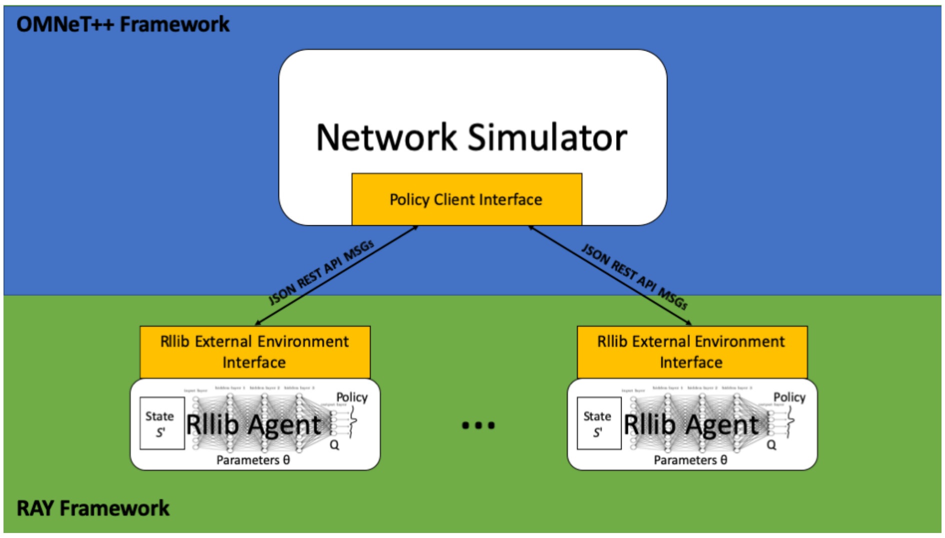

Figure 4.1: The integrated simulation framework used as experimental setup in this work.

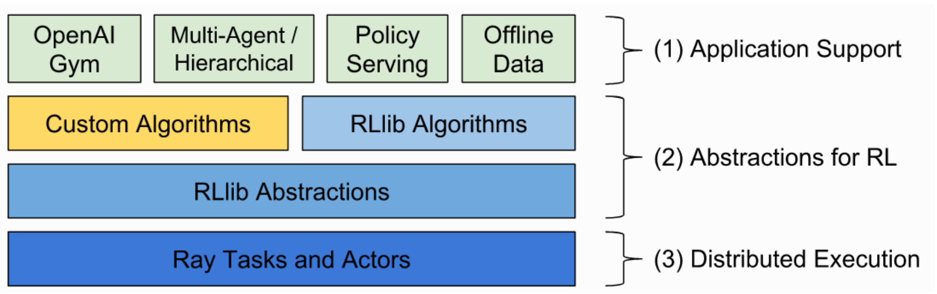

Figure 4.2: A stacked modular view of the RLlib library. Original image from [5]

implemented in RLlib and executed on the Ray framework. The network simulator invokes the services of the RLlib agents, that are executed in a distributed fashion in the Ray framework, thanks to REST messages using the JSON data-interchange format, as shown in Figure 4.1.

Recently ns3-gym [77], a framework similar to the one described here but based on the ns-3 [78] simulator, has been released. We plan to release our framework open-source as well.

4.1.1 The Reinforcement Learning Distributed Execution Framework

RLlib [5] is an open-source library for reinforcement learning that offers both high scalability and a unified API for a variety of applications, as shown in the top level of Figure 4.2. It natively supports TensorFlow, TensorFlow Eager, and PyTorch, but most of its internals are framework agnostic.

RLlib is designed to scale to large clusters, but it is also capable of applying optimizations such as vectorization for single-core efficiency. This allows for effective use of multiple agents also on a single workstation.

Policies are a key concept in RLlib. They are Python classes that define how an agent acts in an environment: agents query the policy to determine their actions. Policy classes encapsulate the core numerical components of RL algorithms. This typically includes the policy model that determines actions to take, a trajectory postprocessor for experiences, and a loss function to improve the policy given postprocessed experiences.

In RLlib all data interchange happens in the form of sample batches. Sample batches encode one or more fragments of a trajectory. Typically, collects batches of size *sample_batch_size*, and concatenates one or more of these batches into a batch of size *train_batch_size* that is the input to the Stochastic Gradient Descent algorithm.

RLlib provides ways to customize almost all aspects of training, including the environment, neural network model, action distribution, and policy definitions, as shown in Figure 4.3. By the way a number of reinforcement learning algorithms are provided, including *Rainbow* that is the one that we adopted in this work, that integrates several extensions to the DQN, as described in the work by M. Hessel et al. [39]. In its default configuration, the parameters of the neural network that approximate the action values are optimized by using Adam [79], a method for efficient stochastic optimization requiring only first-order gradients and with little memory requirement. For the complete configuration of the agent that has been used for this work please refer to Appendix A.1.

Using the TensorBoard tool it is possible to constantly monitor the Agents training process during the, possibly very long, simulations and have an im-

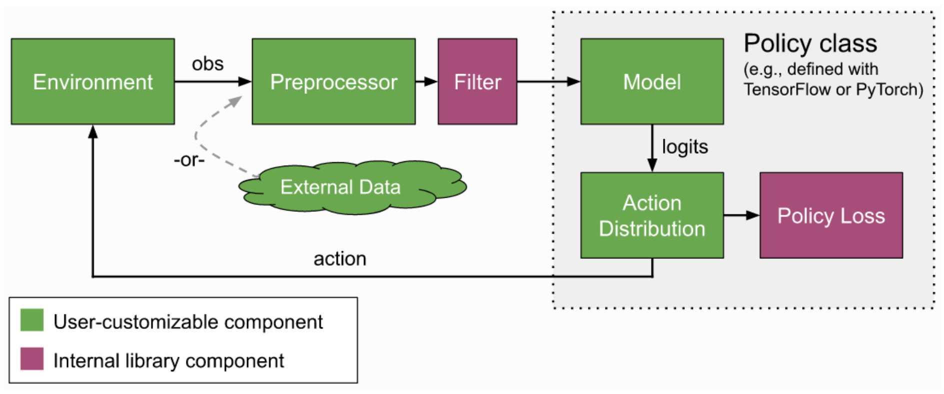

Figure 4.3: RLlib components, their relations and customizability. Original image from [5]

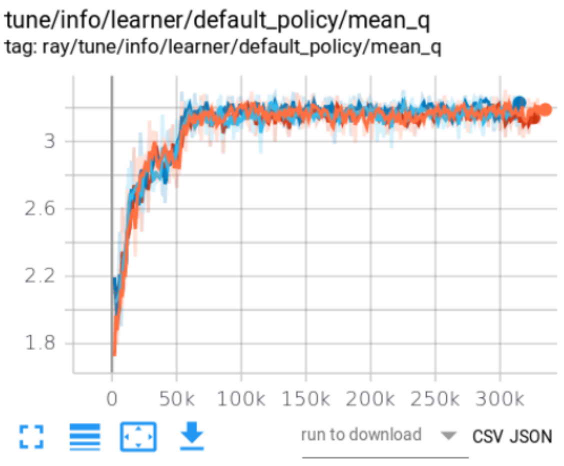

Figure 4.4: Monitoring the training of 4 Agents during the simulation (mean Q value)

mediate feedback. An example of monitoring the 4 agents training during the simulation of 1.6 s of DPSNN traffic with an average neuron firing rate of 21 Hz (see section 4.3.3) is shown in Figure 4.4 and Figure 4.5.

4.1.2 The Network Simulation Framework

OMNeT++ [10] is a C++-based discrete event simulation library developed for modelling communication networks, multiprocessors and distributed systems. Thanks to its extensible and modular design it can be easily used to implement any discrete event simulation. OMNeT++ adopts a hierarchical structure, with modules connected through channels. There are two kind of modules: *simple* modules and *compound* modules. Simple modules are the

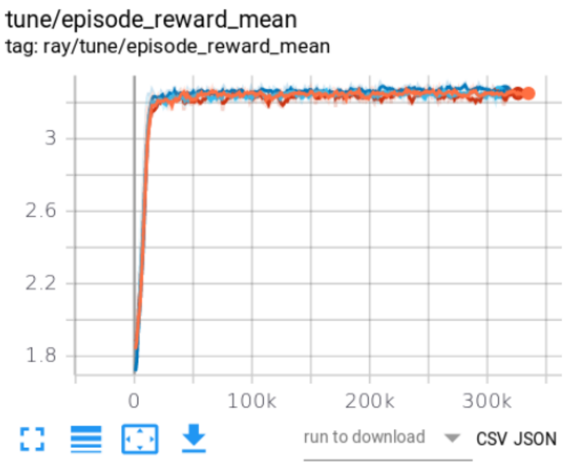

Figure 4.5: Monitoring the training of 4 Agents during the simulation (mean reward)

active components of the simulation, they are implemented in C++ as an object oriented specialization of base class modules. Compound modules are passive containers for any number of simple and compound modules. The connection between modules is defined using the NED (Network DEscription) language. The OMNeT++ framework also provides: support to parallel distributed simulations using the MPI library, a GUI for developing the code and for running and debugging the simulations, tools for data analysis and statistics collection, and support for a plethora of network protocols.

We selected this network simulation framework between several other suitable ones, like ns-3 [78] or INSEE [80] to cite a few, mainly because of the availability of the VCT simulation library for OMNeT++ [81], and the model of the APEnet+ network interconnect based on this library. In particular, the implementation of the new DQN-Routing router component was quite straightforward with the library, using the C++ inheritance mechanism.

4.1.3 Interfacing the Frameworks

The interface between the network simulator and the reinforcement learning frameworks has been implemented leveraging the RLlib support for *External Environments*. In many situations, it does not make sense for an environment to be "stepped" by RLlib, as it is the case interfacing with OpenAI Gym agents [82]. For example, if a policy is to be used in a web serving system, then it is more natural for an agent to query a service that serves policy decisions, and for that service to learn from experience over time. This case also naturally arises with external simulators that run independently outside the control of RLlib, but may still want to leverage RLlib for training. RLlib provides the *ExternalEnv* class for this purpose. Unlike other environments, *ExternalEnv* has its own thread of control. At any point, agents on that thread can query the current policy for decisions. This can be done for multiple concurrent episodes as well. We used *ExternalEnv* to implement a REST policy server that learns over time using the RLlib implementation of the *Rainbow* algorithm.

On the OMNeT++ side, we developed the *PolicyClient* C++ class that uses the JSON data-exchange format to implement a REST API with the following *primitives*:

1. request the start of an episode;
2. provide an observation and get the on-policy action;
3. provide a reward;
4. request the end of an episode.

4.1.4 The Simulated Network Architecture

The reference network architecture for our investigation has been the ExaNet [12] multi-tier hybrid network dedicated to HPC. In this context, we focused on the configuration characterized by a number of nodes in the sub-torus tiers equal to sixteen in a 4×4 bi-dimensional torus, which allowed to effectively simulate the network by means of a single, although powerful, GPU-accelerated workstation. The ExaNet design is based on the APEnet+

interconnect router (APErouter) for its network layer and on the APEnet+ logic channel (APElink) for its data-link layer. In this work we used the original OMNeT++ APEnet+ interconnect model to perform the simulations, and despite some small differences between the ExaNet and APEnet+ versions of these components, we believe that the results obtained can be considered interesting for both the network interconnects. The simulated network mimics four ExaNest Mezzanine boards, each one hosting four QDFB nodes, where the nodes belonging to the same mezzanine are interconnected on-board with a bidirectional ring on the X dimension, and the nodes belonging to different mezzanine boards are connected along the Y dimension. The resulting topology is a 4×4 torus. We set to four the number of DQN-Routing Agents for this network, hence an Agent offers its services to the four nodes of the same mezzanine board. This is a slight deviation on what we have described since now with – one Agent serving one Router – but there is no conceptual difference on the envisaged architecture for DQN-Routing. The reason behind this choice was to enrich the experience of the Agent, providing it a wider view on the network state, with the idea that this would help the learning process; at the same time we wanted to investigate the opportunity of a less demanding scaling of the additional resources requested to actually implement DQN-Routing.

4.2 Traffic Models

Both synthetic and application-generated traffic patterns were adopted in the simulation to assess the performance of the DQN-Routing algorithm. In particular, besides Uniform Random Traffic, we used network communication traces collected during the execution of the DPSNN spiking neural network simulator [13] on a parallel cluster.

4.2.1 Uniform Random Traffic

This traffic pattern can be described as follows: i) all nodes inject packets in the network at a given bandwidth; ii) all the packets have a random payload size – up to 256 128-bit flits – and they are sent to randomly selected nodes at the frequency needed to maintain the chosen bandwidth. The pseudo random number generator uses the Mersenne Twister algorithm implemented in the OMNeT++ framework; this generator has a period of $2^{19937} - 1$ and 623-dimension equidistribution property [83]. The packets are sent using the software queue so, if network congestion occurs, the effective injected bandwidth may be lower that the selected one. This traffic pattern is very useful to characterize networks against a generic application.

4.2.2 DPSNN Simulator

In this work, we used DPSNN as a reference application simulating networks of point-like spiking neurons (Leaky Integrate-and-Fire neurons with Spike Frequency Adaptation, 80% excitatory, 20% inhibitory) [13]. Synapses inject instantaneous post-synaptic currents while synaptic plasticity is disabled. The simulator implements a mixed event-driven (synaptic and neural dynamics) and time-driven (exchange of spiking messages) integration scheme.

In the simulations used herein, the neural network is organized as bidimensional grids of modules (mimicking cortical columns). Each module is composed of 2500 point-like neurons, further organized into subpopulations. The number of synapses projected by each neuron is kept constant with an average value of 2250 synapses per neuron and neurons are evenly distributed among processes. As a result, larger network configurations result in more sparse synaptic connection matrices. Each neuron receives also the stimulus of 800 “external” synapses, each one delivering a Poissonian spike train at a rate of about 3 Hz. After an initial transient, the neural network enters an asynchronous irregular firing regime. Inter-process communication is necessary to deliver spikes to target neurons residing on a process different from the one hosting the source neuron. Spikes are delivered using the AER representation (spiking neuron ID, emission time) [84]; in our case 12 byte per spike are required. The exchange of spikes is currently implemented by means of synchronous MPI collective operations (mpi_all_to_all_v). In each process, all spikes produced by neurons and targeted to neurons belonging to a different process are packed into a single message and delivered. The total number of messages required by the neural network increases with the square of the number of processes on which the simulation is run. Here is a rundown of the application tasks that the DPSNN simulator performs:

- **Computation**: event-driven integration of all neural dynamics and synaptic current injection events, occurring in a single network synchronization time step (set to 1 ms). This includes a component dominated by memory access to: 1- time delay queues of axonal spikes, 2- lists of neuro-synaptic connections, 3- lists of synapses.
- **Communication**: transmission along the interconnect system of the axonal spikes to the subset of processes where target neurons exist.
- **Synchronization**: synchronization barrier inserted to simplify the weighting of computation and communication components.

Fluctuations in computation load or communication congestion cause idling cores and hindered parallelization. The relative weight of the compute increases with the number of neurons per process. On the other side, a higher number of processes results in higher relative communication costs. Because of the regularity of the application, it has been possible to mimic its execution pattern in our simulator using a finite state machine and network traces collected during its execution on a real computing platform [81]. The traffic pattern is characterized by long periods of network inactivity, followed by bursts of packets when the communication of the axonal spikes occurs. The time spent by the application to calculate the next state of the neuron can be considered proportional to the number of spikes received. The average processing time per single spike can be estimated by profiling the execution of the application on a real machine. Although the processing time can change drastically by changing the underling computing devices, we can still compare different networking solutions without necessarily re-calibrate the simulation of the processing part.

4.3 Simulation Results

4.3.1 Metrics and Methods

The most commonly used metrics to measure the performance for an interconnection network are *latency* and *accepted traffic*.

Latency is defined as the time elapsed from the beginning of the packet transmission and the receive of the message at the destination node. This definition can be interpreted in different ways: considering a complete system that includes it the software stack, latency will include the time spent into the different software layers; on the other hand, if we want to characterize the net performance of the underlying interconnection network, only the time spent by the messages travelling in the network should be accounted. In our measurements we adhered to the latter approach, we did not consider the time spent by messages in the *soft queue*.

Once we fix a network architecture, latency varies from packet to packet since its value is affected by: the distance between source and destination nodes, the size of the packet and the congestion of the network. The value for a single packet is not meaningful, especially if we consider a synthetic traffic pattern; so we used the average value to give a global picture of the network behaviour. The standard deviation of the latency is also important to get an indication of network congestion.

In the case that the simulator setup is able to perform a full simulation of the application on the system, including that of the processing part, the simulated time-to-solution is a clear performance metric.

Accepted traffic or throughput is defined as the amount of information delivered by the network per time unit. Because the amount of information delivered depends on the number of nodes, this value must be normalized in order to compare different network configurations, dividing it by the number of nodes in the network.

It is important to note the difference between applied load and accepted traffic: the first is an input parameter of the simulation and determines the amount of data injected into the network; the latter is an output of the simulation and determines the amount of data delivered by the network.

It is worth noticing that in the simulation we performed, the processing time needed for the routing is not taken into account. While this time is reduced to few clock cyles for the simple dimension-order arithmetic routing, this may not be the case for DQN-Routing. Nevertheless in this work we are interested in the assessment of the potential impact of this new algorithm. If it turns out to outperform other known algorithms, a further investigation on the efficient implementation of the required processing will be inserted in our research agenda.

In DQN-Routing, livelock avoidance has been implemented checking the number of *hops* of the packet being routed: if it is above a configurable threshold the packet will be escaped to deterministic DOR routing. For the measurements presented here, we set this threshold to 8, that corresponds to twice the network diameter.

Our measurements for DQN-Routing were performed restarting the agents at the beginning of the simulation and initializing with uniform random weights the Deep-Q Network. The Agent will then start an exploration phase that in

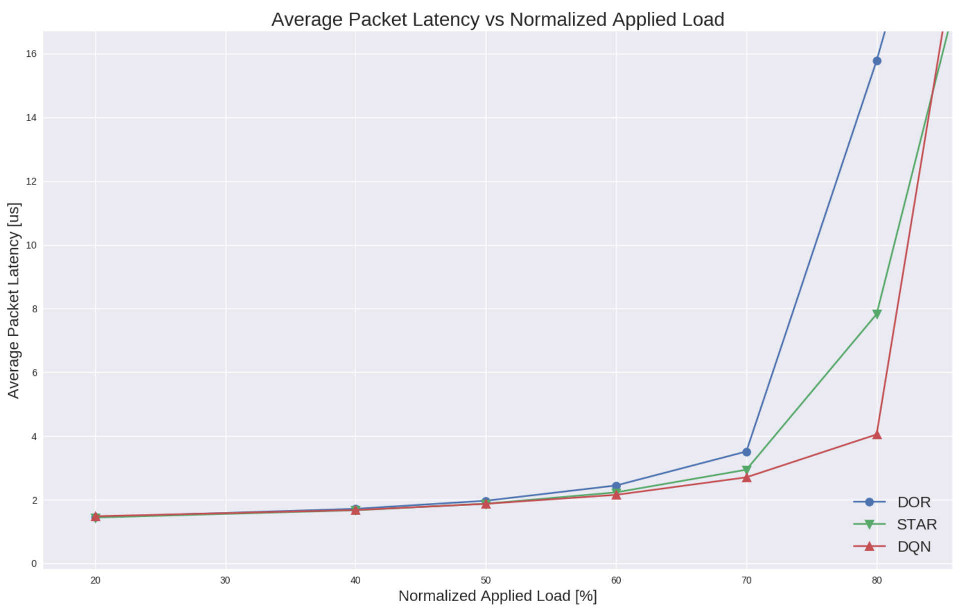

Figure 4.6: Average packet latency vs normalized applied load under uniform random traffic.

our configuration will be performed for the first 100,000 timesteps, i.e. routing action proposals, annealing ϵ (the probability of taking a non-optimal action to explore the environment) from 1.0 to 0.1 over 10,000 time steps. ϵ is then kept constant at 0.1 until 100,000 time steps are reached; after, it will be kept constant at 0.02.

4.3.2 Uniform Random Traffic

When stimulating the network with Uniform Random Traffic at different values of applied load, with the nodes producing traffic using a Bernoulli process and with a uniformly random destination, we obtained the results shown in Figure 4.6.

Besides the fully adaptive, non-minimal path DQN-Routing (DQN, see section 3.2), we considered for a comparison the deterministic dimension-order routing (DOR, see section 2.7.1) and the fully adaptive, minimal path star-channel routing (STAR, see section 2.7.2). The comparison with the

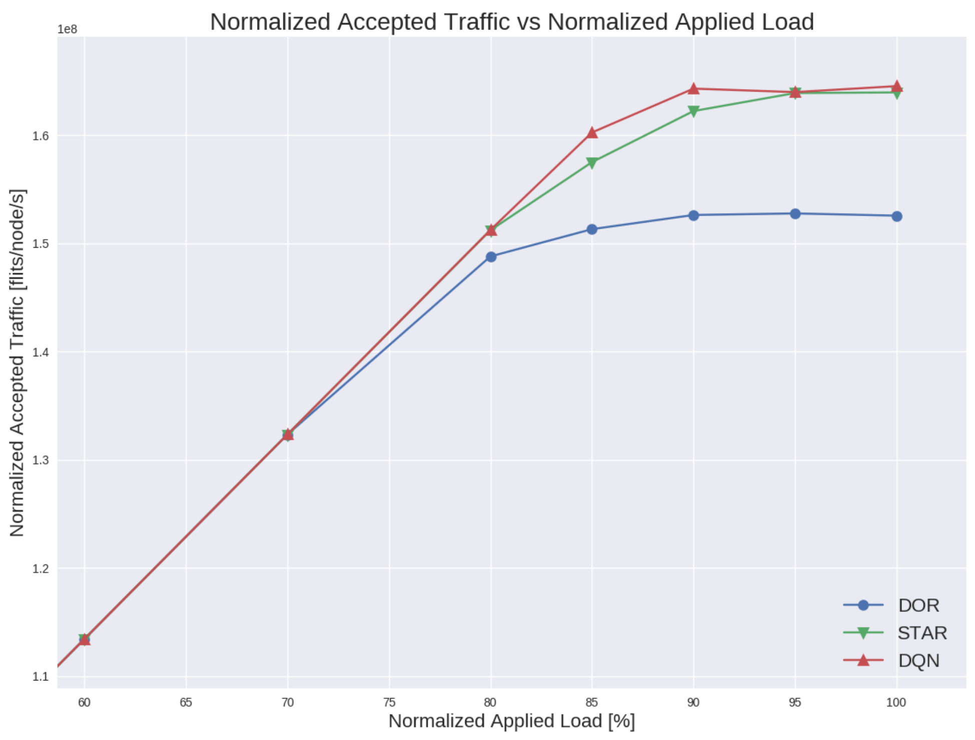

Figure 4.7: Normalized accepted traffic vs normalized applied load under uniform random traffic.

star-channel routing is interesting because it poses exactly the same requirement of DQN-Routing in terms of the number of additional virtual channels requested for every physical channel; this number is equal to one.

As expected, both the fully adaptive routing algorithms provide a better use of the available network resources, resulting in significantly lower average latency than that achieved using the deterministic DOR routing, starting from the 60% of the normalized applied load.

In absolute, the best performance in terms of average latency are obtained by the DQN-Routing. At a 80% of the normalized applied load the network is still below its critical congestion threshold and the average latency is roughly the half (52%) of that obtained with the star-channel routing, while it is roughly one quarter (26%) of that measured using the dimension-order routing.

The same conclusions can be drawn examining the plot in Figure 4.7, that shows the normalized accepted traffic versus the normalized applied load under uniform random traffic. The plot shows a linear region shared the three different routing algorithms. When the network is in the linear region, it is below its critical congestion threshold and can still properly dispatch the injected traffic messages. When the applied load reaches the saturation point, the accepted traffic starts to exit from the linear region of the plot and reaches a plateau. Please note that the plateau does not correspond to the maximum value of the theoretical normalized accepted traffic of the network, that corresponds to $1.92e8$ flits/node/s, because of congestion effects.

Figure 4.7 shows that the critical congestion threshold is between 70$\%$ and 80$\%$ of the normalized applied load for the DOR case, while it is between 80$\%$ and 85$\%$ for star-channel routing. The highest value for the critical congestion threshold is obtained with DQN-Routing, in the range between 85$\%$ and 90$\%$ of the normalized applied load.

4.3.3 DPSNN Traffic

We performed simulations using network traces collected by the execution of 400 ms of the DPSNN application configured to run with 16 processes on a 4×4 grid for different values of the mean neuron firing rates. Increasing the mean firing rate leads to a step up in the applied load. With this distribution of processes – one process mapped on one network node – the communication pattern comes to closely resemble an all-to-all one occurring in bursts separated by very low network traffic.

As described in section 4.3.1, our standard procedure for taking measurements involves restarting the Agents at the beginning of each simulation. In order to investigate the capability of the DQN-Routing algorithm to capture the dynamic of the network traffic pattern generated by the DPSNN applica-

Firing Rate (Hz)	**Mean Latency (μs)**		**STDEV (μs)**		**MAX (μs)**	
	DOR	**DQN**	**DOR**	**DQN**	**DOR**	**DQN**
21	1.010	1.006	0.508	0.491	3.801	4.729
47	1.900	1.861	1.487	1.421	7.369	8.101
71	2.682	2.618	2.429	2.333	10.411	11.270
89	2.883	2.832	2.678	2.597	11.791	12.270
268	9.755	9.672	10.511	10.368	45.799	41.810
556	33.667	31.290	25.985	23.670	110.301	124.154

Table 4.1: Measurements obtained simulating the execution of 400 ms of the DPSNN application distributed on 4x4 processes at different mean neuron firing rates.

tion, we took two sets of measurements: one using the standard procedure and another performing a pre-training of the Agents by running in sequence two identical simulations without restarting the Agents in between them, then collecting results on the second one. With pre-training, we measured a slight improvement in performance (about 4%), i.e. a decrease in the mean and maximum values and in the standard deviation of the communication latency compared with those measured when using the standard procedure.

Results obtained with simulations using the dimension-order routing (DOR) and DQN-Routing with pre-training (DQN) are reported in Table 4.1.

The two algorithms perform in a substantially comparable way; DQN-Routing has slightly higher values of maximum latency due to its non-minimal nature and to the residual exploration performed by the Agents, which amounts to sporadic cases and does not affect performance. On the other way, DQN-Routing has a small but measurable edge in its favour all-round the firing rates as regards the mean latency and, perhaps more interestingly, the standard deviation.

This is probably due to the broader usage of network resources that DQN-Routing is free to employ when congestion is present which would be otherwise left unused with the less flexible dimension-order routing.

5 Conclusions and Future Work

We designed DQN-Routing, a distributed, unicast, fully adaptive non-minimal routing algorithm for torus networks based on Deep Reinforcement Learning, and the general architecture of the router implementing it.

In order to test it and measure its performances, we developed an experimental setup integrating the OMNeT++ [10] network simulation framework with the Ray [76] distributed execution framework including the RLlib [5] reinforcement learning library, as when we started this work there were no suitable frameworks available for our purposes.

The algorithm has been tested and its performance has been measured using both synthetic and application-generated network traffic.

DQN-Routing has proved to outperform both the deterministic dimension-order routing (DOR) [4] algorithm and the fully adaptive, minimal-path star-channel [1] algorithm when using uniform random synthetic traffic, that it is typically used to represent to a generic application.

On the other hand, when tested against the bursty all-to-all traffic pattern generated by the DPSNN distributed spiking neural network simulator [13], the two algorithms performed in a substantially comparable way; while DQN-Routing showed slightly higher values of maximum latency due to its non-minimal nature and to the exploration performed by the Agents, it featured a small but measurable edge in its favour as regards the mean value and, perhaps more interestingly, the standard deviation of the communication latency.

To improve performance in a continuous operation scenario, with changing network traffic patterns, we foresee a mechanism for the automatic restart of the learning phase for the Agent, either periodic or triggered by a decrease of the mean Q-value below a configurable threshold. Besides this automatic mechanism, the restart of a learning phase in the Agent could also be initiated via a *cross-layer* information propagation from the application to the network level, implementing a dedicated API. For example, in the DPSNN reference application the average spiking rate of the simulated neural network is constantly monitored, and the application could communicate to the network level when there is a significant change in this quantity, as this will affect the network status.

To characterize the behaviour of the algorithm in presence of unbalanced traffic sources, we plan to extend this study with hot region, hot spot and permutation-based traffic, to verify whether DQN-Routing is be able to learn where the congestion points are and how to circumvent them.

There are several hints showing that the DQN-Routing algorithm could be effectively adopted for larger and possibly highest dimensional torus networks and even for different network topologies (e. g. Dragonfly): the assessment of this statement is planned as future work. The simulation framework that we have integrated is ready for this task as it was designed with scalability in mind: OMNeT++ supports MPI for parallel execution and RLlib library, that we used to simulate the Agent, is part of the Ray distributed execution framework.

Finally, it has not escaped our notice that DQN-Routing could be useful in other domains that make use of torus/mesh network interconnects, like the Network-On-Chip one. Furthermore in this scenario the implementation of the Agent would be facilitated by the tight integration between the router and processing resources.

Appendices

A

A.1 Configuration of the Agent

```
DEFAULT_CONFIG = with_common_config({
    # === Model ===
    # Number of atoms for representing the distribution of
    #return.
    #When this is greater than 1, distributional
    #Q-learning is used.
    #the discrete supports are bounded by v_min and v_max
    "num_atoms": 1,
    "v_min": -10.0,
    "v_max": 10.0,
    # Whether to use noisy network
    "noisy": False,
    # control the initial value of noisy nets
    "sigma0": 0.5,
    # Whether to use dueling dqn
    "dueling": True,
    # Whether to use double dqn
    "double_q": True,
    # Postprocess model outputs with these hidden layers
    #to compute the state and action values. See also
    #the model config in catalog.py.
```

```
"hiddens": [256],
# N-step Q learning
"n_step": 1,

# === Exploration ===
# Max num timesteps for annealing schedules.
#Exploration is annealed from 1.0 to exploration_fraction
#over this number of timesteps scaled by
#exploration_fraction
"schedule_max_timesteps": 100000,
# Minimum env steps to optimize for per train call.
#This value does not affect learning, only the
#length of iterations.
"timesteps_per_iteration": 1000,
# Fraction of entire training period over which the
#exploration rate is annealed
"exploration_fraction": 0.1,
# Final value of random action probability
"exploration_final_eps": 0.02,
# Update the target network every
#‘target_network_update_freq‘ steps.
"target_network_update_freq": 500,
# Use softmax for sampling actions. Required for
#off policy estimation.
"soft_q": False,
# Softmax temperature. Q values are divided by this
#value prior to softmax.
# Softmax approaches argmax as the temperature
#drops to zero.
```

```
    "softmax_temp": 1.0,
    # If True parameter space noise will be used for
    #exploration
    # See https://blog.openai.com/better-exploration-with
    #-parameter-noise/
    "parameter_noise": False,
    # Extra configuration that disables exploration.
    "evaluation_config": {
        "exploration_fraction": 0,
        "exploration_final_eps": 0,
    },

    # === Replay buffer ===
    # Size of the replay buffer. Note that
    #if async_updates is set, then each worker will have
    #a replay buffer of this size.
    "buffer_size": 50000,
    # If True prioritized replay buffer will be used.
    "prioritized_replay": False,
    # Alpha parameter for prioritized replay buffer.
    "prioritized_replay_alpha": 0.6,
    # Beta parameter for sampling from prioritized
    #replay buffer.
    "prioritized_replay_beta": 0.4,
    # Fraction of entire training period over which
    #the beta parameter is annealed
    "beta_annealing_fraction": 0.2,
    # Final value of beta
    "final_prioritized_replay_beta": 0.4,
# Epsilon to add to the TD errors when updating
#priorities.
"prioritized_replay_eps": 1e-6,
# Whether to LZ4 compress observations
"compress_observations": True,

# === Optimization ===
# Learning rate for adam optimizer
"lr": 5e-4,
# Learning rate schedule
"lr_schedule": None,
# Adam epsilon hyper parameter
"adam_epsilon": 1e-8,
# If not None, clip gradients during optimization
#at this value
"grad_norm_clipping": 40,
# How many steps of the model to sample before
#learning starts.
"learning_starts": 1000,
# Update the replay buffer with this many samples at once.
#Note that this setting applies per-worker
#if num_workers > 1.
"sample_batch_size": 4,
# Size of a batched sampled from replay buffer for
#training. Note that if async_updates is set, then each
#worker returns gradients for a batch of this size.
"train_batch_size": 32,

# === Parallelism ===
    # Number of workers for collecting samples with.
    #This only makes sense to increase if your environment
    #is particularly slow to sample, or if you"re using
    #the Async or Ape-X optimizers.
    "num_workers": 0,
    # Whether to use a distribution of epsilons across
    #workers for exploration.
    "per_worker_exploration": False,
    # Whether to compute priorities on workers.
    "worker_side_prioritization": False,
    # Prevent iterations from going lower than this
    #time span
    "min_iter_time_s": 1,
    #Number of GPUs available in the system
    "num_gpus": 1
})
```

Bibliography

[1] L. Gravano, G. D. Pifarre, P. E. Berman, and J. L. C. Sanz, "Adaptive deadlock- and livelock-free routing with all minimal paths in torus networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 12, pp. 1233–1251, Dec 1994.

[2] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, highly-scalable dragonfly topology," in *2008 International Symposium on Computer Architecture*, June 2008, pp. 77–88.

[3] J. Duato, S. Yalamanchili, and N. Lionel, *Interconnection Networks: An Engineering Approach.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[4] W. J. Dally and C. L. Seitz, "Deadlock-free message routing in multiprocessor interconnection networks," *Computers, IEEE Transactions on*, vol. C-36, no. 5, pp. 547–553, 1987.

[5] "Rllib: Scalable reinforcement learning," accessed: 2019-10-17. [Online]. Available: https://ray.readthedocs.io/en/latest/rllib.html

[6] J. Navaridas, J. Lant, J. A. Pascual, M. Luján, and J. Goodacre, "Design exploration of multi-tier interconnection networks for exascale systems," in *Proceedings of the 48th International Conference on Parallel Processing*, ser. ICPP 2019. New York, NY, USA: ACM, 2019, pp. 49:1–49:10. [Online]. Available: http://doi.acm.org/10.1145/3337821.3337903

[7] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. The MIT Press, 2018. [Online]. Available: http://incompleteideas.net/book/the-book-2nd.html

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement

learning," *arXiv preprint arXiv:1312.5602*, 2013. [Online]. Available: https://arxiv.org/pdf/1312.5602.pdf

[9] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015. [Online]. Available: http://dx.doi.org/10.1038/nature14236

[10] "Omnet++ project," accessed: 2019-10-15. [Online]. Available: https://omnetpp.org/

[11] R. Ammendola, A. Biagioni, O. Frezza, F. Lo Cicero, A. Lonardo, P. S. Paolucci, D. Rossetti, F. Simula, L. Tosoratto, and P. Vicini, "APEnet+: a 3D Torus network optimized for GPU-based HPC systems," *Journal of Physics: Conference Series*, vol. 396, no. 4, p. 042059, 2012. [Online]. Available: http://stacks.iop.org/1742-6596/396/i=4/a=042059

[12] M. Katevenis *et al.*, "Next generation of exascale-class systems: Exanest project and the status of its interconnect and storage development," *Microprocessors and Microsystems*, vol. 61, pp. 58 – 71, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0141933118300188

[13] E. Pastorelli, C. Capone, F. Simula, M. V. Sanchez-Vives, P. Del Giudice, M. Mattia, and P. S. Paolucci, "Scaling of a large-scale simulation of synchronous slow-wave and asynchronous awake-like activity of a cortical model with long-range interconnections," *Frontiers in Systems Neuroscience*, vol. 13, p. 33, 2019. [Online]. Available: https://www.frontiersin.org/article/10.3389/fnsys.2019.00033

[14] R. Ananthanarayanan, S. K. Esser, H. D. Simon, and D. S. Modha, "The cat is out of the bag: cortical simulations with 10^9 neurons, 10^{13} synapses," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, Nov 2009, pp. 1–12.

[15] P. Messina, "The exascale computing project," *Computing in Science Engineering*, vol. 19, no. 3, pp. 63–67, May 2017.

[16] T. Bhattacharya, T. Brettin, J. H. Doroshow, Y. A. Evrard, E. J. Greenspan, A. L. Gryshuk, T. T. Hoang, C. B. V. Lauzon, D. Nissley, L. Penberthy, E. Stahlberg, R. Stevens, F. Streitz, G. Tourassi, F. Xia, and G. Zaki, "Ai meets exascale computing: Advancing cancer research with large-scale high performance computing," *Frontiers in Oncology*, vol. 9, p. 984, 2019. [Online]. Available: https://www.frontiersin.org/article/10.3389/fonc.2019.00984

[17] "Top500 list statistics," accessed: 2019-09-02. [Online]. Available: https://www.top500.org/statistics/list/

[18] "Green500 list statistics," accessed: 2019-09-02. [Online]. Available: https://www.top500.org/green500/lists/

[19] Z. Chen, J. J. Dongarra, and Z. Xu, "Post-exascale supercomputing: research opportunities abound," *Frontiers of IT & EE*, vol. 19, no. 10, pp. 1203–1208, 2018. [Online]. Available: https://doi.org/10.1631/FITEE.1830000

[20] R. Landauer, "Irreversibility and heat generation in the computing process," *IBM Journal of Research and Development*, vol. 5, no. 3, pp. 183–191, July 1961.

[21] Y. Jun, M. c. v. Gavrilov, and J. Bechhoefer, "High-precision test of landauer's principle in a feedback trap," *Phys. Rev. Lett.*,

vol. 113, p. 190601, Nov 2014. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.113.190601

[22] B. Schroeder and G. A. Gibson, “A large-scale study of failures in high-performance computing systems,” *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 337–350, Oct 2010.

[23] S. Gupta, T. Patel, C. Engelmann, and D. Tiwari, “Failures in large scale systems: Long-term measurement, analysis, and implications,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3126908.3126937

[24] M. V. Wilkes, “The memory wall and the cmos end-point,” *SIGARCH Comput. Archit. News*, vol. 23, no. 4, p. 4–6, Sep. 1995. [Online]. Available: https://doi.org/10.1145/218864.218865

[25] P. Jacob, A. Zia, O. Erdogan, P. M. Belemjian, J. Kim, M. Chu, R. P. Kraft, J. F. McDonald, and K. Bernstein, “Mitigating memory wall effects in high-clock-rate and multicore cmos 3-d processor memory stacks,” *Proceedings of the IEEE*, vol. 97, no. 1, pp. 108–122, Jan 2009.

[26] W. Xu, Y. Lu, Q. Li, E. Zhou, Z. Song, Y. Dong, W. Zhang, D. Wei, X. Zhang, H. Chen, J. Xing, and Y. Yuan, “Hybrid hierarchy storage system in milkyway-2 supercomputer,” *Frontiers of Computer Science*, vol. 8, no. 3, pp. 367–377, Jun 2014. [Online]. Available: https://doi.org/10.1007/s11704-014-3499-6

[27] J.-D. Zhai and W.-G. Chen, “A vision of post-exascale programming,” *Frontiers of Information Technology & Electronic Engineering*, vol. 19, no. 10, pp. 1261–1266, Oct 2018. [Online]. Available: https://doi.org/10.1631/FITEE.1800442

[28] R. Lucas, J. Ang, K. Bergman, S. Borkar, W. Carlson, L. Carrington, G. Chiu, R. Colwell, W. Dally, J. Dongarra, A. Geist, R. Haring, J. Hittinger, A. Hoisie, D. M. Klein, P. Kogge, R. Lethin, V. Sarkar, R. Schreiber, J. Shalf, T. Sterling, R. Stevens, J. Bashor, R. Brightwell, P. Coteus, E. Debenedictus, J. Hiller, K. H. Kim, H. Langston, R. M. Murphy, C. Webster, S. Wild, G. Grider, R. Ross, S. Leyffer, and J. Laros III, "Doe advanced scientific computing advisory subcommittee (ascac) report: Top ten exascale research challenges," 2 2014.

[29] X. Liao, L. Xiao, C. Yang, and Y. Lu, "Milkyway-2 supercomputer: system and application," *Frontiers of Computer Science*, vol. 8, no. 3, pp. 345–356, Jun 2014. [Online]. Available: https://doi.org/10.1007/s11704-014-3501-3

[30] X.-k. Liao, K. Lu, C.-q. Yang, J.-w. Li, Y. Yuan, M.-c. Lai, L.-b. Huang, P.-j. Lu, J.-b. Fang, J. Ren, and J. Shen, "Moving from exascale to zettascale computing: challenges and techniques," *Frontiers of Information Technology & Electronic Engineering*, vol. 19, no. 10, pp. 1236–1244, Oct 2018. [Online]. Available: https://doi.org/10.1631/FITEE.1800494

[31] S. Rumley, D. Nikolova, R. Hendry, Q. Li, D. Calhoun, and K. Bergman, "Silicon photonics for exascale systems," *Journal of Lightwave Technology*, vol. 33, no. 3, pp. 547–562, Feb 2015.

[32] G. Shainer, P. Lui, T. Liu, T. Wilde, and J. Layton, "The impact of inter-node latency versus intra-node latency on hpc applications," 2011.

[33] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout, "It's time for low latency," in *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, ser. HotOS'13. USA: USENIX Association, 2011, p. 11.

[34] R. Underwood, J. Anderson, and A. Apon, "Measuring network latency variation impacts to high performance computing application performance," in *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18. New York, NY, USA: ACM, 2018, pp. 68–79. [Online]. Available: http://doi.acm.org/10.1145/3184407.3184427

[35] M. N. Thadani and Y. A. Khalidi, "An efficient zero-copy i/o framework for unix," Sun Microsystems Laboratories, Inc., Tech. Rep., 1995.

[36] J. Liu, J. Wu, and D. Panda, "High performance rdma-based mpi implementation over infiniband," *International Journal of Parallel Programming*, vol. 32, no. 3, pp. 167–198, 2004. [Online]. Available: http://dx.doi.org/10.1023/B:IJPP.0000029272.69895.c1

[37] R. L. Graham, D. Bureddy, P. Lui, H. Rosenstock, G. Shainer, G. Bloch, D. Goldenerg, M. Dubman, S. Kotchubievsky, V. Koushnir, L. Levi, A. Margolin, T. Ronen, A. Shpiner, O. Wertheim, and E. Zahavi, "Scalable hierarchical aggregation protocol (sharp): A hardware architecture for efficient data reduction," in *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*, Nov 2016, pp. 1–10.

[38] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey, "Jellyfish: Networking data centers randomly," in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 225–238. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/singla

[39] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. G. Azar, and D. Silver, "Rainbow:

Combining improvements in deep reinforcement learning." in *AAAI*, S. A. McIlraith and K. Q. Weinberger, Eds. AAAI Press, 2018, pp. 3215–3222. [Online]. Available: http://dblp.uni-trier.de/db/conf/aaai/aaai2018.html#HesselMHSODHPAS18

[40] T. Tieleman and G. Hinton, "Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude," COURSERA: Neural Networks for Machine Learning, 2012.

[41] H. V. Hasselt, "Double q-learning," in *Advances in Neural Information Processing Systems 23*, J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, Eds. Curran Associates, Inc., 2010, pp. 2613–2621. [Online]. Available: http://papers.nips.cc/paper/3964-double-q-learning.pdf

[42] H. v. Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, ser. AAAI'16. AAAI Press, 2016, pp. 2094–2100. [Online]. Available: http://dl.acm.org/citation.cfm?id=3016100.3016191

[43] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *CoRR*, vol. abs/1511.05952, 2015. [Online]. Available: http://arxiv.org/abs/1511.05952

[44] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, "Dueling network architectures for deep reinforcement learning," in *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, 2016, pp. 1995–2003. [Online]. Available: http://jmlr.org/proceedings/papers/v48/wangf16.html

[45] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[46] R. Ammendola, A. Biagioni, O. Frezza, F. Lo Cicero, A. Lonardo, P. S. Paolucci, D. Rossetti, A. Salamon, G. Salina, F. Simula, L. Tosoratto, and P. Vicini, "APEnet+: high bandwidth 3D torus direct network for petaflops scale commodity clusters," *Journal of Physics: Conference Series*, vol. 331, no. 5, p. 052029, 2011.

[47] *The ExaNeSt project*, accessed: 2019-10-22. [Online]. Available: /http://exanest.eu/

[48] C. E. Leiserson, "Fat-trees: Universal networks for hardware-efficient supercomputing," *IEEE Transactions on Computers*, vol. C-34, no. 10, pp. 892–901, Oct 1985.

[49] X. Yuan, "On nonblocking folded-clos networks in computer communication environments," *2011 IEEE International Parallel Distributed Processing Symposium*, pp. 188–196, 2011.

[50] W. J. Dally, "Virtual-channel flow control," *SIGARCH Comput. Archit. News*, vol. 18, no. 2SI, pp. 60–68, May 1990. [Online]. Available: http://doi.acm.org/10.1145/325096.325115

[51] J. Duato, "A necessary and sufficient condition for deadlock-free adaptive routing in wormhole networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 10, pp. 1055–1067, Oct 1995.

[52] M. Albanese, P. Bacilieri, S. Cabasino, N. Cabibbo, F. Costantini, G. Fiorentini, F. Flore, A. Fonti, A. Fucci, M. Lombardo, S. Galeotti, P. Giacomelli, P. Marchesini, E. Marinari, F. Marzano, A. Miotto, P. Paolucci, G. Parisi, D. Pascoli, D. Passuello,

S. Petrarca, F. Rapuano, E. Remiddi, R. Rusack, G. Salina, and R. Tripiccione, “The ape computer: An array processor optimized for lattice gauge theory simulations,” *Computer Physics Communications*, vol. 45, no. 1, pp. 345 – 353, 1987. [Online]. Available: http://www.sciencedirect.com/science/article/pii/001046558790172X

[53] C. Battista, S. Cabasino, F. Marzano, P. S. Paolucci, J. Pech, F. Rapuano, R. Sarno, G. M. Todesco, M. Torelli, W. Tross, P. Vicini, N. Cabibbo, E. Marinari, G. Parisi, G. Salina, F. D. Prete, A. Lai, M. P. Lombardo, R. Tripiccione, and A. Fucci, “The ape-100 computer: (i) the architecture,” *International Journal of High Speed Computing*, vol. 05, no. 04, pp. 637–656, 1993. [Online]. Available: http://www.worldscientific.com/doi/abs/10.1142/S0129053393000268

[54] F. Aglietti, A. Bartoloni, C. Battista, S. Cabasino, M. Cosimi, A. Michelotti, A. Monello, E. Panizzi, P. Paolucci, W. Rinaldi, D. Rossetti, H. Simma, M. Torelli, P. Vicini, N. Cabibbo, W. Errico, S. Giovannetti, F. Laico, G. Magazzù, and R. Tripiccione, “The teraflop supercomputer apemille: architecture, software and project status report,” *Computer Physics Communications*, vol. 110, no. 1, pp. 216 – 219, 1998. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S001046559700180X

[55] F. Belletti, S. F. Schifano, R. Tripiccione, F. Bodin, P. Boucaud, J. Micheli, O. Pene, N. Cabibbo, S. de Luca, A. Lonardo, D. Rossetti, P. Vicini, M. Lukyanov, L. Morin, N. Paschedag, H. Simma, V. Morenas, D. Pleiter, and F. Rapuano, “Computing for lqcd: apenext,” *Computing in Science Engineering*, vol. 8, no. 1, pp. 18–29, Jan 2006.

[56] R. Ammendola, M. Guagnelli, G. Mazza, F. Palombi, R. Petronzio, D. Rossetti, A. Salamon, and P. Vicini, “APENet: LQCD clusters a la

APE," *Nuclear Physics B-Proceedings Supplements*, vol. 140, pp. 826–828, 2005.

[57] A. Biagioni, F. Lo Cicero, A. Lonardo, P. S. Paolucci, M. Perra, D. Rossetti, C. Sidore, F. Simula, L. Tosoratto, and P. Vicini, "The Distributed Network Processor: a novel off-chip and on-chip interconnection network architecture," *arXiv:1203.1536*, Mar. 2012, http://arxiv.org/abs/1203.1536.

[58] R. Leupers, L. Thiele, A. A. Jerraya, P. Vicini, and P. S. Paolucci, "Shapes:: a tiled scalable software hardware architecture platform for embedded systems," in *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '06)*, Oct 2006, pp. 167–172.

[59] R. Ammendola, A. Biagioni, O. Frezza, F. Lo Cicero, A. Lonardo, P. S. Paolucci, D. Rossetti, F. Simula, L. Tosoratto, and P. Vicini, "QUonG: A GPU-based HPC system dedicated to LQCD computing," in *Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on*, July 2011, pp. 113–122.

[60] R. Ammendola, A. Biagioni, O. Frezza, F. Lo Cicero, A. Lonardo, P. S. Paolucci, D. Rossetti, A. Salamon, F. Simula, L. Tosoratto, and P. Vicini, "A 34 Gbps data transmission system with FPGAs embedded transceivers and QSFP+ modules," in *Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC), 2012 IEEE*, 2012, pp. 872–876.

[61] R. Ammendola, A. Biagioni, O. Frezza, A. Lonardo, F. Lo Cicero, M. Martinelli, P. Paolucci, E. Pastorelli, D. Rossetti, F. Simula, L. Tosoratto, and P. Vicini, "Architectural Improvements and Technological Enhancements for the APEnet+ Interconnect System,"

Journal of Instrumentation, vol. 10, no. 02, p. C02005, 2015. [Online]. Available: http://stacks.iop.org/1748-0221/10/i=02/a=C02005

[62] R. Ammendola, A. Biagioni, O. Frezza, A. Lonardo, F. Lo Cicero, P. Paolucci, D. Rossetti, F. Simula, L. Tosoratto, and P. Vicini, “APEnet+ 34 Gbps Data Transmission System and Custom Transmission Logic,” *Journal of Instrumentation*, vol. 8, no. 12, p. C12022, 2013. [Online]. Available: http://stacks.iop.org/1748-0221/8/i=12/a=C12022

[63] P. Kermani and L. Kleinrock, “Virtual Cut-Through: A New Computer Communication Switching Technique,” *Computer Networks*, vol. 3, pp. 267–286, 1979.

[64] M. Katevenis *et al.*, “The ExaNeSt Project: Interconnects, Storage, and Packaging for Exascale Systems,” in *2016 Euromicro Conference on Digital System Design (DSD)*, Aug 2016, pp. 60–67.

[65] J. A. Boyan and M. L. Littman, “Packet routing in dynamically changing networks: A reinforcement learning approach,” in *NIPS*, 1993.

[66] C. J. C. H. Watkins and P. Dayan, “Q-learning,” *Machine Learning*, vol. 8, no. 3, pp. 279–292, May 1992. [Online]. Available: https://doi.org/10.1007/BF00992698

[67] L. Peshkin and V. Savova, “Reinforcement learning for adaptive routing,” *CoRR*, vol. abs/cs/0703138, 2007. [Online]. Available: http://arxiv.org/abs/cs/0703138

[68] R. J. Williams, “Simple statistical gradient-following algorithms for connectionist reinforcement learning,” *Mach. Learn.*, vol. 8, no. 3-4, pp. 229–256, May 1992. [Online]. Available: https://doi.org/10.1007/BF00992696

[69] L. C. B. III and A. W. Moore, “Gradient descent for general reinforcement learning,” in *Advances in Neural Information Processing Systems 11*, M. J. Kearns, S. A. Solla, and D. A. Cohn, Eds. MIT Press, 1999, pp. 968–974. [Online]. Available: http://papers.nips.cc/paper/1576-gradient-descent-for-general-reinforcement-learning.pdf

[70] A. Valadarsky, M. Schapira, and D. Shahaf, “Learning to route with deep rl,” in *NIPS Deep Reinforcement Learning Symposium*, 2017.

[71] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, “Trust region policy optimization,” in *Proceedings of the 32nd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, F. Bach and D. Blei, Eds., vol. 37. Lille, France: PMLR, 07–09 Jul 2015, pp. 1889–1897. [Online]. Available: http://proceedings.mlr.press/v37/schulman15.html

[72] D. Mukhutdinov, A. Filchenkov, A. Shalyto, and V. Vyatkin, “Multi-agent deep learning for simultaneous optimization for time and energy in distributed routing system,” *Future Generation Computer Systems*, vol. 94, 12 2018.

[73] X. You, X. Li, Y. Xu, H. Feng, and J. Zhao, “Toward packet routing with fully-distributed multi-agent deep reinforcement learning,” *CoRR*, vol. abs/1905.03494, 2019. [Online]. Available: http://arxiv.org/abs/1905.03494

[74] P. Leach, M. Mealling, and R. Salz, “A universally unique identifier (uuid) urn namespace,” 2005, accessed: 2019-10-17. [Online]. Available: http://www.ietf.org/rfc/rfc4122.txt

[75] “Amba specifications,” 2019, accessed: 2019-10-17. [Online]. Available: https://developer.arm.com/architectures/system-architectures/amba/specifications

[76] "Ray distributed software framework," accessed: 2019-10-17. [Online]. Available: https://ray.readthedocs.io/en/latest/index.html

[77] P. Gawlowicz and A. Zubow, "ns3-gym: Extending openai gym for networking research," *CoRR*, vol. abs/1810.03943, 2018. [Online]. Available: http://arxiv.org/abs/1810.03943

[78] G. F. Riley and T. R. Henderson, *The ns-3 Network Simulator*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 15–34. [Online]. Available: https://doi.org/10.1007/978-3-642-12331-3_2

[79] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015. [Online]. Available: http://arxiv.org/abs/1412.6980

[80] F. J. Ridruejo Perez and J. Miguel-Alonso, "Insee: An interconnection network simulation and evaluation environment," in *Euro-Par 2005 Parallel Processing*, J. C. Cunha and P. D. Medeiros, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 1014–1023.

[81] F. Pisani, "Interconnection networks simulations for computing systems dedicated to scientific applications at the exascale," Tesi di laurea, Università degli Studi di Roma, 2015/2016, accessed: 2019-10-17. [Online]. Available: https://apegate.roma1.infn.it/mediawiki/index.php?title=Rome_APE_group_publications

[82] "Openai gym," accessed: 2019-10-17. [Online]. Available: https://gym.openai.com/

[83] M. Matsumoto and T. Nishimura, "Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number genera-

tor," *ACM Trans. Model. Comput. Simul.*, vol. 8, no. 1, pp. 3–30, Jan. 1998. [Online]. Available: http://doi.acm.org/10.1145/272991.272995

[84] J. Lazzaro, J. Wawrzynek, M. Mahowald, M. Sivilotti, and D. Gillespie, "Silicon auditory processors as computer peripherals," *IEEE Transactions on Neural Networks*, vol. 4, no. 3, pp. 523–528, May 1993.