On the Analysis of Computational Delays in Reinforcement Learning-Based Rate Adaptation Algorithms

Ricardo Trancoso(✉), João Pinto, Ruben Queiros, Helder Fontes, and Rui Campos

INESC TEC and Faculdade de Engenharia, Universidade do Porto, Porto, Portugal
{ricardo.j.espirito,ruben.m.queiros,helder.m.fontes,
rui.l.campos}@inesctec.pt

Abstract. Several research works have applied Reinforcement Learning (RL) algorithms to solve the Rate Adaptation (RA) problem in Wi-Fi networks. The dynamic nature of the radio link requires the algorithms to be responsive to changes in link quality. Delays in the execution of the algorithm due to implementional details may be detrimental to its performance, which in turn may decrease network performance. These delays can be avoided to a certain extent. However, this aspect has been overlooked in the state of the art when using simulated environments, since the computational delays are not considered. In this paper, we present an analysis of computational delays and their impact on the performance of RL-based RA algorithms, and propose a methodology to incorporate the experimental computational delays of the algorithms from running in a specific target hardware, in a simulation environment. Our simulation results considering the real computational delays showed that these delays do, in fact, degrade the algorithm's execution and training capabilities which, in the end, has a negative impact on network performance.

Keywords: Reinforcement Learning · Rate Adaptation · Computational Delay

1 Introduction

The IEEE 802.11 standard, commonly known as Wi-Fi, enables the establishment of Wireless Local Area Networks. The standard has had many amendments to keep it up-to-date with the growing requirements of the vast application scenarios. For example, new configuration parameters, such as more spatial streams and larger channel bandwidth, were introduced in the recent standard amendments [1]. When configured correctly for a given link quality, these parameters may increase the efficiency of Wi-Fi networks. One of the parameters that can be changed is the Modulation and Coding Scheme (MCS). Since the link conditions are not static, the MCS has to be changed dynamically. To address this

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2024
Published by Springer Nature Switzerland AG 2024. All Rights Reserved
J.-L. Guisado-Lizar et al. (Eds.): SIMUtools 2023, LNICST 519, pp. 299–310, 2024.
https://doi.org/10.1007/978-3-031-57523-5_23

challenge, multiple Rate Adaptation (RA) algorithms are proposed in the state of the art.

There are multiple heuristic-based RA algorithms. The most commonly used in practice are Minstrel [2] and Iwlwifi [3]. However, these algorithms have shortcomings [4], which limit their ability to select the optimal MCS rate. Algorithms that employ Reinforcement Learning (RL) techniques have emerged as an alternative [4–10]. RL techniques collect observations from the environment. With these observations, the algorithm makes a decision on the best action. Finally, the algorithm calculates a numerical reward, which indicates how suitable the action was. The algorithm learns to repeat actions that yielded a high reward and to avoid actions which led to the opposite. Through this process, the algorithm can eventually find solutions to a problem autonomously and even adapt to unforeseen circumstances. In the context of RA, the environment is the radio link, and the algorithm learns how to configure the Wi-Fi parameters to increase efficiency in dynamic link conditions. Since link conditions are constantly changing, an RL algorithm requires up-to-date observations and actions. Therefore, delays during the execution of an RL-based RA algorithm may be detrimental to its performance.

In the state of the art, computational delays of RL-based RA algorithms have been overlooked. Usually, only the conceptual stages of the algorithms are described, such as the RL model used and the information collected for the observations and the reward; other implementational details are typically not referred. In [4], the authors mention the use of an asynchronous framework to prevent halting of the algorithm while waiting for a process to finish, but do not provide more details. In [5–10], computational delays are not even mentioned. Characterizing the computational delays, which are not modeled in simulation, is relevant since they may impact the algorithm's real-world performance.

The main contributions of this paper are three-fold. First, we bring up awareness to the execution time problem and the importance of considering computational delays when it comes to RL-based RA algorithms; this has been overlooked in the state of the art and may reveal that several existing algorithms can experience degraded performance, when considering the real computational delays. Second, we describe the methods we use to reduce the computational delays of DARA [10], an RL-based RA algorithm, without deviating from its original conceptual design. This can guide implementation alternatives that may be used to reduce the execution time and improve the realistic achievable performance of other RL-based RA algorithms. Finally, we measured the delays of the DARA algorithm in a real scenario and created a framework to simulate these delays. We use this augmented simulation to evaluate and compare the effect these computational delays have on the algorithm's performance in terms of network throughput and other metrics.

The paper is structured as follows. Section 2 characterizes the problem. Section 3 discusses the methods used to reduce the delays of an RL-based RA algorithm, as well as proposes a methodology for comparing the impact of these delays. Section 4 presents the results of the previous comparison. Finally, in Sect. 5, we draw the conclusions.

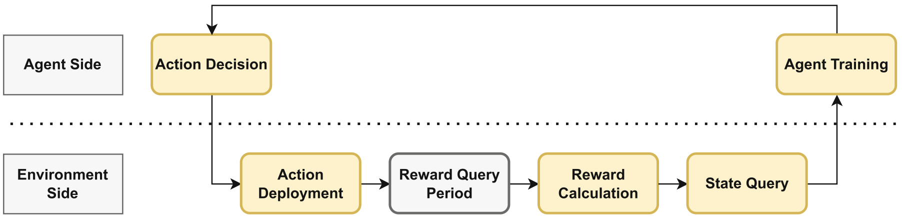


Fig. 1. Overview of the DARA algorithm with its main operations highlighted, and split between agent and environment side.

2 Problem Characterization

In this section, we characterize the problem, using the Data-driven Algorithm for Rate Adaptation (DARA) [10] as a representative state of the art RL-based RA algorithm for Wi-Fi networks. As previously stated, RL relies on observations from the environment, so in a dynamic environment such as the radio link, delays in those observations affect the performance of the algorithm. However, the processes of an RL agent are not instant and have computational delays associated. These delays may not be present in simulation, with the processes instead being functionally instant, as the simulation pauses for the agent to calculate its action, before resuming as normal. This may lead to a significant difference in performance between a simulated and a real environment.

After performing a solution-agnostic analysis of different RL-based RA algorithms, we have identified that they share five key operations: 1) **action decision**, using the agent to choose the optimal expected action; 2) **agent training**, using the reward so that the agent may learn from its previous actions; 3) **action deployment**, applying the chosen action; 4) **reward calculation**, gathering information and calculating the numerical value of the reward; and 5) **state query**, gathering data on the current state of the environment.

In Fig. 1, we can see an overview of the DARA operations, based on the well known RL loop, with the aforementioned five operations highlighted in orange. DARA uses a Deep Q-Network (DQN) [11] algorithm that decides an action out of the eight possible MCS rates defined in IEEE 802.11n, considering a static configuration with Single Input Single Output, 20 MHz channel bandwidth and 800 ns guard interval. DARA decides an action every 100 ms. In [10] DARA was implemented in simulation without taking any computational delays into account. Given that it shares the five key operations with other state of the art algorithms, hereafter we consider it as an application example of this work.

First, DARA was implemented in a real environment to measure experimentally how long was each of its execution steps. This was done on the w-iLab.2 testbed [12] provided by the Fed4FIRE+ project. Our network topology consisted of two nodes: a ZOTAC node as the access point, and a DSS node as the client. The DARA algorithm was deployed on the client node. The client node was equipped with an Intel Core i5 2.6 GHz 4-core CPU, a 60 GB SSD

hard drive, and a 4 GB DDR2 800 MHz PC2-6400 CL6 RAM. The devices ran a Ubuntu 14.04 Linux distribution, on kernel version 3.13.0. Note that the hardware the algorithm is executed on is strongly tied to the observed computational delays. To be able to run DARA in an experimental setting, it required some modifications to collect data regarding the status of the real Wi-Fi link and apply its actions (MCS changes). This first experimental version of DARA did not focus on the optimization of computational delays, as they were thought to be potentially negligible, and would be an object of study. Further details about this implementation will be provided in Sect. 3. This version of DARA will be addressed in this paper as "base DARA", as it serves as an example of an algorithm that overlooks the effect of computational delays on its performance. We performed a preliminary measurement of ten thousand execution steps, and found that the average execution step time of one RL loop was 528.8 ms. This is substantially above the intended 100 ms interval of action which is also used by Minstrel and is related to the coherence time of the Wi-Fi channel. Therefore, this motivated us to study computational delays, possible ways to reduce them, and what their impact in network performance is. For this reason, in this paper we highlight factors that can reduce computational delays and gather information on how much these computational delays impact the performance of RL-based RA algorithms, since the resulting network performance is highly dependent on their responsiveness.

3 Methodology and Alternative Implementations

In this section, the process to reduce these computational delays is described, which can be applicable to a wide range of other RL-based RA algorithms. Afterwards, the methodology to evaluate the impact of computational delays is explained.

3.1 Delay Reduction

DARA's average execution time of 528.8 ms was significantly above the 100 ms step time goal. Our goal was to minimize these delays without changing the algorithm's conceptual design, and without changing the hardware that was used. In what follows, we describe some of the techniques that were applied to reduce delays. We will discuss three different implementation alternatives for data collection, three alternatives for data parsing, a kernel module modification and a configuration parameter through which the more significant delay decreases were achieved. These techniques are focused on the environment side of the DARA algorithm, as shown in Fig. 1. This is because we wanted to preserve DARA's original design, and changes to the agent side imply changes to this design, such as using different agents or neural network model sizes. Because the DARA algorithm is implemented in Python, we could measure the time the program spends inside its functions using the *timeit* module [13]. For the techniques with multiple implementation alternatives, we measured the time their respective functions took to execute, and performed a preliminary analysis to pick the fastest option.

In the course of the reward and state query, the DARA algorithm needs to gather information. This information comes from files in the device that are dynamically updated and need to be read and processed. The files often contain information beyond what is needed, or they are read based on a non-ideal data type (e.g., as a string rather than as an integer). For this reason, in addition to reading the files, it is also important to parse them in order to extract the required data. Because this is a crucial step in the algorithm's functioning, it is worthwhile to explore efficient ways for both collecting the information and parsing it.

We considered three alternative implementations to collect the information:

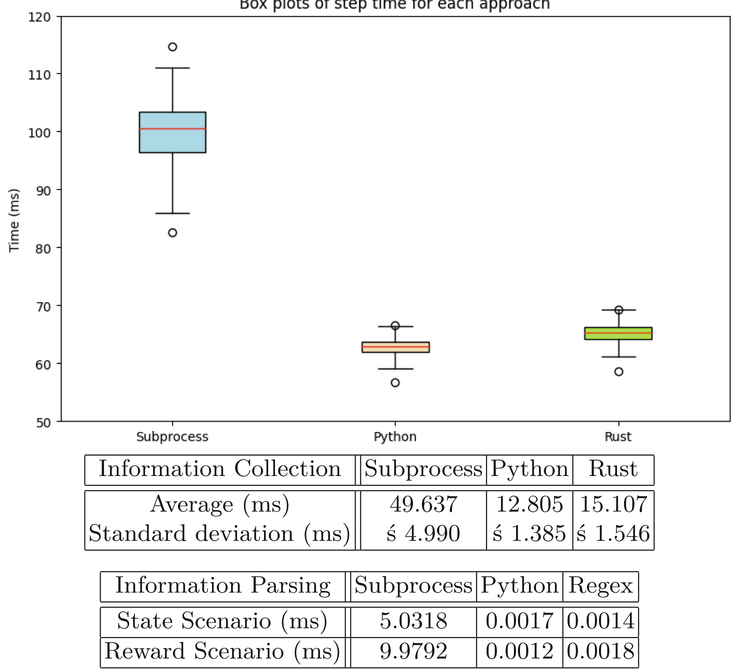
- **Subprocess** – A Python module to spawn and interact with Linux shells, which enables the use of bash commands. This option was used in the base DARA implementation.
- **Pure Python** – A pure Python approach that uses the built-in *read* function in order to access the files. This method may avoid the overheads of the previous option.
- **Rust extension** – A Rust-based approach. Rust is a compiled language and thus may be faster than Python. This option attempts to leverage the potential benefits of using a compiled language to extend Python, since replacing it would require rewriting the algorithm entirely.

As for parsing the information, we evaluated three alternative implementations:

- **Subprocess** – It enables the use of bash commands and it can pipe the output through multiple filtering commands. This option was used in the base DARA implementation to serve as a baseline for comparison.
- **Pure Python** – We considered the use of Python built-in string functions on the output to filter it.
- **Regex** – We considered the use of Regex search patterns to find and extract the information required from the output.

These alternatives are not an exhaustive list of possible implementations. Also note that both the reward and the state information need to be parsed, and the way their data is obtained is different. Therefore, we evaluated the information parsing alternatives in these two scenarios: parsing the reward, and parsing the state information. Our goal is to demonstrate how implementation differences affect computational delays, and compare the alternatives. The results of our preliminary analysis are shown in Fig. 2.

The file read by the algorithm for the state query is present on a Linux Ubuntu 14.04 distribution. However, the file containing statistics for the calculation of the reward is not. The base DARA implementation obtained this information through Minstrel, a heuristic RA algorithm implemented in Linux. Minstrel is implemented using the *mac80211* Linux kernel module. This module is used for managing wireless devices, thus having access to low-level information on wireless connections. Nevertheless, this low-level information is usually locked



Information Collection	Subprocess	Python	Rust
Average (ms)	49.637	12.805	15.107
Standard deviation (ms)	ś 4.990	ś 1.385	ś 1.546

Information Parsing	Subprocess	Python	Regex
State Scenario (ms)	5.0318	0.0017	0.0014
Reward Scenario (ms)	9.9792	0.0012	0.0018

Fig. 2. Statistics on information collection and parsing alternatives.

to the kernel, and cannot be readily accessed. The base DARA implementation read a table that is computed every 100 ms, even if Minstrel is disabled. This table, unlike the data in the kernel module, is accessible in user-space, although the data is not as up-to-date due to the 100 ms period. This step alone forces the base implementation to take at least 100 ms on each action decision.

For these reasons, our enhanced implementation of DARA modifies the mac80211 module to create a virtual empty file that when read from user space, runs a kernel space function that exposes the values stored in the variables related to frame transmission successes and attempts directly from the kernel. We obtain accurate and current information for the calculation of the reward without being limited by the 100 ms waiting period associated with the table update. For the sake of testing other approaches, we used a preliminary time period of 50 ms instead. However, in practice, this query can be done during the "sleep" time of the algorithm between execution steps, i.e., after the algorithm finishes applying its chosen MCS, but before the next execution step starts. Therefore, this delay can be reduced to functionally zero, which is only possible because the algorithm no longer has to wait for the table update as a result from this kernel module modification.

In Fig. 2, we have a box plot for each approach (Subprocess, Python and Rust) that represents the time of a full execution step. The approaches are rep-

resented on the x-axis, while the y-axis represents the time in milliseconds that each approach takes for each algorithm execution step. Do note that the box plot accounts for the total execution step, which includes the 50 ms mentioned previously. This is why the y-axis starts at 50 ms. The lower and upper boundaries of the boxes represent the first and third quartiles, respectively. The lower and upper whiskers represent the minimum and maximum recorded value within 1.5 times the interquartile range below or above the first and third quartiles, respectively. The dots represent the minimum and maximum recorded values. We also have two tables with the time each approach takes for information collection, and parsing. The tables only aggregate the time spent specifically for the collection or parsing instead of the total time. Given the results in Fig. 2, our final implementation used Python to read the files and parse the reward statistics file, and Regex to parse the state query file. Additionally, it made use of the changes to the mac80211 module, which removed the previous limitation of taking 100 ms minimum.

In the end, while the base version of DARA that did not factor computational delays had a response time of 528.8 ms, after changes to the implementational details, we managed to achieve an average response time of 34.8 ms in the same hardware, while maintaining the algorithm's conceptual design. This new version of DARA with reduced delays was called Enhanced DARA or E-DARA.

3.2 Simulation of Delays

After successfully reducing the delays by an order of magnitude, we had two versions of DARA ready that we could compare in addition the original simulation version without delays, those being the base DARA version and E-DARA. The execution time of ten thousand steps of each version were collected experimentally on real devices. To evaluate the impact of computational delays in network performance, we also measured the throughput of each version of DARA. To ensure the reproducibility of wireless channel conditions, we developed a simulation scenario in ns-3. However, this time, the delays will also be simulated, based on the data received from the experimental implementation. This was done by creating a normal distribution of the delays based on the collected average and variance of the measured delays. In this new simulation, after the agent receives the observation, a random delay based from its normal distribution is introduced before the action is applied. Therefore, the resulting effect is as if the simulation no longer stopped for the agent to function, similar to a real environment.

We collected the data on 100 simulations with different seeds and thus, random delays. The simulated scenario has two nodes, one static where the agent is deployed, and another one moving away at a constant velocity, up to a distance of 1100 m, and the throughput between the two nodes is measured. This results in a smooth decrease of SNR throughout the simulation which ensures that the node will cover the whole spectrum of MCS usage, starting with a high MCS at short distances, and ending on a low MCS at longer distances. The nodes communicate using the IEEE 802.11n standard, with a 20 MHz channel band-

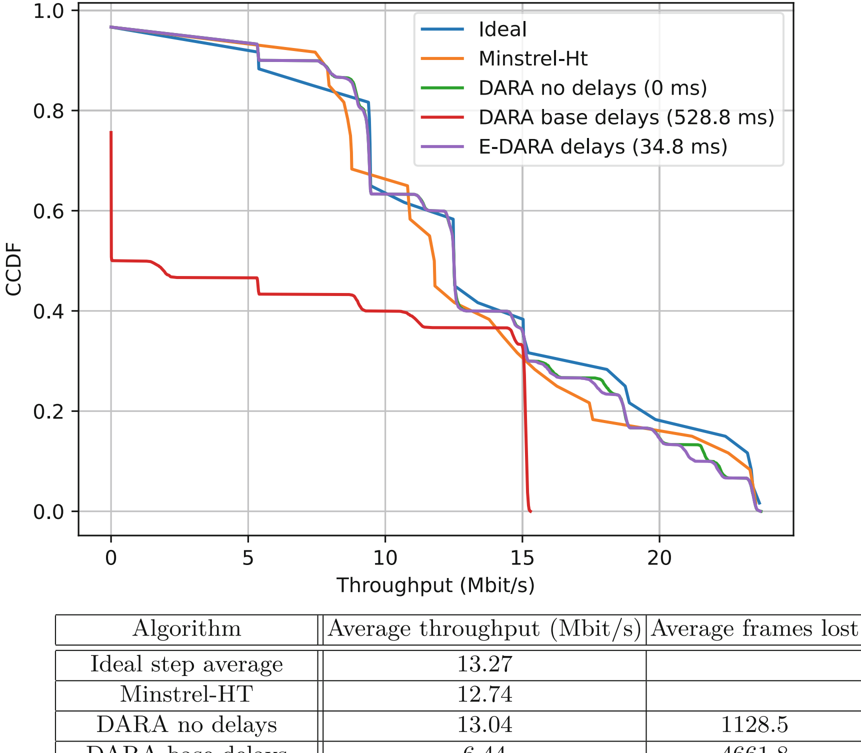
width at a frequency of 5180 MHz. The Friis propagation model and a constant propagation delay are considered.

We then compared the network throughput and frame loss statistics achieved by multiple versions of the DARA algorithm. Those DARA versions are: 1) Simulation DARA with no delays during training and exploitation; 2) DARA with its base 528.8 ms delays during training and exploitation; 3) E-DARA with its reduced 34.8 ms delays during training and exploitation; and 4) E-DARA with no delays during training but 34.8 ms delays during exploitation. This last version of DARA was included to provide a preliminary perspective on training an algorithm in ideal simulated conditions, but then applied to a real scenario. If this training process without delays is more effective, it could be a helpful resource to keep in mind since as long as the simulation is accurate enough, it may be worthwhile to train an agent in simulation before deploying it to a real environment. For further comparison, we also include the simulated network performance from using the ns-3 Ideal RA algorithm, and the ns-3 implementation of Minstrel-HT, as those were the points of reference in the original DARA paper [10].

4 Results

In Fig. 3 we have a complementary cumulative density function of the throughput of each version of the DARA algorithm, as well as a table displaying the average throughput and the average frames lost, which refers to the average number of frames that are lost in total for each simulation. The Minstrel-HT and Ideal RA algorithms are shown just as a reference, and do not consider any computational delays. In Fig. 4, we have the throughput over time of an example of one of the hundred simulations that were performed. Note that due to a warm-up time in the simulation, the plot starts at 4 s. DARA with no delays and with enhanced delays are very similar, both performing consistently better in throughput than Minstrel-HT. They are even better than ns-3 's Ideal algorithm in a few cases. This is because Ideal's goal is not to maximize throughput, but to maintain the Frame Error Ratio below a given threshold. However, DARA with base delays falls drastically behind all the other algorithms, being 49.54% worse than DARA with enhanced delays. The original implementation in a simulation that did not take the delays into account failed to portray this poor performance.

These results indicate that considering and reducing computational delays is important due to their impact on the performance of the algorithm. Completely overlooking the delays may result in simulations that cannot identify the inadequacy of an algorithm for real scenarios, such as was the case for DARA with base delays. Although delays can be reduced and result in a negligible difference in throughput, the version of DARA with enhanced delays still had 5.36% higher frame losses than the version without any delays. It is important to note that these results come from a well-behaved scenario where one device slowly moves away from another, in ideal propagation conditions. In the real world, a more dynamic environment may prove that the network performance is even more sensitive to delays.



Algorithm	Average throughput (Mbit/s)	Average frames lost
Ideal step average	13.27	
Minstrel-HT	12.74	
DARA no delays	13.04	1128.5
DARA base delays	6.44	4661.8
DARA enhanced delays	13.00	1189.0

Fig. 3. Comparison of network performance of Ideal, Minstrel-HT, DARA with no delays, DARA with base delays and DARA with reduced delays.

In Fig. 5 we show a comparison between E-DARA versions, in which one was trained with no delays, and the other was trained with 34.8 ms delays. During exploitation, however, both versions had similar delays. Despite this, we can see that the version that was trained with no delays performs slightly better. One interesting implication of this result is that if the simulation environment is accurate enough, it is possible to train the algorithm with no delays to achieve a higher performance when it is deployed to a real environment afterwards.

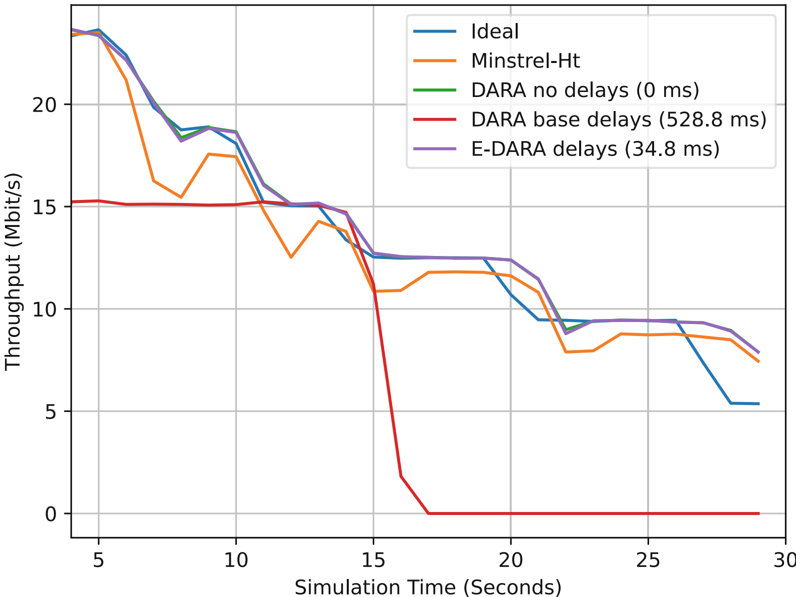
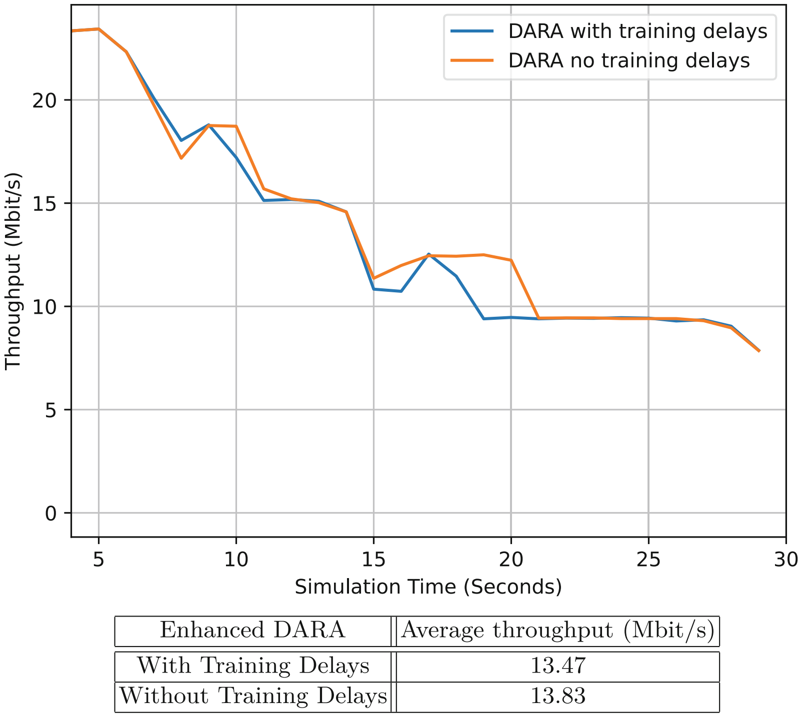


Fig. 4. Example of the throughput of Ideal, Minstrel-HT, DARA with no delays, DARA with base delays and DARA with reduced delays in one simulation.



Enhanced DARA	Average throughput (Mbit/s)
With Training Delays	13.47
Without Training Delays	13.83

Fig. 5. Comparison of throughput between DARA trained with and without delays.

5 Conclusions

Computational delays on RL-based RA algorithms are often overlooked in the state of the art. Information about their impact on the performance of these algorithms is hard to find, and this work intends to fill this void and encourage further work on the topic. We measured experimentally the computational delays of DARA, an example of an RL-based RA algorithm. Afterwards, we managed to reduce the execution time by one order of magnitude, without changing the hardware or the algorithm's core functionality.

The data on the delays allowed us to replicate them and their effect in simulation, bridging the gap between results obtained experimentally and those obtained in simulated scenarios. The importance of these delays was evidenced by comparing different versions of the DARA algorithm, which showed there is a difference in algorithm performance that may not be immediately apparent, between when computational delays are considered, and when they are not. It may be possible to reduce the delays to the point where their effect is close to negligible in terms of throughput and frames lost. However, they may impact separate metrics differently. This emphasizes that more attention on computational delays when discussing RL algorithms may be warranted. Furthermore, the lack of analysis of an algorithm's delays may result in its unsuitability for experimental scenarios to go unnoticed.

Acknowledgments. This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020. The third author thanks the funding from FCT, Portugal under the PhD grant 2022.10093.BD

References

1. IEEE standard for information technology–telecommunications and information exchange between systems local and metropolitan area networks–specific requirements part 11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications amendment 1: Enhancements for high-efficiency WLAN. IEEE Std. 802.11ax-2021 (Amendment to IEEE Std. 802.11-2020), pp. 1–767 (2021). https://doi.org/10.1109/IEEESTD.2021.9442429
2. The minstrel rate control algorithm for mac80211. https://wireless.wiki.kernel.org/en/developers/documentation/mac80211/ratecontrol/minstrel. Accessed 1 Feb 2022
3. Grünblatt, R., Guérin-Lassous, I., Simonin, O.: Simulation and performance evaluation of the Intel rate adaptation algorithm. MSWiM (2019). https://doi.org/10.1145/3345768.3355921
4. Chen, S.C., Li, C.Y., Chiu, C.H.: An experience driven design for IEEE 802.11ac rate adaptation based on reinforcement learning. In: IEEE INFOCOM 2021 (2021). https://doi.org/10.1109/infocom42981.2021.9488876
5. Karmakar, R., Chattopadhyay, S., Chakraborty, S.: SmartLA: reinforcement learning-based link adaptation for high throughput wireless access networks. Comput. Commun. (2017). https://doi.org/10.1016/j.comcom.2017.05.017

6. Karmakar, R., Chattopadhyay, S., Chakraborty, S.: IEEE 802.11ac Link adaptation under mobility. In: 2017 IEEE 42nd Conference on LCN (2017). https://doi.org/10.1109/lcn.2017.90
7. Karmakar, R., Chattopadhyay, S., Chakraborty, S.: An online learning approach for auto link-configuration in IEEE 802.11ac wireless networks. Comput. Netw. (2020). https://doi.org/10.1016/j.comnet.2020.107426
8. Peserico, G., Fedullo, T., Morato, A., Vitturi, S., Tramarin, F.: Rate adaptation by reinforcement learning for Wi-Fi industrial networks. In: 2020 25th IEEE International Conference on ETFA (2020). https://doi.org/10.1109/etfa46521.2020.9212060
9. Karmakar, R., Chattopadhyay, S., Chakraborty, S.: Dynamic link adaptation in IEEE 802.11ac: a distributed learning based approach. In: 2016 IEEE 41st Conference on LCN (2016). https://doi.org/10.1109/LCN.2016.20
10. Queiros, R., Almeida, E., Fontes, H., Ruela, J., Campos, R.: Wi-Fi rate adaptation using a simple deep reinforcement learning approach. In: 2022 IEEE ISCC (2022). https://doi.org/10.1109/iscc55528.2022.9912784
11. Mnih, V., et al.: Playing Atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602 (2013)
12. w-iLab.2 hardware and inventory. https://doc.ilabt.imec.be/ilabt/wilab/hardware.html. Accessed 26 Jan 2022
13. Timeit module documentation. https://docs.python.org/3/library/timeit.html. Accessed 21 Dec 2022