ns3-marl: Scalable Multi-Agent Reinforcement Learning for Realistic Network Simulations

Halefom Kahsay and Young-June Choi
Department of Artificial Intelligence, Ajou University, Suwon 16499, South Korea
Email: {halefom, choiyj}@ajou.ac.kr

***Abstract*—Multi-agent reinforcement learning (MARL) is increasingly applied to networking research, where direct experimentation is rarely feasible and simulation-based training has become the standard approach for developing and evaluating adaptive network protocols. Coupling discrete-event simulators written in C++, such as *ns*-3, with Python-based MARL frameworks such as RLlib, PettingZoo, and EPyMARL requires bridging a language and process boundary, typically through inter-process communication mechanisms such as shared memory or ZeroMQ sockets. In prior simulation-to-learning integration frameworks, extending this per-decision blocking mechanism to multi-agent use serializes agent decisions, increases training wall time, and can produce correctness failures including observation timing drift, stale-data corruption, and channel deadlock. We present *ns3-marl*, a shared-memory protocol in which all N agents submit observations to a local buffer without blocking, while a single synchronization at the epoch boundary exchanges the complete batch with the training framework. This reduces synchronization from one blocking barrier per agent to one barrier per epoch, eliminating both wall-time overhead and correctness failures while preserving standard synchronous MARL semantics. We validate *ns3-marl* across five *ns*-3 scenarios, ranging from a synthetic IPC benchmark to IEEE 802.11ax contention-window control, with zero IPC failures and convergent policies in all runs.**

***Index Terms*—Multi-agent reinforcement learning, *ns*-3, inter-process communication, Wi-Fi 6, discrete-event simulation.**

I. Introduction

Reinforcement learning (RL) and its multi-agent extension (MARL) are increasingly applied to communication networking research, spanning resource allocation [1], scheduling, and handover management in next-generation wireless systems [2]. Direct experimentation on live networks is rarely feasible due to cost, safety, and reproducibility constraints; simulation-based training has therefore become the standard approach for developing and evaluating adaptive network protocols. Among simulation platforms, *ns*-3 [3], [4] is the de-facto standard for protocol-level networking research. Its discrete-event engine faithfully reproduces medium-access control and physical-layer (MAC/PHY) dynamics — contention, retransmissions, queueing, and scheduler interactions — that are critical for validating whether learned policies transfer to real deployments, particularly for standards-aligned work on IEEE 802.11ax (Wi-Fi 6) and beyond [2].

Coupling a discrete-event network simulator with a MARL training framework imposes a two-language bridge that is structurally unavoidable, not a design choice. High-fidelity simulators execute causally ordered MAC/PHY events in C++, whereas Python-native MARL frameworks such as RLlib, PettingZoo, and EPyMARL coordinate rollouts, collect experience, and compute policy updates [5]–[7]. Because the simulator and trainer commonly run as independent OS processes with protected address spaces, observations and actions must cross an explicit communication boundary [8].

A number of simulation-to-learning integration frameworks address this boundary. *ns3-gym* introduced a Gym interface for *ns*-3 using ZeroMQ sockets [9], *ns3-ai* replaced sockets with Boost shared memory to reduce per-step latency [8], and DEFIANCE extends ns3-ai with native MARL support and RLlib integration [5], [10]. These frameworks make learning-based *ns*-3 experiments practical, but their common sequential exchange model serves each agent decision through an independent blocking exchange. When this model is extended to an N-agent synchronous MARL setting, each decision epoch serializes N waits across the simulator-learning boundary. This increases training wall time and can produce correctness failures at $N>1$, including observation timing drift, stale-data corruption when the simulator exits mid-episode, and channel deadlock.

Across initial multi-agent training runs in several *ns*-3 scenarios, sequential per-epoch wall time grows linearly with agent count, consistent with $O(N)$ scaling. The per-agent IPC transfer itself completes in milliseconds, but the sequential structure forces the Python MARL loop to wait for each agent's simulation-time event in turn [8], so process wall time can far exceed the simulated episode duration. At its core, the mismatch is one of synchronization philosophy because common MARL frameworks impose a *synchronous* step contract [5]–[7], whereas the *ns*-3 event engine is *asynchronous* and produces decision callbacks as simulation time advances. We identify this repeated blocking wait as an *orchestration mismatch* and address it with *ns3-marl*, an epoch-batch shared-memory IPC protocol for *ns*-3 MARL simulation.

This paper makes three contributions:

- We present the design and implementation of *ns3-marl* as a shared-memory MARL execution path built on ns3-ai and DEFIANCE [8], [10].
- We validate the protocol across five *ns*-3 scenarios, from a synthetic IPC benchmark to IEEE 802.11ax contention-window control, showing stable execution with zero IPC failures.
- We package the system as a reproducible research tool with scripts, examples, and measurement artifacts suitable for extending *ns*-3 MARL studies.

The remainder of this paper is organized as follows. Sec-

979-8-3195-0167-7/26/$31.00 ©2026 IEEE

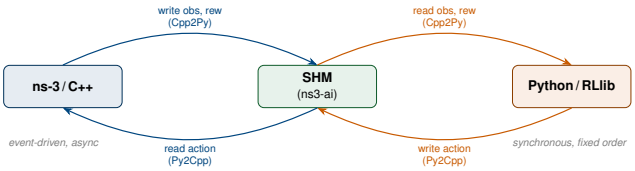


Fig. 1: ns3-ai shared-memory data-flow. `Cpp2Py` and `Py2Cpp` are passive memory regions written and read by C++ and Python respectively; semaphores control read/write ordering. The shared-memory node is a relay, not an active sender.

tion II summarizes simulation-to-learning integration frameworks and the orchestration mismatch that motivates this work. Section III presents the ns3-marl architecture and vectorized epoch IPC protocol. Section IV reports validation across five *ns*-3 scenarios, and Section V concludes.

II. Background

Realizing the potential of MARL across the wireless protocol stack, including IEEE 802.11 (Wi-Fi), LTE, and 5G scenarios supported by *ns*-3 [3], [4], requires embedding MARL agents directly inside the simulation loop. Within *ns*-3, ns3-gym introduced a Gym-compatible ZeroMQ bridge [9], ns3-ai replaced sockets with Boost shared memory [8], and DEFIANCE added native MARL support with RLlib integration [5], [10].

The ns3-ai data-flow, shown in Fig. 1, uses shared-memory buffers for C++-to-Python observations and Python-to-C++ actions. Although this transport reduces communication latency, the inherited per-agent blocking exchange still serializes multi-agent epochs. The bottleneck therefore arises not from memory transfer alone, but from a scheduling conflict between synchronous MARL step semantics and asynchronous *ns*-3 decision callbacks [5], [6]. This can increase wall time and make joint observations internally time-incoherent when later agents are observed after earlier agents have acted.

III. System Architecture

We organise ns3-marl into three separable components with clean dependency boundaries (Fig. 2). The *IPC backend* owns the epoch buffer and a purpose-built batch-epoch SHM adapter that multiplexes all N submissions over the ns3-ai SHM channel [8]; it contains no *ns*-3 includes, keeping the core portable and unit-testable without an *ns*-3 build. The *MARL agent layer* provides a Gymnasium-compatible base class that exposes a standard step-and-reset interface consumable by any cooperative MARL training stack—RLlib [5], EPyMARL [7], PettingZoo [6], and similar frameworks—routing all agent interactions through the batch protocol transparently. The *scenario collection* contains the five validated scenarios and benchmark runners used in this paper. The IPC backend's batch protocol is motivated by two compounding limitations of the sequential per-agent synchronization model it replaces: $O(N)$ per-epoch synchronization cost and intra-epoch time incoherence across agents.

The two IPC execution models are contrasted in Fig. 4. In the *sequential* model [8], [10], the *ns*-3 event loop issues N independent blocking calls: for each agent i, the simulator

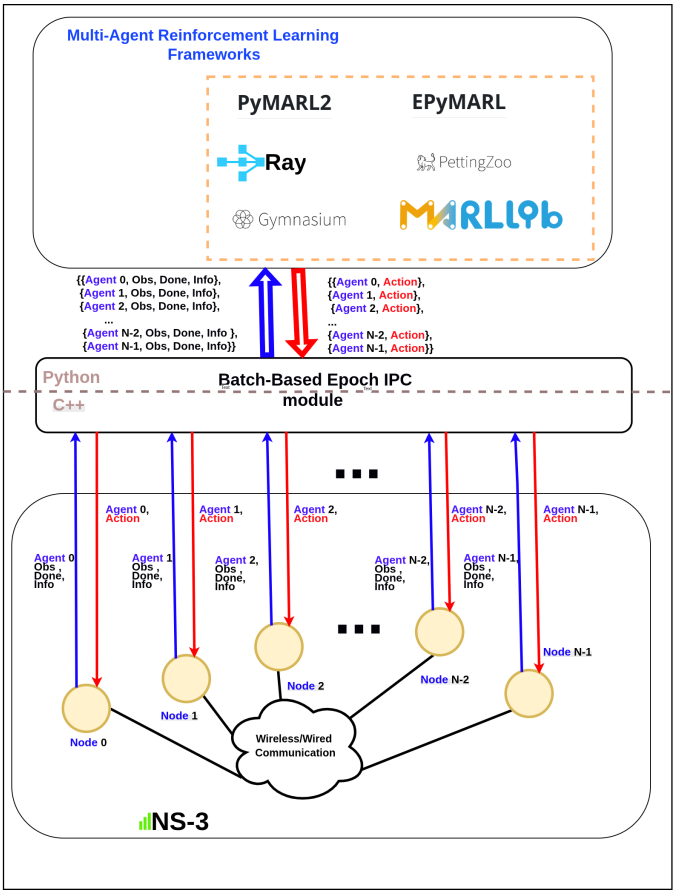


Fig. 2: ns3-marl system architecture. N submit calls populate the local epoch buffer; one flush performs the single shared-memory round-trip. Python processes all N observations and returns all N actions in one reply.

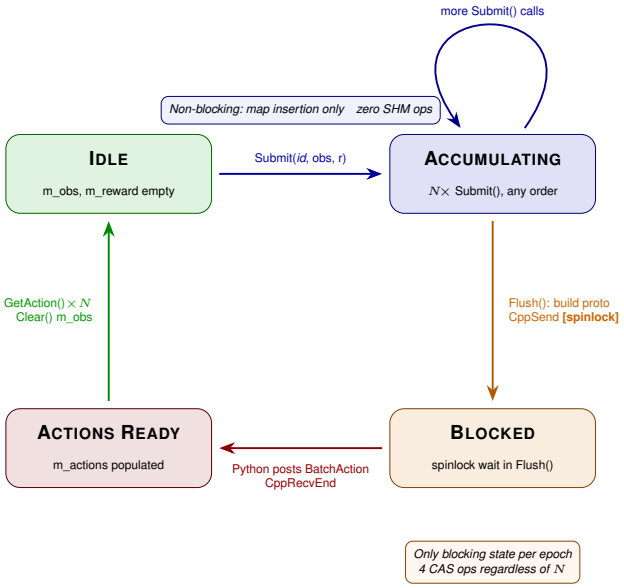


Fig. 3: Per-epoch protocol state machine.

writes observations to shared memory, blocks at the compare-and-swap (CAS) spinlock, and Python computes action a_i, and the simulator resumes. Because the event loop cannot

advance during any blocked call, the N round-trips are strictly serialized. Each per-agent latency decomposes as $\ell_i = W_i + \delta_i$; δ_i is the IPC exchange itself (shared-memory write, CAS spinlock acquisition, and action read-back), while W_i is the wall-clock time Python spends waiting for *ns*-3 to advance its discrete-event clock to agent i's next decision point—a cost driven by simulation computation, not by policy inference time. Wi-Fi 6 traces in Section IV-C confirm that W_i can exceed δ_i by nearly three orders of magnitude. The sequential protocol also violates Dec-POMDP joint-observation semantics [11]: within one epoch, each observation o_i is delivered at a distinct simulation time τ_i; therefore, the assembled state vector $[o_1(\tau_1), \ldots, o_N(\tau_N)]$ is time-incoherent across agents.

The *vectorized epoch* model (Fig. 3) resolves both failure modes by decoupling observation collection from IPC synchronization. In the ACCUMULATING phase, each agent calls `Submit()`, a non-blocking local copy of its observation and reward into a C++ epoch buffer that does not touch the spinlock, semaphore, or shared-memory segment. After all N submissions, `Flush()` enters the only BLOCKED phase, serializes the complete epoch buffer via Protocol Buffers into one shared-memory message, blocks once while Python computes actions for all N observations jointly, and receives the full action batch in one reply. The ACTIONS READY phase then serves each agent by a local map lookup with no IPC barrier before the buffer is cleared for the next epoch. At steady state the flush round-trip completes in approximately 1.4–1.9 ms, of which over 98% is Python policy computation; shared-memory transfer and Protocol Buffer serialization account for less than 2%. The shared-memory transfer size grows as $O(N \times |o_i|)$, but the synchronization *barrier count*—the key bottleneck—is $O(1)$.

The batch message carries the epoch time, a monotone epoch identifier, an episode-termination flag, and one record per agent containing its observation, reward, and missing-agent status; the reply carries the matching epoch identifier and one action per agent. On the Python side, the MARL agent layer exposes this exchange as the standard multi-agent tuple of observations, rewards, termination flags, truncation flags, and diagnostic info [5]. Agent identifiers remain stable across resets, and episode termination is driven by the simulator-provided batch flag so that the C++ event loop and Python training framework agree on episode boundaries.

Protocol correctness follows from three invariants: all submissions for an epoch precede its flush, exactly one flush is issued per epoch, and missing agents are explicitly marked rather than waited on indefinitely. These invariants restore Dec-POMDP joint-observation semantics [11] because all N observations are exchanged atomically for the same epoch interval $[t_k, t_k + \Delta t]$. They also improve reliability: if the simulator exits, Python either receives a complete epoch batch or observes failure at the single synchronization point, instead of blocking on a partially served per-agent sequence with no spinlock timeout [8]. A fully asynchronous flush-and-poll variant is designed but excluded from the reported benchmarks.

Fig. 4 summarizes the resulting complexity difference. In

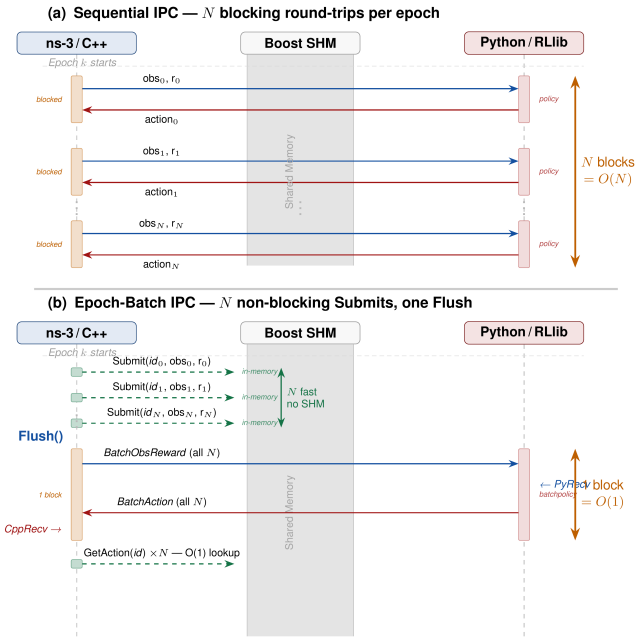


Fig. 4: IPC protocol comparison. (a) Sequential protocol. (b) Vectorized epoch IPC protocol.

the sequential path, each epoch incurs one shared-memory round-trip per agent, so the synchronization cost is $T_{\text{seq}} = \sum_{i=0}^{N-1}(W_i + \delta_i)$. In ns3-marl, all `Submit()` calls are local writes and only `Flush()` crosses the process boundary, giving $T_{\text{batch}} = \max_i(W_i) + \delta_{\text{batch}}$. The reduction is therefore in synchronization barriers, from N per epoch to one per epoch, while transferred payload size still grows with the number and size of agent observations.

IV. VALIDATION ACROSS SCENARIOS

We validate the *ns3-marl* vectorized epoch IPC protocol introduced in Section III against three questions. First, does replacing the N-barrier sequential delivery with a single epoch-boundary flush actually reduce per-epoch synchronization cost from $O(N)$ to $O(1)$? Second, does this reduction hold when real simulation physics or network computation accompanies each epoch, not just synthetic arithmetic? Third, can the protocol sustain stable multi-agent PPO training to convergence on a real wireless control task? The following subsections combine an IPC-isolated benchmark, broader scenario coverage, and a real Wi-Fi 6 MARL training task. Across all five scenarios, the total synchronization event count exceeds 106,200 with zero IPC errors; per-scenario readiness gates (protocol contract, episode accounting, action sensitivity, multi-episode hygiene) passed at all tested N values.

A. *addition-batch: IPC Scaling Benchmark*

The addition-batch scenario is a synthetic benchmark with negligible simulation computation, designed to isolate IPC overhead. Each agent observes a scalar $o_i^{(k)} \in \{0, \ldots, 9\}$, selects $a_i^{(k)} \in \text{Discrete}(4N+9)$, and receives

$$r_i^{(k)} = \exp\left(-|a_i^{(k)} - o_i^{(k)} - i|\right), \tag{1}$$

TABLE I: IPC wall-time benchmark (addition-batch, direct measurement)

N	Batch (μs)	Sequential (μs)	Speedup
1	415	217	0.52×
4	508	793	1.56×
8	544	1,640	3.01×
16	662	3,198	4.83×
32	960	6,413	**6.68**×

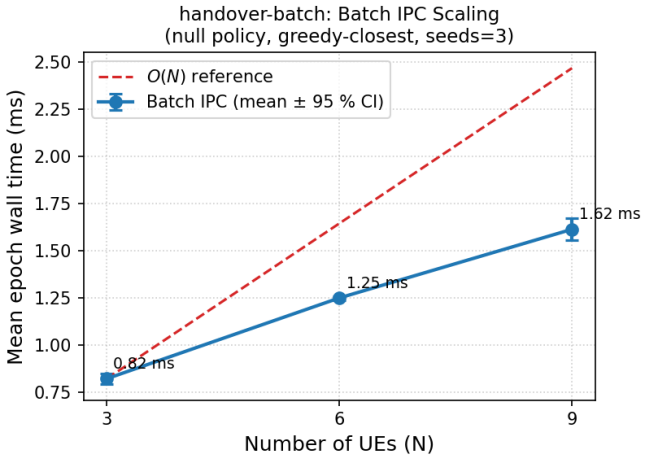


Fig. 5: IPC wall-time scaling for addition-batch.

which is maximized at the oracle action $a_i^* = o_i + i$. Measurements are averaged over steady-state epochs excluding warm-up. Both paths run identical arithmetic logic over $N \in \{1, 4, 8, 16, 32\}$; only the IPC protocol differs. Table I and Fig. 5 report the results.

A log-linear fit yields exponents of $N^{0.06}$ for batch and $N^{1.0}$ for sequential IPC. At $N = 1$, batch is slower because Protocol Buffer serialization exceeds a single ns3-ai round-trip; this cost is amortized for $N \geq 4$, where the batch path crosses over and reaches 6.68× speedup by $N = 32$.

B. Scenario Coverage Beyond Synthetic IPC

Three additional scenarios test whether the protocol remains useful when the simulator performs non-trivial work between decision epochs. The balance2-batch scenario embeds differential-equation physics for $N \in \{3, 6, 9\}$ inverted-pendulum agents and achieves a 3-seed mean speedup of 3.4× at $N = 9$. The handover-batch scenario exercises LTE mobility control with X2 handover signalling over four eNBs and achieves approximately 35% IPC wall-time saving at $N = 9$. The lte-batch scenario implements the complete DEFIANCE four-application pattern for RF-layer control and achieves approximately 32% saving at $N = 9$, while also completing 3-seed PPO training. These smaller gains, relative to the IPC-isolated benchmark, are expected because physics integration and LTE SINR callbacks add simulation-side work; the batch protocol still preserves one synchronization barrier per epoch.

Table II collects the IPC comparison across all four benchmarking scenarios at $N = 9$. In all cases, the batch path is sublinear in N; the residual growth beyond the batch-only exponent ($N^{0.06}$) is attributable to simulation-side computation (per-epoch physics and LTE signal-to-interference-plus-noise ratio

TABLE II: Cross-scenario IPC comparison at $N = 9$ (null policy; balance2-batch: 3-seed mean; addition-batch: extrapolated from $N = 8, 16$)

Scenario	Seq. est. (μs)	Batch (μs)	Saving
addition-batch	∼1,850	∼560	∼3.3×
balance2-batch	960	283	3.4×
handover-batch	∼2,469	1,615	∼35%
lte-batch	∼2,160	1,462	∼32%

(SINR) callbacks), not to IPC overhead. The vectorized epoch protocol transforms rather than eliminates the waiting cost: $T_{\text{batch}} \approx \max_i(W_i) + \delta$ versus $T_{\text{seq}} \approx \sum_i (W_i + \delta_i)$; so, the epoch flush pays the cost of the single longest simulation-side wait, not their sum. In the addition-batch benchmark, 98% of the epoch synchronization duration is Python policy computation and less than 2% is shared-memory and serialization overhead, with a mean Flush duration of $\delta_{\text{Flush}} = 1.4$–$1.9$ ms. These four scenarios confirm affirmative answers to questions 1 and 2 above; Section IV-C addresses question 3.

C. wifi6-batch: IEEE 802.11ax CW Control

The wifi6-batch scenario is the most complex validation case and the primary real-network MARL task. Random-policy traces show why sequential per-agent delivery is untenable at MAC-layer timescales: the simulator reports only 17–19 ms for the actual data exchange, while Python observes 2,548 ms for agent 5 at 0.75 s, 5,662 ms for agent 6 at 0.80 s, and up to 16,395 ms for agent 0 at 1.60 s for the same agent-step accounting path. This gap is simulation-advance wait, not policy computation, and ns3-ai's userspace busy-wait spinlock keeps the waiting C++ thread active until Python writes the completion flag [8].

For N agents, the M/G/1 server-vacation model [12], [13] captures the resulting slowdown:

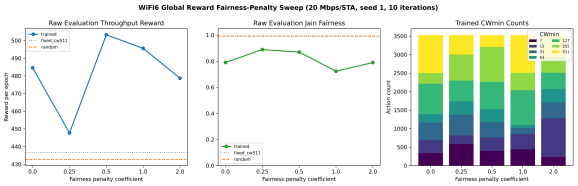
$$\rho = \frac{T_{\text{wall}}}{T_{\text{sim}}} = 1 + \frac{\bar{W} N}{\Delta t}, \tag{2}$$

where $\bar{W}$ is the mean per-agent simulation-advance wait and Δt is the epoch interval. The observed Wi-Fi 6 traces are consistent with this model.

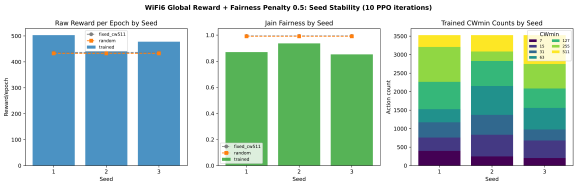
The Wi-Fi 6 scenario is formulated as a Dec-POMDP [11] with $N = 6$ homogeneous STA agents operating under IEEE 802.11ax in a single BSS; the AP acts as epoch supervisor and is not part of the agent set. Each epoch k spans 100 ms; the episode runs for 30 s (295 epochs), with an association warm-up before the first epoch. Each STA i observes a local MAC state vector:

$$o_i^{(k)} = \left[q_i^{(k)},\ t_i^{(k-1)},\ d_i^{(k-1)},\ w_i^{(k)} \right], \tag{3}$$

where q_i is queue depth, t_i is the previous-epoch successful transmission count, d_i is the previous-epoch drop count, and w_i is the current minimum contention window value. The seven actions map to $\text{CW}_{\min} \in \{7, 15, 31, 63, 127, 255, 511\}$, and the implementation sets $\text{CW}_{\max} = 2\,\text{CW}_{\min} + 1$. The uplink traffic load is 20 Mbps/STA (total 120 Mbps), placing the system in the high-contention regime where CW adaptation has measurable effect.



(a) Fairness sweep.



(b) Multi-seed stability.

Fig. 6: Wi-Fi 6 training with $\lambda_{\mathrm{cw}} = 20$ and $\lambda_f = 0.5$.

TABLE III: Wi-Fi 6 evaluation ($\lambda_f = 0.5$, seeds 1–3 vs. baselines)

Policy	Reward/epoch	Jain J
Trained ($\mu \pm \sigma$, seeds 1–3)	473.91 ± 31.39	0.886 ± 0.044
Fixed $\mathrm{CW}_{\min} = 511$	436.60	0.994
Random	432.75	0.991

The final shared reward aligns agents around throughput while penalizing drops, persistent low-CW operation, and throughput disparity:

$$R^{(k)} = \sum_i t_i^{(k)} - \lambda_d \sum_i d_i^{(k)} - \lambda_{\mathrm{cw}} \sum_i \phi\left(w_i^{(k)}\right) - \lambda_f \left(1 - J\left(\mathbf{t}^{(k)}\right)\right) \sum_i t_i^{(k)}, \tag{4}$$

where $J(\mathbf{t}) = (\sum_i t_i)^2 / (N \sum_i t_i^2)$ is the Jain fairness index [14]. Policies are evaluated using the unshaped team reward (Stage 3 without the fairness penalty) so that results remain comparable to fixed-CW baselines.

We sweep $\lambda_f \in \{0, 0.25, 0.5, 1.0, 2.0\}$ at fixed $\lambda_{\mathrm{cw}} = 20$ and $\lambda_d = 1$, training for 10 PPO iterations (seed 1) and evaluating with the unshaped team reward. Fig. 6a shows a non-monotone relationship: moderate fairness pressure ($\lambda_f = 0.5$) improves both raw throughput and fairness relative to no fairness pressure ($\lambda_f = 0$), because without it the policy over-selects aggressive low-CW actions that increase collisions. Excessive pressure ($\lambda_f \geq 1$) dominates the shaped objective without improving raw fairness, because the Jain term drives down throughput faster than it resolves disparity. The coefficient $\lambda_f = 0.5$ is selected as the operating point for the multi-seed evaluation.

We train PPO with $\lambda_f = 0.5$ for 10 iterations on seeds 1, 2, and 3, then evaluate each trained policy against two fixed-CW baselines: $\mathrm{CW}_{\min} = 511$ (conservative, high-fairness) and a random policy. Table III and Fig. 6(b) report the aggregate results.

The trained policy improves raw throughput by 9.5% over the random baseline (473.91 vs. 432.75 reward/epoch) and 8.6% over the conservative fixed-CW baseline (436.60). The within-seed reward variance is below 0.1%, confirming IPC stability; the cross-seed standard deviation of 31.39 is consistent with 10-iteration PPO on a contention-sensitive reward with independent random seeds.

We report the fairness gap, where the trained policy achieves $J = 0.886 \pm 0.044$, below the fixed-CW baseline's $J = 0.994$. This gap is a property of the short training horizon (10 PPO iterations), not a fundamental limit of the reward design. The fairness-penalty sweep confirms that $\lambda_f = 0.5$ improves Jain fairness relative to no penalty; closing the gap to the fixed-CW baseline requires a longer training horizon (30–50 iterations with larger batch sizes), which we identify as the primary next step.

V. Conclusion

We have presented *ns3-marl*, an epoch-batch shared-memory protocol that replaces N per-epoch blocking synchronizations with one batch exchange. By collecting all agent data atomically within a shared epoch interval, the protocol preserves Dec-POMDP-style joint observations [11] while remaining compatible with synchronous MARL training stacks. Across five *ns*-3 scenarios, measurements show sub-linear wall-time scaling relative to the sequential baseline, Wi-Fi 6 PPO training converges, and all runs complete with zero IPC failures.

Acknowledgment

This internship was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) under the Artificial Intelligence Convergence Innovation Human Resources Development (IITP-2026-RS-2023-00255968) grant funded by the Korea government (MSIT).

References

[1] W. Wydmański and S. Szott, "Contention window optimization in IEEE 802.11ax networks with deep reinforcement learning," in *Proc. IEEE WCNC*, 2021.
[2] B. Bellalta, "IEEE 802.11ax: High-efficiency WLANs," *IEEE Wireless Communications*, vol. 23, no. 1, pp. 38–46, 2016.
[3] T. Henderson, M. Lacage, and G. Riley, "Network simulations with the ns-3 simulator," in *Proc. ACM SIGCOMM (Demo)*, 2008.
[4] ns-3 Project, "ns-3 tutorial," https://www.nsnam.org/docs/tutorial/html/index.html, 2024.
[5] E. Liang *et al.*, "RLlib: Abstractions for distributed reinforcement learning," in *Proc. ICML*, 2018, pp. 3053–3062.
[6] J. K. Terry *et al.*, "PettingZoo: Gym for multi-agent reinforcement learning," in *Proc. NeurIPS*, 2021, arXiv:2009.14471.
[7] G. Papoudakis *et al.*, "Benchmarking multi-agent deep reinforcement learning algorithms in cooperative tasks," in *Proc. NeurIPS Datasets and Benchmarks*, 2021.
[8] H. Yin *et al.*, "ns3-ai: Fostering artificial intelligence algorithms for networking research," in *Proc. WNS3*, 2020, pp. 57–64.
[9] P. Gawłowicz and A. Zubow, "ns3-gym: Extending OpenAI Gym for networking research," in *Proc. ACM MSWiM*, 2019, pp. 113–120.
[10] A.-B. Burmeister *et al.*, "DEFIANCE: Simulate reinforcement learning deployments in realistic network scenarios," in *Proc. ICNS3 2025*, 2025.
[11] F. A. Oliehoek and C. Amato, *A Concise Introduction to Decentralized POMDPs*. Springer, 2016.
[12] B. T. Doshi, "Queueing systems with vacations — a survey," *Queueing Systems*, vol. 1, no. 1, pp. 29–66, 1986.
[13] H. Takagi, *Vacation and Priority Systems, Part 1*. North-Holland, 1991.
[14] R. K. Jain, D.-M. W. Chiu, and W. R. Hawe, "A quantitative measure of fairness and discrimination for resource allocation in shared computer systems," Digital Equipment Corporation, Tech. Rep. DEC-TR-301, 1984.