Towards Efficient and Deterministic Dataflow Systems for Machine Learning

Jacky Kwok
Edward A. Lee, Ed.
Ion Stoica, Ed.

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-76
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-76.html

May 10, 2024

Acknowledgement

I wish to extend my sincere gratitude to everyone who has supported me throughout my academic journey.

Towards Efficient and Deterministic Dataflow Systems for Machine Learning

by

Jacky Kwok

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Spring 2024

The thesis of Jacky Kwok, titled Towards Efficient and Deterministic Dataflow Systems for Machine Learning, is approved:

Chair	Edward A. Lee [illegible]	Date	May 1, 2024
	Ion Stoica Ion Stoica	Date	April 30, 2024
		Date	

University of California, Berkeley

Towards Efficient and Deterministic Dataflow Systems for Machine Learning

Abstract

Towards Efficient and Deterministic Dataflow Systems for Machine Learning

by

Jacky Kwok

Master of Science in Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Edward A. Lee, Chair

Professor Ion Stoica

This thesis brings together two reports that focus on optimizing a dataflow programming language for machine learning workloads using the reactor model. The first paper introduces an efficient parallel reinforcement learning framework that outperforms existing solutions, such as Ray, in simulation throughput, multi-agent inference and training on a single node. The proposed approach achieves this by reducing the work needed for synchronization using the reactor model and decreasing the I/O overhead through optimizing the coordination of Python worker threads. This work has been accepted as a full paper at the 36th ACM Symposium on Parallelism in Algorithms and Architectures. The second paper presents a High-Performance Robotic Middleware (HPRM), which builds on top of the reactor model and employs optimizations including in-memory object store, adaptive serialization, and eager protocol with real-time sockets to ensure low-latency and deterministic communication for autonomous systems. HPRM demonstrates substantial latency reduction compared to the Robot Operating System (ROS) 2 and achieves higher throughput in CARLA autonomous driving applications. The work presented in these two papers contributes to the goal of developing high-performance and reliable systems for machine learning by leveraging the benefits of the reactor model and optimized communication mechanisms.

Contents

Acknowledgments

I am grateful to everyone who has supported me throughout my academic journey. I would like to extend a special thanks to Professors Edward A. Lee and Ion Stoica, whose guidance and expertise were crucial to my research at Berkeley.

Furthermore, I wish to acknowledge and thank Dr. Marten Lohstroh for his thoughtful advice on both of my papers, Erling Rennemo Jellum for his contribution to the implementation of real-time sockets, and Shulu Li for his work on the autonomous driving benchmarks.

I am truly fortunate to have been surrounded by such a supportive network of mentors, collaborators, friends, and family.

Chapter 1

Efficient Parallel Reinforcement Learning Framework using the Reactor Model

Parallel Reinforcement Learning (RL) frameworks are essential for mapping RL workloads to multiple computational resources, allowing for faster generation of samples, estimation of values, and policy improvement. These computational paradigms require a seamless integration of training, serving, and simulation workloads. Existing frameworks, such as Ray, are not managing this orchestration efficiently, especially in RL tasks that demand intensive input/output and synchronization between actors on a single node. In this study, we have proposed a solution implementing the reactor model, which enforces a set of actors to have a fixed communication pattern. This allows the scheduler to eliminate work needed for synchronization, such as acquiring and releasing locks for each actor or sending and processing coordination-related messages. Our framework, Lingua Franca (LF), a coordination language based on the reactor model, also supports true parallelism in Python and provides a unified interface that allows users to automatically generate dataflow graphs for RL tasks. In comparison to Ray on a single-node multi-core compute platform, LF achieves 1.21x and 11.62x higher simulation throughput in OpenAI Gym and Atari environments, reduces the average training time of synchronized parallel Q-learning by 31.2%, and accelerates multi-agent RL inference by 5.12x. This report has been accepted as a full paper at the 36th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '24).

1.1 Introduction

The field of machine learning (ML) has witnessed an exponential increase in computational requirements for training models, which tend to be increasingly larger deep neural networks. This complexity has necessitated the creation of new frameworks focused on training these networks and leveraging specialized hardware to reduce training times. Examples include TensorFlow [1], MXNet [2], PyTorch [3], and Deepspeed [4]. Beyond classical supervised learning, emerging AI applications are increasingly required to operate in dynamic environments and pursue long-term goals, problems that reinforcement learning (RL) is well suited for. RL is a paradigm where agents learn to make decisions by performing actions in an environment and receiving feedback in the form of rewards. This approach has already led to significant achievements such as AlphaGo [5], and more recently, the success of ChatGPT [6]. RL applications span various domains, including traffic systems [7], UAVs [8], large language models (LLMs) [6], and dexterous manipulation [9].

Deep RL, unlike its traditional counterparts, is typically applied in continuous state space environments, increasing the complexity of the task and thus the computational burden. In fact, with many system optimization problems, the CPU is heavily utilized by deep RL training workloads [10]. However, the scalability of deep RL, particularly in learning complex state-action associations, hinges on efficiently leveraging both CPUs and GPUs [11, 12]. The combination of RL with deep neural networks necessitates a balanced approach in computational resource allocation. The processing speed, especially when updating policies involving millions of parameters, becomes a critical factor. The need to balance CPU and GPU resources, coupled with the limitations of existing frameworks, highlights the need for innovative approaches in developing efficient, scalable, and versatile systems capable of supporting the dynamic and complex nature of modern artificial intelligence and machine learning applications. This paper addresses these challenges and explores potential solutions, paving the way for more efficient and effective reinforcement learning systems.

To this end, we introduced an optimized version of Lingua Franca (LF) [13], a polyglot coordination language for concurrent and time-sensitive applications. Our optimized LF is tailored to address the unique challenges of RL applications. LF stands out in its ability to effectively handle a diverse range of workloads, from lightweight, stateless computations needed for simulation to the more intensive, long-running computations required for training. A key feature of LF is its unified interface, which is adept at representing RL tasks as dataflow graphs. This offers a visual representation of the underlying RL processes, displaying the dependencies between reactors, ports, state variables, and their data. This enhances the understanding of the system's structure, which is crucial for efficiently managing the diverse computational demands of RL applications. LF also seamlessly integrates with RL workloads, including training, simulation, and serving. On a single-node multi-core compute platform, in comparison to Ray [14], LF achieves 1.21x and 11.62x higher simulation throughput in OpenAI Gym [15] and Atari environments, reduces the average training time of synchronized parallel Q-learning by 31.2%, and accelerates multi-agent RL inference by 5.12x.

Contributions: In this paper, we demonstrate how the reactor-oriented paradigm implemented in LF optimizes the utilization of computational resources and efficiently parallelizes RL workloads. We then describe the compilation process and the mechanisms of LF runtime, illustrating how reactors exploit parallelism and differ from the actors that underpin Ray. Our investigation also reveals that multithreading offers greater advantages than multiprocessing in parallel RL on a single node. To leverage these advantages, we introduced an optimized Reactor C runtime that supports Python without the Global Interpreter Lock, enabling true parallelism and abstracting away the burden of coordinating worker threads for RL tasks. We further present an extensive evaluation on the generations of samples from Open AI Gym and Atari Environments, synchronized parallel Q-learning, and inference of multi-agent RL. Our results demonstrate that LF outperforms Ray in terms of training, inference, and simulation for RL.

1.2 Parallel Reinforcement Learning and Requirements

Reinforcement Learning algorithms aim to enhance an agent's policy performance within a given environment, often represented through a simulator. These algorithms alternate cycles of data collection with the latest policy, value estimation using the latest data (and possibly data collected from previous policies), and policy improvement. Depending on the signal provided by the reward function on the collected data and the efficiency of the algorithm itself, the reinforcement learning process can vary in the amount of data and computational resources needed to run until satisfactory performance is achieved. For most tasks of interest, these data and compute requirements are large, motivating parallel reinforcement learning frameworks that can efficiently leverage multiple computational resources.

Many aspects of the reinforcement learning problem can be parallelized [16]. Data collection can be split across multiple worker threads, learning can be split across multiple worker threads with each thread maintaining its own value function and policy parameters, updates to the neural networks can be performed in parallelized fashions using popular deep learning frameworks [1, 17, 18], and simulators can themselves be parallelized, for example on accelerated computing infrastructure [19, 20, 21, 22].

Variants in RL Algorithms

Single-Agent Training: The most fundamental scenario in RL is training a single agent, which involves repeatedly applying the steps of rollout, replay, and optimization. Synchronous algorithms like A2C [23] and PPO [24] follow these steps in sequence, whereas asynchronous variants (e.g., A3C [23], Ape-X [25], APPO [26], IMPALA [27]) overlap rollout and optimization steps to enhance data throughput.

Multi-Agent Training: Multi-agent training involves multiple agents interacting within the environment, either cooperatively or competitively. While the dataflow structure resembles that of single-agent training, complexities arise when customizing training for individual agents. For instance, if agents require optimization at different frequencies or are trained with distinct algorithms, the training dataflow must accommodate multiple iterative loops with varying parameters. Additionally, training populations of agents, such as Double Q-learning [28], has demonstrated great promise in RL for stabilizing training, improving exploration and asymptotic performance, and generating a diverse set of solutions.

Model-Based Algorithms: Model-based RL algorithms aim to learn the transition dynamics of the environment to increase training efficiency [29]. This adds a supervised training component to parallel RL, involving training one or more dynamics models with environment-generated data.

Challenges and Opportunities in Framework Optimization:

RL algorithms like Q-learning and SARSA (State-Action-Reward-State-Action) traditionally rely on sequential learning of a value table. The advent of deep neural networks has enabled these algorithms to approximate complex functions without an explicit value table. The combination of RL with deep neural networks in deep RL necessitates a balanced approach in computational resource allocation. The processing speed, especially when updating policies involving millions of parameters, becomes a critical factor. Current popular algorithms, such as Deep Q Networks (DQN) and Advantage Actor Critic (A2C), often employ a hybrid approach using both CPUs and GPUs. This setup allows for the efficient execution of different phases of the RL process, with policy evaluation typically occurring on the CPU and policy updates on the GPU. The evolution of machine learning, especially in the realm of RL, has ushered in a new era of computational requirements and challenges.

Frameworks like Ray have made strides in CPU/GPU orchestration for RL, but they face efficiency challenges due to the actor model's inherent communication overhead and synchronization demands. Ideal frameworks for RL should support heterogeneous computations, flexibility in computational models, dynamic execution, and large data handling, while integrating seamlessly with deep learning libraries and simulation frameworks.

We argue that parallel RL frameworks should better handle millions of tasks per second by providing deterministic concurrency. Current frameworks, on the other hand, fall short in efficiently meeting the evolving demands of AI applications, indicating a significant gap in the field. This research aims to propose a novel system architecture that addresses these comprehensive requirements, bridging the gap in the current landscape and pushing the boundaries of what is possible in parallel RL.

1.3 Introduction to Actor and Reactor Model

Actor Model

The actor model [30], a foundational concept in concurrent computing, emerged in the 1970s as a response to the increasing complexity and interactivity in computer systems. It introduced a novel way to handle concurrent operations by conceptualizing "actors" as the primary units of computation. These actors are analogous to objects in object-oriented programming (OOP) [31], but they are designed specifically to address the challenges of concurrency. Each actor represents a self-contained unit with its own local state, and actors interact with each other exclusively through asynchronous message passing. This model contrasts with traditional approaches that often rely on shared state and synchronization mechanisms like locks, which can lead to issues like deadlocks and race conditions.

Actors in the model operate independently and concurrently, providing a natural framework for distributed and parallel systems. When an actor receives a message, it can perform several actions: it can create more actors, send messages to other actors, modify its own internal state, or decide how to respond to the next message it receives. This makes the model highly adaptable and scalable, suitable for applications ranging from simple concurrent programs to complex distributed systems. The asynchronous nature of message passing in the actor model is key to its effectiveness in dealing with concurrency. It allows actors to send and receive messages without waiting for a response, thereby preventing bottlenecks and enabling continuous operation even when certain components are busy or delayed.

Furthermore, the actor model introduces a flexible approach to message processing, without enforcing any strict order in which messages must be processed. This characteristic is particularly beneficial in systems where message delivery times can vary unpredictably. Actors can process incoming messages in different sequences, and the model does not guarantee that messages will arrive in the order they were sent. This flexibility allows for more efficient utilization of resources and can lead to more robust system designs that are tolerant of delays and variable message delivery times. Overall, the actor model's emphasis on independent, concurrent actors and asynchronous communication makes it a powerful paradigm for building scalable and resilient distributed or concurrent systems.

Reactor Model

The reactor model [32] shows great potential as an alternative to the actor model, enabling efficient and deterministic concurrency. The newly proposed reactor model represents an advancement in deterministic reactive systems, providing a structured framework for creating complex, reactive RL applications. Central to this model are the concepts of "reactors" and "reactions." Reactors can be interpreted as deterministic actors, but instead of responding to messages, they react to discrete events, each linked to a specific logical time, denoted by a "tag." These events can trigger reactions within a reactor, similar to message handlers in actor systems, but with a key difference: reactions in reactors are governed by a defined order, ensuring determinism. Reactions are activated by discrete events, which can also be generated by reactions. Each event associates a value with a tag, representing its logical release time within the system. Reactions can access and modify state shared with other reactions in the same reactor, but interaction between different reactors is solely through events. This design choice ensures the model's deterministic nature because the order of reaction executions is predictable and subject to strict constraints, such as tag order and execution order for reactions within the same reactor.

The reactor model can be thought of as a "sparse synchronous model" [33]. This means that synchronous-reactive interactions at a particular logical time may be confined to isolated parts of the system. When a reaction executes, it exclusively accesses the reactor's state. Moreover, for reactions in the same reactor that are triggered at the same tag, their execution order is predefined, enforcing deterministic behavior. Reactors are composed of ports (inputs and outputs), hierarchy, local state, and actions. The term "reactors" not only relate to actors in actor systems but also aligns with the synchronous reactive programming paradigm, prominent in languages like Esterel, Signal, and Lustre [34]. Unlike traditional actors, reactors don't directly reference their peers. Instead, they use named and typed ports for interconnection, enabling a hierarchical design. This hierarchy, besides facilitating deterministic behavior, also serves as a scoping mechanism for ports and imposes constraints on connection types.

Additionally, reactors feature actions, a variant of ports used for scheduling future events within the same reactor or as a synchronization mechanism between internal logic and asynchronous external events. This design choice provides a bridge between deterministic internal logic and the nondeterministic external world, like sensor data, environment states, or network messages. The reactor model also incorporates state variables. The shared resources, like replay buffers, model parameters, environment states in RL, are key motivators for grouping reactions in a single reactor. However, reactors themselves do not share state, ensuring isolation between reactors and allowing parallel execution of reactions in different reactors, unless a connection necessitates sequential execution.

Connections between reactors establish explicit communication channels. These connections reveal dependencies that are crucial for scheduling decisions honoring data dependencies. The reactor model simplifies the declaration of these dependencies by breaking down functionality into reactions with well-defined lexical scopes, thus eliminating dependencies out of scope. The reactor model's execution is governed by a run-time environment. This environment is responsible for maintaining a global event queue and a reaction queue, managing logical time, and executing reactions. Details regarding the scheduler are described below.

Ray

Several new distributed RL frameworks have recently emerged, such as Acme [35], MSRL [36], and Menger [37]. However, these frameworks primarily focus on distributing workloads across multiple nodes rather than optimizing and parallelizing RL tasks on a single-node computing platform. In contrast, Ray [38, 14] is an open-source framework designed for distributed computing and has been widely used to parallelize RL workloads from single-node to multi-nodes. Ray utilizes the actor model, which is central to its architecture and operation, making it highly effective for concurrent and distributed computing tasks. The actor model in Ray is used to encapsulate state and behavior, with actors being distributed across a node or a cluster and communicating through asynchronous message passing. It is important to highlight that in this study, our primary focus is on enhancing the efficiency of distributing actors within a node. Actors can be used for performance reasons (like caching soft state or ML models), or they could be used for managing long-living connections to databases or to web sockets. They can maintain state across multiple tasks, which is particularly useful for applications that require managing large, mutable states, such as machine learning models or large datasets.

The local scheduler in Ray is a component of a worker node called Raylet. Raylet manages the worker processes and consists of two components, a task scheduler and an object store. The task scheduler takes care of scheduling and executing work on a node. It addresses issues such as a worker being busy, not having the proper resources to run a task, or not having the values it needs to run a given task. Expensive serialization and deserialization as well as data copying are common performance bottleneck. Shared memory, specifically that which is managed directly by the operating system kernel, emerges as a superior mechanism compared to conventional approaches like socket connections. The fundamental attribute of the in-memory object store is that all objects within the store are immutable and retained in shared memory. The object store has its eviction policy, removing objects from the store or transferring them to other nodes when the allocated size limit is exceeded. This design choice ensures optimal access speeds, particularly when multiple workers on a singular node need to engage with the data. Each node provisioned by Ray is equipped with an object store, within that node's Raylet. Functionally, the object store takes care of memory management and ultimately makes sure workers have access to the objects they need in Ray.

Lingua Franca (LF) [39, 40] is a polyglot coordination language designed to facilitate the development of concurrent systems by focusing on deterministic interactions with the environments. It operates on the reactor model, where reactors are the fundamental units of composition, each encapsulating reactions to external stimuli. LF's main feature is its deterministic nature, meaning that given a set of inputs, a LF program will always produce the same outputs, greatly enhancing testability and efficiency. This determinism is achieved by using a superdense model of time (where events are ordered by time and microstep), ensuring causality and the absence of non-deterministic feedback loops in the reaction network.

The architecture of LF includes a compiler (lfc). The compiler process involves parsing and validating the LF code, checking for syntax errors, instantiation cycles, and cyclic dependencies among reactions. Valid code can then be transpiled into target code in languages like C, C++, TypeScript, and Python, which is then combined with a runtime system to manage the execution of reactors. LF also allows for graphical program representation, enhancing program structure understanding and error identification.

In LF, reactors can be defined with parameters (immutable after initialization), ports (for

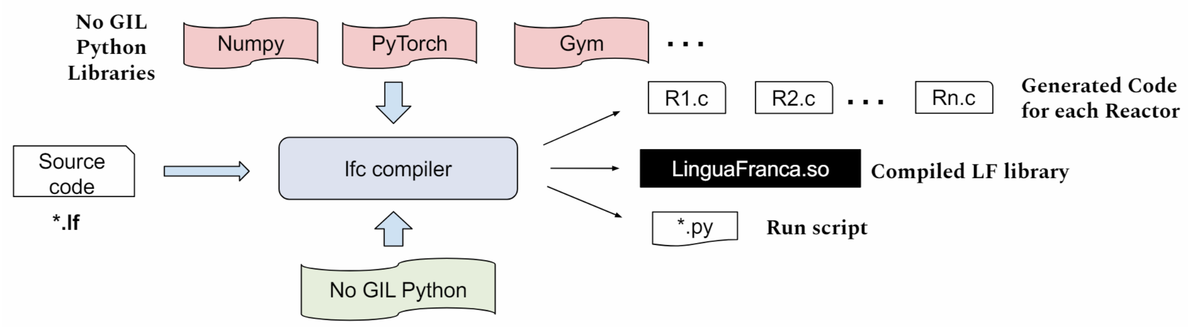

Figure 1.1: LF: compilation process

data input/output), actions (for scheduling internal events), and timers (for periodic events). Reactors can also declare state variables to maintain state across logical time. Reactions in LF, which contain the core logic, are defined with triggers (conditions under which they execute), sources (additional data inputs), and effects (outputs or actions they can trigger). LF supports reaction deadlines, where an alternative code block executes if a reaction misses its specified deadline, thus enforcing timing constraints.

LF allows for flexibility in connecting reactors through multiports (ports handling multiple data channels) and banks (multiple reactor instances). Connections can be logical, implying synchronization between ports, or physical, introducing intentional nondeterminism for scenarios where strict event ordering is not required. LF's syntax permits the creation of complex interaction patterns among reactors, offering a robust toolset for constructing scalable and maintainable concurrent systems.

Lingua Franca

The formal semantics of LF is grounded in the theory of discrete-event systems, utilizing a generalized ultrametric space for modeling the behavior of LF programs [41]. This approach guarantees that each LF program is deterministic and adheres to causality, essential for reliability in concurrent system design. LF's semantic model is fully abstract, providing both an operational perspective (how the program executes) and a denotational perspective (the meaning of program constructs), ensuring consistency. An operational semantics of the reactor model based on a formalization in LEAN [42] is also available.

Fig. 1.1 depicts the compilation process of the LF framework. This process begins with the source code written in Lingua Franca, with a .lf extension. The source code is then processed by a modified lfc compiler, tailored to work without Python's Global Interpreter Lock (GIL). This study uses a fork of Python 3.9.10 without the GIL. Additionally, the compilation process involves various no-GIL versions of the Python libraries such as NumPy, PyTorch, Gym, etc.

This compiler generates C code for each reactor, with files named R1.c, R2.c, and so on

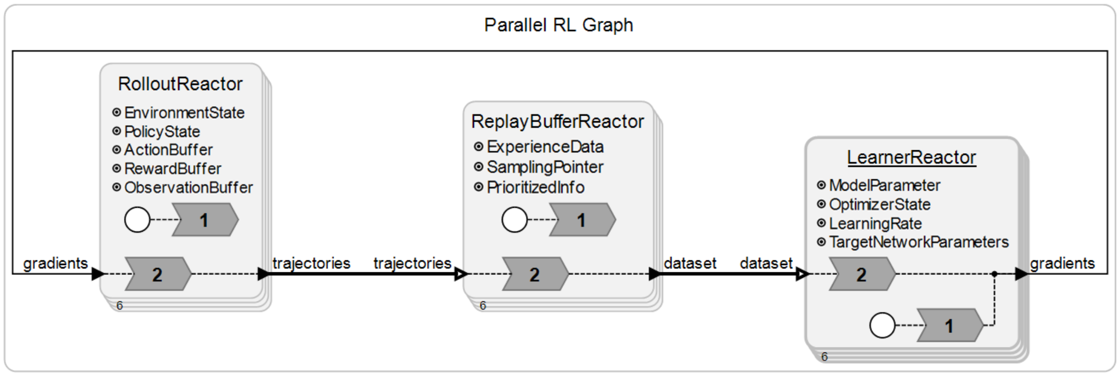

Figure 1.2: Generated Dataflow Graph for parallel RL tasks

as well as the compiled Lingua Franca library, LinguaFranca.so, which can be imported and used in Python scripts (*.py). The runtime implementation serves as a bridge between the high-level coordination language and the underlying Python environment without the GIL, enabling users to write truly concurrent Python programs. It abstracts away the burden of coordinating worker threads.

Representing RL tasks as a Dataflow Graph

The diagram synthesizer in LF provides a streamlined process to represent parallel RL tasks as dataflow graphs [13]. By simply initializing reactors and setting their input and output ports, LF can automatically generate the corresponding dataflow graph. This diagrammatic feature is seamlessly integrated with Visual Studio Code and Eclipse, offering a visual representation of the underlying RL processes. The resulting diagram (Fig. 1.2) clearly displays the reactors, ports, state variables, and their data dependencies, simplifying the understanding of the system's structure and aiding in the debugging of parallel RL tasks. Users are therefore better positioned to manage the parallelization of RL processes. LF also offers a compact syntax for ports that are capable of sending or receiving across various channels, and a syntax for multiple instances of a reactor. These concepts are known as multiports and banks of reactors. For example, we have created a bank of six instances of ReplayBufferReactor and one instance of LearnerReactor, connecting them using multiport. Detailed information about the implementation is provided in the appendix.

RolloutReactor: This reactor is pivotal in interacting with the environment. It gathers trajectories by executing a policy and recording the resulting states, rewards, and other pertinent outcomes. It receives gradients from the LearnerReactor for policy updates and sends experience (e.g. trajectories) to the ReplayBufferReactor.

- *EnvironmentState*: The present state of the environment.
- *PolicyState*: Stores the parameters of a neural network used as the policy.

- *ActionBuffer*: A temporary repository for actions decided by the policy.
- *RewardBuffer*: Temporary repository for rewards after actions.
- *ObservationBuffer*: Temporary repository for new observations post-action.

ReplayBufferReactor: Acting as a centralized experience replay buffer, this reactor stores trajectories from the RolloutReactor for subsequent sampling. It provides batched experience to the LearnerReactor for policy updates.

- *ExperienceData*: An accumulation of experiences (state, action, reward, subsequent state, and termination info).
- *SamplingPointer*: Indices or pointers facilitating efficient sampling.
- *PrioritizedInfo*: For prioritized replay buffers, additional information will be used to manage sampling.

LearnerReactor: The LearnerReactor updates policies based on sampled experiences from the ReplayBufferReactor or directly from the RolloutReactor, and broadcasts updated gradients to the RolloutReactor.

- *ModelParameter*: The parameters of neural networks.
- *OptimizerState*: Elements related to the optimization process, such as momentum variables and correction terms.
- *LearningRate*: The current learning rate, either static or dynamically adjusted.
- *TargetNetworkParameters*: In algorithms like DQN [43], these are slowly updated parameters offering a stable learning target.

1.4 Optimizations for Parallel Reinforcement Learning

Scheduling Algorithm

The execution process of programs in LF involves a scheduler responsible for overseeing all scheduled future events, managing the logical time progression, and executing triggered reactions in the order dictated by the dependency graph. The scheduling mechanism in LF, depicted in Fig. 1.3, operates with an event queue that strictly follows a tag order for processing upcoming events. Upon adding an event to the queue and processing it, the scheduler identifies triggered reactions, placing them in a reaction queue. These reactions are subsequently moved to a ready queue and executed by worker threads once all dependencies, as

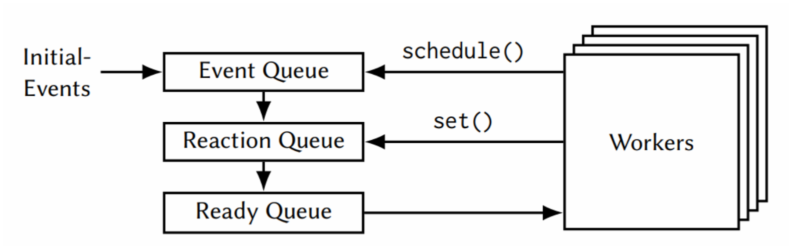

Figure 1.3: Scheduling mechanism in the LF runtime [39]

outlined by the Action-Port Graph (APG), are satisfied. Within this scheduling framework, the scheduler ensures a sequential execution of reactions that depend on one another, aiming to maximize parallelism.

The scheduling mechanism is similar to Directed Acyclic Graph (DAG)-based strategies but differs by accommodating reactions within the Action-Port Graph (APG) that may not always need execution. Since the scheduler cannot predict in advance which reactions will be triggered at a given tag, it cannot precompute an optimal schedule. To address this, the scheduler assigns a level to each reaction, allowing reactions at the same level to execute concurrently while still maintaining dependencies between different levels. This reduces synchronization overhead by eliminating the need for runtime graph analysis to determine dependencies. The scheduler processes reactions sequentially, advancing to the next level only after completing all reactions at the current level. This approach substantially reduces synchronization overhead and contention on shared resources, contributing to its efficiency.

Scheduling Optimizations

The scheduling algorithm described in the above section is relatively straightforward to implement, but achieving competitive performance requires additional optimizations. These optimizations, implemented in the C runtime in LF, are detailed below.

Coordination of worker threads: Conceptually, the scheduler and workers are distinct; however, in practice, having a central scheduler with separate worker threads can cause significant synchronization overhead. To mitigate this, our implementation allows any worker thread to become the scheduler, allowing it to move ready reactions to the queue or advance the logical time once all reactions are processed. Additionally, LF leverages the fact that we know the number of parallel reactions (reactions at the same level) to execute from the APG, and thus can use a counting semaphore to regulate the number of active workers.

Lock-free data structure: The three queues (event, reaction, and ready queue) and other data structures are shared across all workers. Mutex-based synchronization would be inefficient due to high contention over these shared resources. Accordingly, LF adopts lock-free data structures wherever feasible. For example, the ready queue is a fixed-size buffer with an atomic counter, which corresponds to the maximum number of parallel reactions defined by the APG level. Each time a worker tries to execute a reaction it atomically decrements the counter. If the counter is negative, it indicates an empty queue, and the worker should proceed accordingly. Otherwise, the counter provides the index within the buffer from which to read. This operation is safe without additional synchronization, as all workers are waiting for upcoming new reactions.

Enabling Multi-threading for Reinforcement Learning

Parallel processing is a cornerstone in the field of machine learning, enabling the handling of computationally intensive tasks and large-scale data. The parallel execution models, namely multithreading and multiprocessing, provide different advantages. This section aims to dissect these models to guide the selection of an appropriate parallel execution strategy for RL.

Multithreading in RL:

1. Shared Memory Space: Multithreading involves threads operating within the same memory space, which facilitates faster and more efficient data sharing among threads compared to multiprocessing.

2. Resource Efficiency: The creation and management of threads consume fewer resources than processes. This efficiency stems from the shared memory space and the absence of a need for complex inter-process communication mechanisms.

3. I/O Bound Task Optimization: Multithreading proves advantageous for I/O-bound operations, where the ability to perform other tasks while waiting for I/O operations enhances efficiency.

4. Context Switching: The shared process and memory space of threads enable faster context switching than multiprocessing, as less information needs to be saved and restored.

Multiprocessing in RL

1. CPU Bound Task Optimization: Multiprocessing is typically more suitable for CPU-bound tasks. However, most of the CPU-bound tasks (e.g. gradient updates) in RL should be offloaded to GPU instead.

2. Fault Tolerance and Stability: The isolated nature of processes in multiprocessing ensures that a crash in one process does not impact others, thus enhancing application stability. The scheduler in LF guarantees the safety of threads and permits users to specify actions in the event of a failure.

3. Bypassing the GIL in Python: In Python, the Global Interpreter Lock (GIL) limits thread execution, multiprocessing provides a viable alternative for parallel CPU computations. This has been resolved by leveraging the No GIL version of Python.

Optimizing Thread Allocations

The principle that running a task with multiple threads can speed up the process is generally true, as each thread can handle a portion of the work simultaneously. However, the scaling is not always linear due to various factors like thread allocation and CPU architecture. In our benchmarks, the Intel CPU has a hybrid architecture, featuring a combination of performance and efficiency cores. Performance cores are designed for high-speed and intensive tasks, while efficiency cores are optimized for lower power consumption and handling background tasks. The original LF runtime randomly allocates threads for workers. As a result, when synchronous tasks are distributed across these cores, the overall speed is bounded by the slower efficiency cores. To achieve linear scaling, the Reactor C runtime has been optimized to prioritize using the performance cores before spawning threads.

1.5 Performance Comparison Between LF and Ray

To demonstrate the improved performance of LF over Ray for parallel RL workloads, we run the following experiments. We believe these experiments are comprehensive and adequate to support our assertion that our optimized implementation of the reactor model for RL significantly advances the state of the art for parallel RL. The benchmarks can be accessed through the following GitHub repository:`https://github.com/jackyk02/parallel_rl_benchmarks`.

1. We demonstrate lower overhead of broadcast and gather operations with LF than with Ray, and show that the overhead difference scales favorably for LF as we increase both the number of actors/reactors and the communication object size. While not an RL benchmark, this toy workload resembles the computation done in RL, and is useful for highlighting why exactly it is that LF outperforms Ray.

2. In most parallel RL settings, (e.g., as is typically the case with policy gradient and Q-learning algorithms), the majority of the parallelism that can be obtained by porting these algorithms to leverage parallel compute is in the data collection phase of the algorithm (as opposed to for example value estimation or policy improvement). As such, we extensively test parallel data collection in popular RL simulated environments with both LF and Ray.

3. To demonstrate that LF integrates seamlessly with other forms of parallel compute paradigms, namely GPU acceleration, and to demonstrate the versatility LF provides to implement various reinforcement learning algorithms, we implement and evaluate synchronized parallel Q-learning with deep neural networks.

4. Finally, we observe that multi-agent RL lends itself very well to being parallelized, as each agent can maintain its own learning and data collection actors. We evaluate in such a multi-agent RL (MARL) setting and observe favorable results for LF.

Experimental Setup

The actor model is a popular approach for developing parallel RL applications, with Ray being notable for its usability and efficiency in handling multiple actors. However, LF introduces a different model of computation, imposing more restrictions compared to the actor model. We hypothesize that the reactor model with LF can outperform Ray, which is very widely used today for parallel RL. Drawing evidence from this study [39], the reactor model has been shown to surpass the performance of the traditional Actor Framework, Akka and C++ Actor Framework, by factors of 1.86x and 1.42x, respectively, for non-RL workloads. More specifically, the reactor model with a fixed set of actors and fixed communication patterns allows the scheduler to eliminate work needed for synchronization. Furthermore, with our customized implementation, LF leverages No GIL Python for multithreading, which offers several benefits, including efficient data sharing within a shared memory space and faster context switching. Previous studies[44] have demonstrated that using the No GIL version results in a speed-up of 10.9% across 58 tests in the pyperformance benchmark suite. The reduced overhead is also demonstrated by both Fig. 1.4 and Fig. 1.5 . The term, overhead, refers to the duration required for one actor to send a payload to another and for it to be received. This process may include serialization and deserialization, acquiring and releasing locks for each actor, sending and processing coordination-related messages, as well as transferring data over the network. All measurements were performed on AWS EC2 m5.8xlarge instance, equipped with an Intel® Xeon® Platinum 8175M CPU @ 2.50GHz featuring 32 vCPUs. This setup includes 128 GiB of RAM and offers a 10 Gbps network bandwidth. The system runs on Ubuntu 20.04 and use Python 3.9.10, with NumPy version 1.22.3 and gym version 0.19.0.

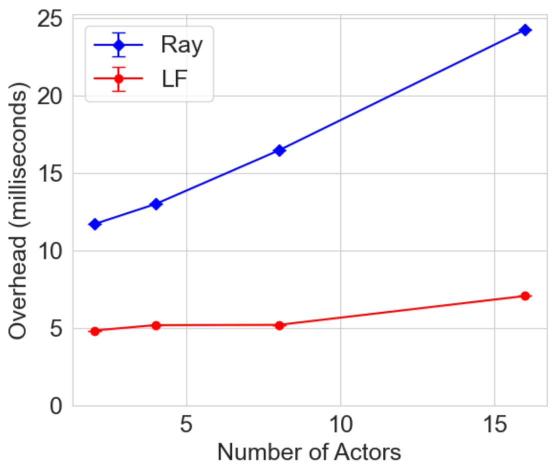

Figure 1.4: Mean Overhead of Broadcast and Gather 10MB Object with Different Number of Actors using Ray and LF.

Number of Actors

Fig. 1.4 illustrates the mean overhead of broadcasting and gathering a 10MB object across different numbers of actors in a parameter server setup. Two frameworks are compared: Ray and LF. The x-axis represents the number of actors, which are 2, 4, 8, and 16, while the y-axis shows the overhead in milliseconds. Both frameworks exhibit an increase in overhead as the number of actors grows, but Ray consistently has a higher overhead than LF. For instance, with 16 actors, Ray's overhead is close to 20 milliseconds, whereas LF's is just above 5 milliseconds.

Object Size

Fig. 1.5 presents a comparison of the mean overhead for broadcasting and gathering operations on objects of varying sizes using 16 actors, between Ray and LF frameworks, including a 99% confidence interval (CI). The x-axis displays the object size in megabytes (MB), ranging from 0 to 500 MB, while the y-axis indicates the overhead in milliseconds. From the graph, we can observe that as the object size increases, the overhead for both Ray and LF also increases. However, Ray's overhead grows at a higher rate than LF's. For instance, with the largest object size of 500 MB, Ray's overhead approaches 800 milliseconds, whereas LF's overhead is about half of that, around 400 milliseconds.

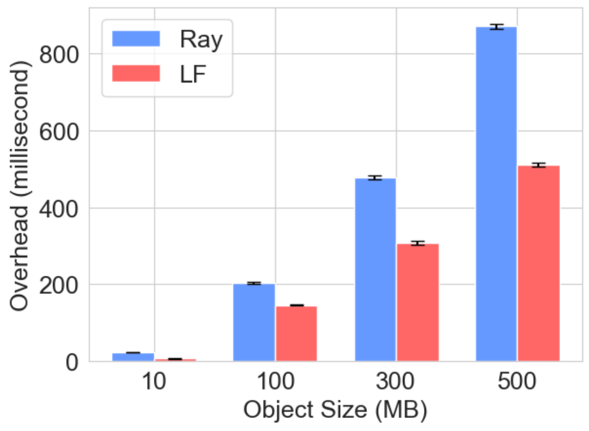

Figure 1.5: Mean Overhead of Broadcast and Gather on 16 actors with Different Object Sizes using Ray and LF.

Both Fig. 1.4 and Fig.1.5 support the hypothesis that the reactor model employed by LF can outperform traditional actor models like Ray, particularly in scenarios with a greater number of actors and larger object size, due to optimizations such as reduced synchronization work and efficient multithreading with No GIL Python.

Open AI Gym Environments

In Fig. 1.6, we see that LF outperforms Ray in terms of simulation throughput for Open AI Gym Environments by 1.21x on average, with a particularly significant lead in the Blackjack environment. This suggests that LF is more efficient, especially in situations where there is a lower CPU demand for action updates within the environment. It's important to note that vectorized environments are asynchronous and do not parallelize inference of policy. Therefore, it is not included in our benchmarks.

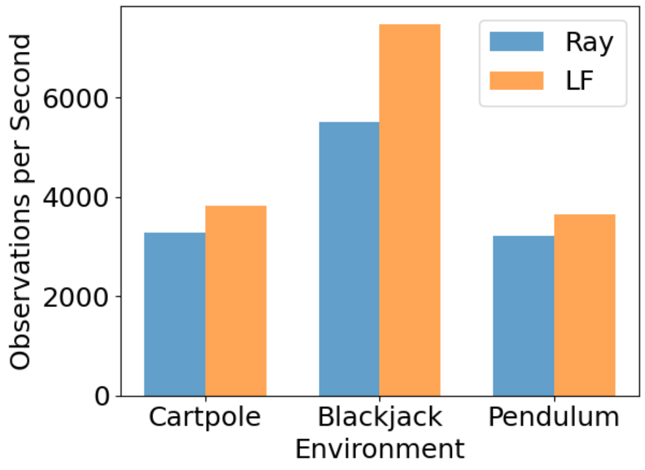

Figure 1.6: Simulation Throughput of Ray and LF with 16 actors in Various Gym Environments.

Atari Environments

In Fig. 1.7, we compare the performance of Ray and LF on Atari environments, specifically on Pacman, Pong, and SpaceInvader. Here, LF again significantly outperforms Ray in terms of observations per second. On average, LF is roughly 11.62x faster than Ray across these environments. The substantial performance difference can be attributed to LF's efficiency in handling high I/O (input/output) bound tasks. Atari environments are more complex than the previously mentioned OpenAI gym environments. They represent each state as an 80x80 numpy array, which requires more computational resources to serialize and deserialize, especially when data needs to be sent over a network. Ray's use of pickle5 for serialization does help to increase throughput by efficiently serializing NumPy arrays, but it still introduces overhead during network transmission and the serialization/deserialization process. This overhead is particularly significant in environments where state updates are frequent and must be communicated quickly. LF's ability to handle the demands of complex simulation environments is thus a key advantage.

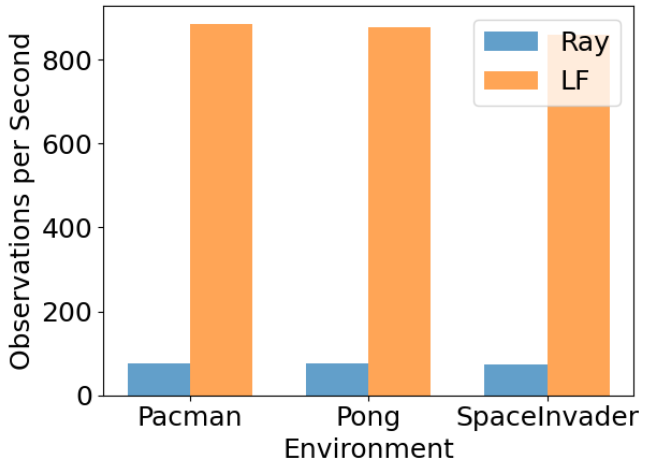

Figure 1.7: Simulation Throughput of Ray and LF with 16 actors in Various Atari Environments.

Synchronized Parallel Q-learning

Deep Q-Networks (DQNs) [43] are an advancement in reinforcement learning that utilize deep neural networks to estimate Q-values. The Q-values are predictions of the expected discounted returns after taking certain actions given particular states in an environment. DQNs extend the capabilities of traditional Q-learning by handling larger state and action spaces, which are common in complex problems. These networks can scale up effectively with more data or increased model complexity; therefore, DQN usually utilizes GPUs for gradient updates. However, since AWS EC2 m5.8xlarge does not include a GPU, benchmarking was conducted on a workstation with an Intel i9-13950HX CPU @ 2.20GHz featuring 32 vCPUs, NVIDIA RTX4090, and 32 GiB of RAM. The system also runs on Ubuntu 20.04 and uses Python 3.9.10, with numpy 1.22.3, torch 1.9.0, and gym 0.20.0.

DQN has been implemented in a synchronized parallel Q-learning setup within a BlackJack environment [43]. The dataflow graph is shown in Fig. 1.8. The network is trained to take a blackjack hand as input and output scores for each of the possible actions in the game, which represent the expected rewards of taking those actions. The computational tasks are distributed with the rollout and replay buffer being executed on a CPU, while the DQN Reactor uses a GPU. This setup takes advantage of GPUs for complex matrix operations and multicore CPUs for sequential decision-making simulations in Open AI Gym environments.

The benchmark results, as seen in Fig. 1.9, demonstrate performance improvements using LF over Ray. In tests with a 500 sample batch size from the replay buffer, the average training time decreased by 31.2%. The data also shows that while Ray's training time increases with larger mini-batch sizes, LF's performance remains stable, indicating its ability to handle larger batches efficiently.

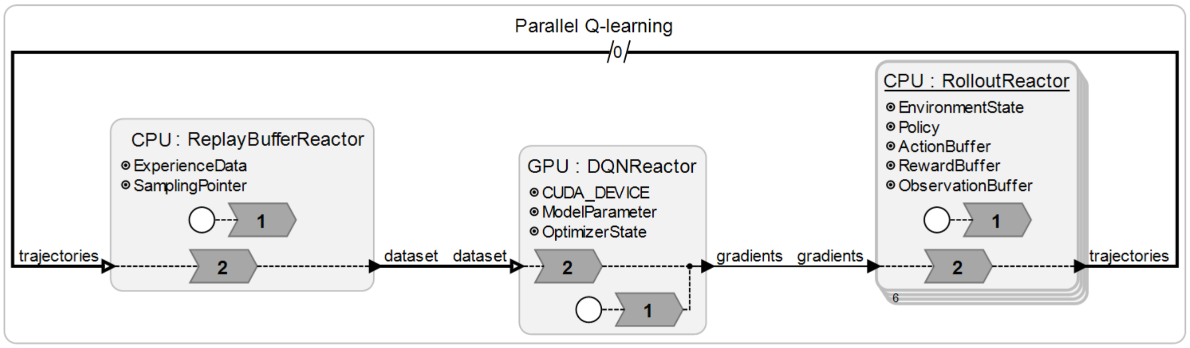

Figure 1.8: Dataflow Graph of Parallel Q-learning

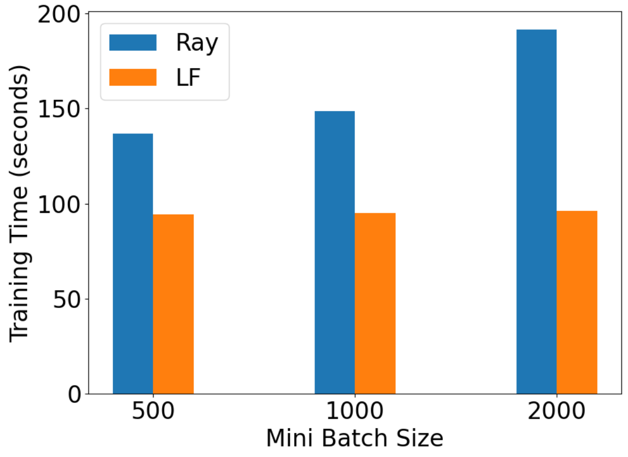

Figure 1.9: Synchronized Parallel Q-learning with Different Batch Sizes from a Replay Buffer

Multi-Agent RL Inference Comparison

In Multi-Agent RL (MARL), multiple agents operate in a common environment, where each of them tries to optimize its own return by interacting with the environment and other agents [45]. In centralized MARL, a central controller aggregates information across the agents, including joint actions, rewards, and observations, and policies across different agents. In decentralized MARL, which is more common in cooperative situations, each agent makes decisions based on its local observations [46].

We validate LF in a MARL setting with *ma-gym* [47], a MARL library based on OpenAI Gym [15]. We use the TrafficJunction4-v0 environment, in which four agents are trying to pass a crossroad without crashing. This is a decentralized setting where each agent gets its own local observation. As shown in Fig. 1.10, LF requires significantly less inference time than Ray (less than 1/2 the fitted line's slope). It is also noteworthy that as the number of agents increases as shown in Fig. 1.11, LF's inference time scales better (again, less than 1/2 the rate of increase than that of Ray). In TrafficJunction environments with 10 agents, LF achieves a 5.12x speed-up compared to Ray.

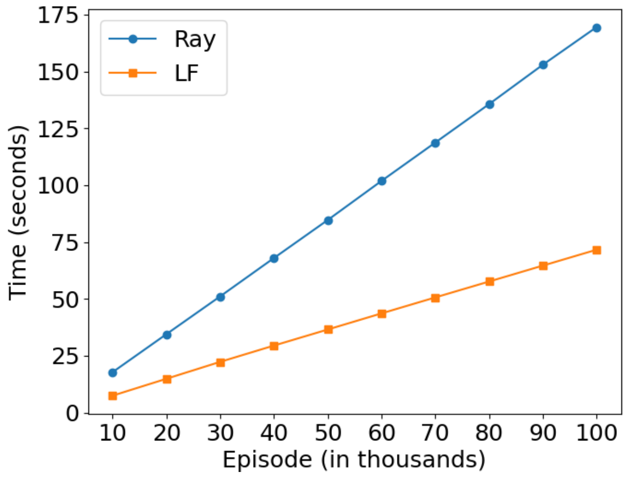

Figure 1.10: Inference Time Between Ray and LF over Episode Counts

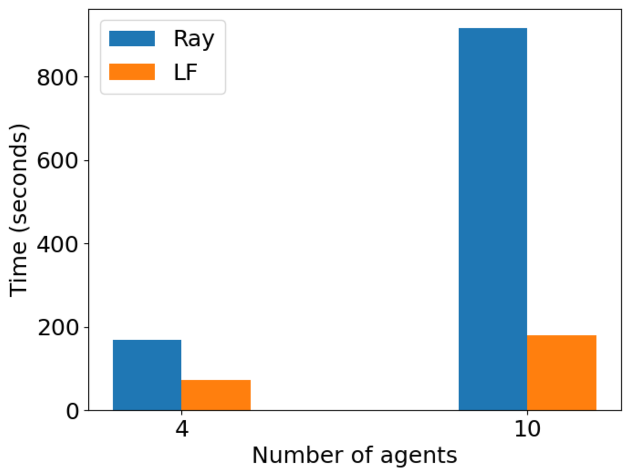

Figure 1.11: Inference Time Comparison between Ray and LF across Various Numbers of Agents

1.6 Conclusion

We demonstrate that LF outperforms the widely used framework Ray in handling training, serving, and simulation tasks in RL. We achieve this by reducing the work needed for synchronization using the reactor model and decreasing the I/O overhead through optimizing the coordination of Python worker threads. Our empirical evaluations demonstrate LF's superior performance: a 1.21x and 11.62x higher simulation throughput in OpenAI Gym and Atari environments, a 31.2% reduction in average training time for synchronized parallel Q-learning, and a 5.12x acceleration in multi-agent RL inference. We aim to incorporate the optimizations for single-node described in this work into LF's federated execution, enabling efficient distributed training and serving across nodes. We also plan to delve deeper into the potential applications of our optimizations in deploying deep RL on embedded systems, and will compare it with frameworks such as the Robot Operations System, a middleware commonly used in robotics.

1.7 Lingua Franca Code

```
target Python {
  threading: True
}

preamble {=
   #Import packages
   #e.g Torch, Gym, Numpy
=}

reactor RolloutReactor {
  input[6] gradients
  output[6] trajectories

  state EnvironmentState
  state PolicyState
  state ActionBuffer
  state RewardBuffer
  state ObservationBuffer

  reaction(startup) {=
    #Initialize Environment
  =}

  reaction(gradients) -> trajectories {=
    #Perform rollouts for the Environment
  =}
}

reactor ReplayBufferReactor {
  input[6] trajectories
  output[6] dataset

  state ExperienceData
  state SamplingPointer
  state PrioritizedInfo

  reaction(startup) {=
    #Initialize ReplayBuffer
  =}

  reaction(trajectories) -> gradients {=
    #Append Trajectories into ReplayBuffer
  =}
}

reactor LearnerReactor {
  output[6] gradients
  input[6] dataset

  state ModelParameter
  state OptimizerState
  state LearningRate
  state TargetNetworkParameters

  reaction(startup) -> gradients {=
    # Initialize the policy
  =}

  reaction(dataset) -> gradients {=
    # Update the policy
  =}
}

main reactor {
  rollout = new[6] RolloutReactor()
  replay = new[6] ReplayBufferReactor()
  learner = new[6] LearnerReactor()

  # Specifiy
  (learner.gradients)+ -> rollout.gradients
  (rollout.trajectories)+ -> replay.trajectories
  (replay.dataset)+ -> learner.dataset
}
```

Chapter 2

HPRM: High-Performance Robotic Middleware for Intelligent Autonomous Systems

The rise of intelligent autonomous systems, especially in robotics and autonomous agents, has created a critical need for robust communication middleware that can ensure real-time transmission and processing of extensive sensor data. Current robotics middleware like Robot Operating System (ROS) 2 faces challenges with nondeterminism and high communication latency when dealing with large data across multiple subscribers on a multi-core compute platform. To address these issues, we present High-Performance Robotic Middleware (HPRM), built on top of the deterministic coordination language Lingua Franca (LF). HPRM employs optimizations including an in-memory object store for efficient zero-copy transfer of large payloads, adaptive serialization to minimize serialization overhead, and an eager protocol with real-time sockets to reduce handshake latency. Benchmarks show HPRM achieves up to 173x lower latency than ROS2 when broadcasting large messages to multiple nodes. We then demonstrate the benefits of HPRM by integrating it with the CARLA simulator and running deep reinforcement learning agents along with object detection workloads. In the CARLA autonomous driving application, HPRM attains 4.79x higher throughput than ROS2. The deterministic coordination semantics of HPRM, combined with its optimized IPC mechanisms, enable efficient and predictable real-time communication for intelligent autonomous systems. This work is currently under review for publication.

2.1 Introduction

Due to the advancements in AI, the area of intelligent autonomous systems is rapidly growing. These systems, especially in the context of robotics and autonomous agents, are critical in both performance and reliability due to their capability to analyze extensive sensor data in real-time. They require a robust communication infrastructure to ensure real time transmission and processing of data.

In the architecture of autonomous systems, modules are typically organized as coarse-grained processes [48]. This design paradigm, which emphasizes functional independence and resource isolation, ensures that a failure in one module does not compromise the integrity or functionality of other modules or the system as a whole. Consequently, the exchange of data across different modules is predominantly facilitated through Inter-Process Communication (IPC) [49, 50] techniques. For instance, in a scenario where a robot is tasked with identifying specific objects for humans, the image data captured by the camera module undergoes several steps: serialization into a buffer, copying into the system kernel, transferring to the target process, and finally, deserialization. These operations usually lead to high latency in applications that make use of high-resolution cameras or LiDAR sensors.

Frameworks like the Robot Operating System (ROS) [51] and MQTT [52] have seen significant adoption in critical, concurrent, and distributed settings, including autonomous vehicles and industrial automation. These frameworks are valued for their convenience, modularity, and the use of a publish-subscribe mechanism, which can easily be leveraged for message exchange in distributed systems. However, the publish-subscribe mechanism, particularly in high-stakes environments like autonomous driving, introduces a level of non-determinism [53] due to varying communication timing, potentially resulting in unpredictable message handling sequences. This unpredictability is a significant concern in environments where the consequences of errors are severe. In addition, to support message passing in autonomous navigation systems, frameworks such as ROS2 generally utilize sockets-based communication [54]. However, this method falls short in scenarios involving the processing of large data packets across numerous subscribers, as it leads to an increase in communication latency with message size.

In this study, we offer an alternative to ROS2—High-Performance Robotic Middleware (HPRM). HPRM is an open-source robotic middleware built on top of a coordination language, Lingua Franca (LF) [13]. LF, which is based on the reactor model [32], is a polyglot coordination language that combines the most effective semantic elements from well-established computational models. This includes the actor model, Logical Execution Time (LET), synchronous reactive languages, and discrete event systems like SystemC. LF advances the field by integrating time as a primary element within its programming paradigm, thereby facilitating deterministic interactions across various physical and logical timelines. HPRM is meant to enhance the capability of robotic middleware in handling large volumes of sensor data and ML workloads using efficient IPC techniques. Specifically, it uses an in-memory object store to efficiently transfer large objects across different processes, adaptive serialization for different types of sensor data in Python, and an eager protocol and real-time sockets to minimize the handshake latency for transmitting control and object references. Our approach significantly reduces the overhead associated with local IPC compared to ROS2 [55].

We demonstrate the benefits of HPRM by integrating it with CARLA [56] to be running reinforcement learning (RL) agents and object detection in autonomous driving scenarios. HPRM is also seamlessly integrated with ROS2 and can easily install ROS2 dependencies and make use of LF's precise timing semantics. The HPRM runtime system is implemented in C and applications are modular, just like ROS and MQTT, allowing independent processes to be deployed.

2.2 Motivation and Requirements

Motivation

Frameworks like ROS2, are becoming more prevalent in critical applications, including autonomous driving, where the implications of unpredictable behaviors are significant. However, the coordination mechanism in ROS2 introduces nondeterminism [53], leading to arbitrary ordering in the handling of messages. This inherent nondeterminism in the publish-subscribe communication models poses a risk and could compromise the reliability of such systems.

Furthermore, robotics middleware [57], such as ROS2, faces considerable delays in message delivery, which can compromise the efficacy of real-time robotic operations when dealing with large volumes of data or multiple subscribers. Consider the scenario in autonomous navigation systems, where a planning module has to process large-scale inputs from perception before sending actions to other components like a localization module and vehicle control systems. For instance, the large-scale inputs may consist of high-definition video that can easily exceed 50 MB for a short clip. LiDAR sensors can generate about 10-70 MB/s, cameras can produce 10-20 MB/s depending on the resolution and frame rate. Moreover, many developers within the ROS community have experienced latency problems when publishing large data [58][59][60]. Therefore, minimizing communication delays is pivotal for improving the real-time responsiveness of robotic systems, thereby improving the overall user experience in scenarios that demand real-time data processing.

Kronaur, et al. [61] highlights the proportional increase in communication latency relative to message size using ROS2. Specifically, they observe that for messages around 4MB, the median delay for ROS2 is around 10ms. In scenarios of 1MB data being distributed to five subscribers, ROS2 exhibited a median latency nearing 80ms.

Requirements

This section outlines the key features and capabilities that HPRM must possess to address the challenges faced by current robotics middleware. These requirements focus on ensuring efficient communication, reliable performance, and ease of use for developers working on intelligent autonomous systems. The three main requirements for HPRM are:

- **Efficient Communication:** HPRM must prioritize minimal data movement and employ zero-copy mechanisms wherever possible to maintain low communication latency, even as message sizes increase. The framework should effectively handle the transmission of large payloads, significantly reducing overhead compared to current methods and improving overall system efficiency.
- **Real-time Performance and Fault Tolerance:** HPRM must guarantee real-time properties, maintaining high reliability and successful message delivery even in scenarios with heavy workloads and multiple subscribers. The framework should include mechanisms for detecting and handling faults caused by violations of timing requirements, allowing for application-specific fault handlers.
- **Ease of Use and Integration:** HPRM should abstract away the complexities of managing object read and write access, as well as the decision-making process for selecting appropriate transport mechanisms based on message size and type. The framework should be designed for seamless integration with popular robotics middleware platforms, such as ROS2, requiring minimal to no modifications for compatibility. Additionally, HPRM should enable users to easily deploy their applications across various embedded platforms without concern for platform-specific details.

2.3 ROS2 vs. Lingua Franca

ROS2

ROS2 comprises a suite of tools and libraries designed to support robotics application development. It allows developers to encapsulate software components within distinct units known as nodes, each running within its own OS process. These nodes can either reside on the same device or be spread out over several machines, communicating through the network. Regardless of their physical placement, nodes engage through a publisher-subscriber (pub-sub) system, where publishers announce topics and subscribers associate specific callback functions with those topics. In this paper, we employ ROS2, which utilizes a communication framework compliant with the Data Distribution Service (DDS) [62] to facilitate the pub-sub mechanism. The inherent pub-sub structure raises concurrency issues within the application logic, issues which are challenging to identify and resolve.

The zero-copy feature has been incorporated into both Cyclone DDS and Fast DDS. However, this feature is currently only available for rclcpp, the C++ implementation of the ROS2 client library. As of now, there is no support for the ROS2 Python client library [63].

When ROS2 nodes are running on the same hardware, a NIClevel loopback is applied without any network transmission [64, 65]. In such methods, messages are copied several times throughout processes and OS-kernel levels, leading to unnecessary memory copy and system calls. Furthermore, since a socket is a point-to-point communication interface, collective communications become inefficient. For example, if one process publishes a message to the other three processes, the entire communication stream is repeated three times.

Wang, et al. propose a hybrid solution termed Towards Zero Copy (TZC)[66], designed to optimize the handling of large messages in ROS2. In TZC, messages are separated; a lightweight descriptor traverses the conventional path over a ROS topic via TCPROS, while the main body of the message resides in shared memory. TZC utilizes a unique double reference counting system anchored in shared memory, employing a double-linked list with reference-counted nodes and leveraging Boost's shared_ptr for conventional ROS message delivery. Despite its innovation, TZC's dependency on TCP introduces sensitivity to the sequence of connection establishments, leading to inefficiencies not present in more straightforward solutions. Furthermore, TZC's lack of a robust mechanism to manage message lifecycles could potentially leave unclaimed payloads, risking memory leaks if their descriptors fail to be accurately transmitted. Furthermore, TZC is not actively developed or maintained and is not compatible with ROS2.

Lingua Franca

Lingua Franca (LF) is presented as an open-source polyglot coordination language designed to facilitate deterministic interactions among concurrent and reactive components known as reactors. The characteristic of LF that underpins our research is its deterministic nature [67]. LF orchestrates event flow through a system, where events are tagged, facilitating transmission from one reactor's port to another's. Each event is marked with a logical tag from a totally-ordered set G, ensuring every reactor processes events in a sequential tag order. Each event tag consists of a timestamp $t \in T$ indicating logical time and a microstep $m \in N$ for capturing super-dense time, allowing for precise event scheduling.

LF's design supports polyglot programming, enabling reactions within reactors to be authored in a variety of programming languages, including C, C++, Python, TypeScript, or Rust. This polyglot capability ensures that LF can be seamlessly integrated into diverse development environments by compiling LF programs into the chosen target language [39]. HPRM is developed on top of the Python target and will support C++ in the future. Currently, LF's Python and C-runtime support various embedded platforms, including Arduino, Raspberry Pi, and Zephyr RTOS.

Furthermore, for extremely latency-critical tasks, the optimal solution would be to avoid serialization through intra-process communication, allowing direct access to messages without copying or serialization. LF allows users to easily switch to intra-process communication in Python with HPRM. Kwok et al. [11] have enabled users to write truly concurrent Python programs without the limitations imposed by the Global Interpreter Lock.

An example LF program demonstrates the definition of a reactor, including its members. The connectivity is achieved through input and output ports, with the $\rightarrow$ operator establishing logical connections and an optional "after" clause for introducing time delays. Reactors maintain state variables accessible to their reactions, and they utilize timers and actions for event generation, with the ability to incorporate asynchronous external events via physical actions.

```
target Python
reactor A {
  input x
  output y
  reaction(x) -> y {=
    # ... something here ...
  =} deadline(10 msec) {=
    print("Deadline violation detected.")
  =}
}
reactor B {
  input x
  output y
  reaction(x) {=
    # ... something here ...
  =}
  reaction(startup) -> y {=
    # ... something here ...
  =}
}
main reactor {
  a = new A()
  b = new B()
  a.y -> b.x after 0
  b.y -> a.x
}
```

Listing 2.1: Lingua Franca Example Code

Time is treated as the core element in LF, with the framework providing access to both logical and physical clocks. The design principle is such that logical time closely follows physical time, maintaining a temporal coherence that ensures logical events occur near their physical counterparts but not before. Reactions can be assigned deadlines, and LF supports deadline handlers for managing situations where deadlines are violated, thereby maintaining system responsiveness and reliability. Reactor A has a specified deadline of 10 milliseconds (this value can be adjusted as a parameter of the reactor). If the reaction to event x is triggered for more than 10 milliseconds in physical time, the fault handler code will be executed in place of the first body of code.

LF employs socket-based IPC methods. However, such a socket communication mechanism is not satisfactory for the processing of large-scale sensor data or machine learning (ML) workloads, and communication latency would increase with the growth of the message size.

Our contribution through this paper is the expansion of LF's deterministic properties, enabling efficient IPC between reactors while preserving determinism.

2.4 Coordination and Optimizations

Centralized Coordination

In HPRM, we employ a centralized coordination mechanism. The Runtime Infrastructure (RTI) is employed to manage communication and synchronization among distributed components, named federates. In this strategy, the RTI is responsible for monitoring and regulating event tags during advancement of logical time, thereby assuring that federates process messages in a global consistent order. The RTI keeps track of the information below for each federate, identified as f:

- **Tag Advance Grant (TAG_f)**: The latest tag sent to federate f, enabling it to update its current event tag to $TAGf$. Initially, $TAGf$ is set to $-\infty$.
- **Logical Tag Complete (LTC_f)**: This represents the most recent tag reported by federate f, signifying the completion of all tasks (computations and communications) associated with that tag or any preceding it.
- **Next Event Tag (NET_f)**: This indicates the latest event tag from federate f, essentially the earliest future event in its queue. An empty queue is denoted by a special maximal tag, ∞. Absence of an NET message would be represented as $-\infty$.

For a federate p to advance to a logical time t in response to its upstream reaction, it must first receive authorization from the RTI. This authorization is contingent upon the RTI's assurance that p has received all messages up to and including time t.

A fundamental rule in this model is that a federate's logical time does not precede the physical time as indicated by its local physical clock.

$$\text{s.out} \rightarrow \text{p.in after 200 msec;}$$

In the connection above, a message with timestamp t from sender s cannot be sent before the local clock at s reaches t and also cannot be sent before the RTI grants to s a time advance to t. It is noted that given that s lacks upstream federates, the RTI always grants it a time advance.

If we denote the communication latency as L, the message from s to p will reach p only after physical time $t + L$ measured by s's physical clock. If there is a clock discrepancy E between s's and p's hosts, p will receive the message at physical time $t + E + L$ measured by t's physical clock. The delay parameter a (200 msec in the example) in the after clause then determines the timestamp $t + a$ for the message as received by p. At the receiving end, if $E + L > a$, then federate p will lag behind physical time by at least $E + L - a$. However, if $a > E + L$, it does not cause p's logical time to lag behind physical time. The RTI, having authorized s to move to time t, cannot permit p to advance to a time $t + a$ or beyond until it confirms the message's delivery to p. To mitigate risks associated with delays and ensure prompt processing of physical actions and meeting deadlines, it's advisable to set the after delay a on connections to federates receiving network messages to exceed any anticipated $E + L$.

The centralized coordination approach ensures the precise and timely execution of activities and events across a federated network.

Decentralized Coordination

The decentralized coordination model extends PTIDES, a real-time protocol also applied in Google Spanner, a globally distributed database. This model draws inspiration from works by Lamport, Chandy, and Misra [68, 69]. In this decentralized coordination strategy, the RTI plays a limited role, coordinating startup, shutdown, and clock synchronization. It is not involved in the execution of the distributed program.

In this approach, each federate is associated with a Safe-to-Process (STP) offset defined by the user. For a given federate f_i, we define $S_i \in T$ as its STP offset. A federate is restricted from progressing to any tag $g = (t, m)$ until the condition $Ti \geq t + S_i$ is satisfied, where T_i denotes the physical time on f_i's machine. If f_i is associated with physical actions, then $S_i \geq 0$. In other cases, S_i may assume positive, negative, or zero values. The STP offset's purpose is to ensure that all potentially influencing events from other federates, with tags preceding g, are received by f_i by the time the physical clock fulfills the aforementioned condition, thus facilitating processing in tag order.

Federates communicate directly through sockets in a peer-to-peer architecture, bypassing the RTI, and logical time advancement does not require RTI to be involved. Federates can proceed with their logical time to t once their physical clock aligns with or after $t + STP$. Similar to the after clause, if the STP offset is greater than the total of network latency, clock synchronization error, and execution times combined, then every event will be handled in the order of their tags. Since the assumptions about network latency and others can be violated, HPRM also provides a handler for STP violation.

The decentralized coordination model is designed to make software components react, even as communication latencies increase, prioritizing availability. Conversely, the centralized model ensures the predefined behavior of software components, even with delayed inputs, prioritizing consistency. This distinction highlights the decentralized model's emphasis on availability over consistency in scenarios of network degradation. Users can easily switch to decentralized coordination by specifying the target property in HPRM, which allows for flexible adaptation to varying applications.

Previous research [70] has shown that even under minimal stress on ROS2, it can observe dangerous out-of-order message sequences 0.2% of the time (600 out of 300k tests). This error rate increases by two orders of magnitude under stress. In contrast, it has been verified that using centralized coordination implementation yields zero errors over 300k test runs for this scenario. Using decentralized coordination in LF, no errors are found for realistic message publishing periods down to 1ms. Errors only began appearing for unrealistically small periods below 1ms, but unlike ROS2, these errors were detectable.

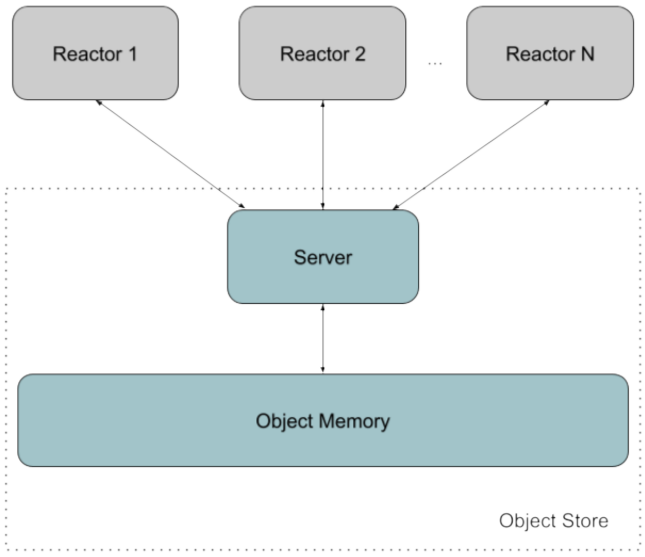

Figure 2.1: Architecture of in-memory object stores

Optimizations

It is noted that all benchmarks were conducted on a personal workstation, equipped with an Intel® i7-13620H featuring @4.90 GHz with 10 cores and a NVIDIA RTX4060. This setup includes 32 GiB of DDR4 RAM. The system runs on Ubuntu 22.04 and uses Python 3.10.12.

In-memory Object Store

The shared memory (SM) module was introduced in Python 3.8 and has been used as a workaround to enable zero-copy in ROS2. By mapping the relevant region of shared memory into each process's address space, the module allows processes to access the same data without needing to copy data into separate buffers, thus saving CPU cycles and memory bandwidth. The Python SM module is used to create a block of shared memory that can be accessed by multiple processes. Processes can share complex data types more easily by using this shared memory block. It allows for the creation, destruction, and management of shared memory segments, and it supports the creation of NumPy arrays [71] that can directly map to a shared memory block. However, this approach is inefficient compared to the in-memory object stores when transferring large objects.

In-memory object stores also enable zero-copy data transfer, reducing memory usage and improving performance. HPRM seamlessly integrates with the Plasma in-memory object store [72], automatically enabling it for the transfer of large payloads (greater than 64KB) between processes. This use of in-memory object store is inspired by Ray [14]. The architecture of Plasma object store is shown in Fig 2.1. Plasma runs as a separate process and is written in C++ and is designed as a single-threaded event loop based on the Redis event

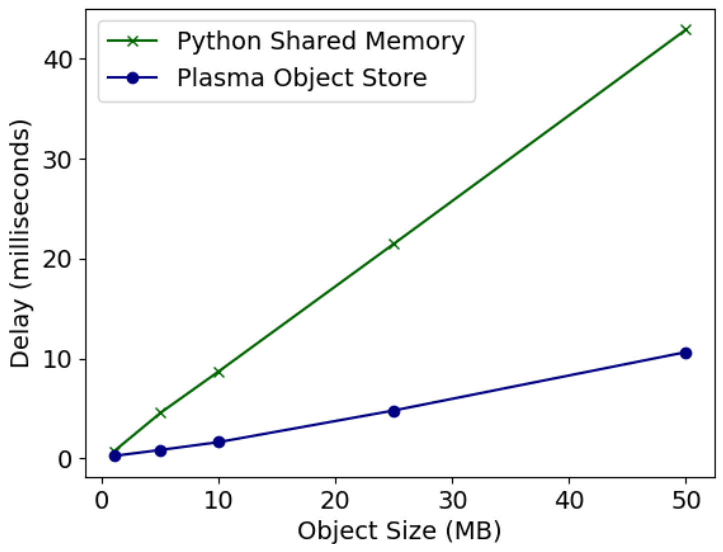

Figure 2.2: Comparison of delay in writing NumPy arrays to shared memory between Python shared memory module and Plasma object store

loop library. The plasma client library can be linked into applications. Clients communicate with the Plasma store via messages serialized using Google Flatbuffers.

Plasma outperforms Python's SM module for several reasons. Firstly, Plasma implements bulk eviction policies to manage memory more efficiently at scale. By evicting objects in bulk, the store can reduce the overhead associated with eviction, such as the cost of deciding which objects to evict and the process of eviction. Secondly, Plasma is designed to store data in a columnar format, which is optimized for efficient memory access and CPU cache utilization. This columnar format enables faster read and write operations, as well as better compression, leading to improved performance and reduced memory footprint compared to Python's SM module.

Figure 2.2 compares the delay in milliseconds when writing NumPy arrays of varying sizes to shared memory, utilizing both the Python SM module and the Plasma object store. The Python SM module's delay appears to increase linearly with the object size, growing significantly faster than the Plasma object store's delay. In contrast, the Plasma object store's delay increases at a much slower rate as the size of the object grows. At the largest object size of 50 MB, the Python SM module's delay exceeds 40 milliseconds, while the Plasma object store's delay remains just under 10 milliseconds. This demonstrates the Plasma object store's superior performance and scalability when dealing with large objects, making it a more efficient choice for applications that require high-throughput data transfer between processes.

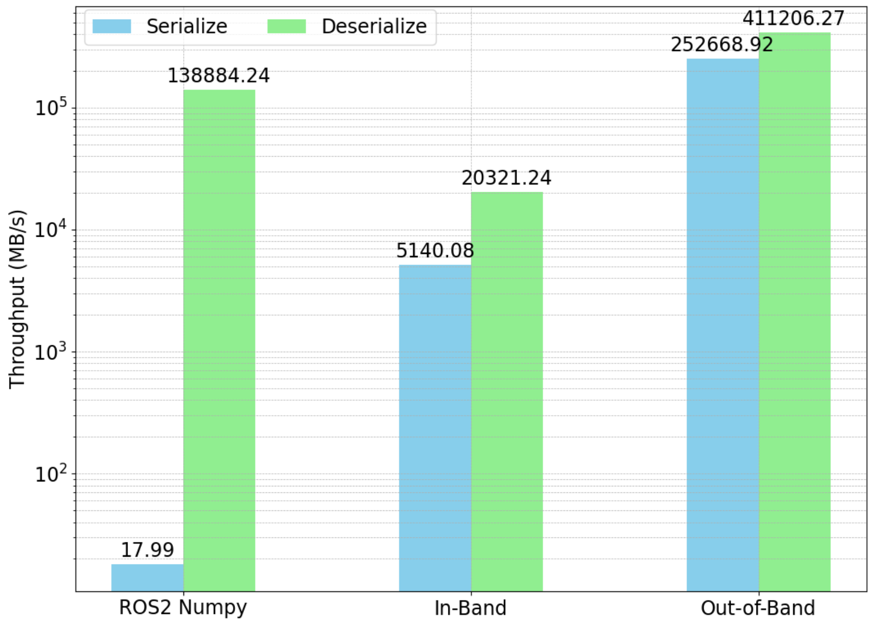

Figure 2.3: The serialization and deserialization throughput of ROS2 Numpy Package, in-band serialization, and out-of-band serialization

Adaptive Serialization

Traditional pickle serialization in Python often requires making one or more copies of the data being serialized. For example, when a large object is serialized, pickle first creates a bytes representation of the object, which is then written to the output stream. This process inherently involves copying the data. Out-of-band serialization, on the other hand, allows large data buffers to be handled separately from the main serialization stream. By using PickleBuffer objects, it's possible to avoid these additional memory copies, as the data does not need to be copied into the pickle stream but can instead be transmitted directly to the consumer in its original form.

By separating the metadata from the actual data buffers, out-of-band serialization allows the transmission of large data buffers without embedding them into the serialized pickle stream. This separation is particularly beneficial for applications that transmit large amounts of data between processes or over the network, as it enables the direct transfer of memory buffers without the overhead of serialization and deserialization processes.

Out-of-band serialization is faster than regular pickle serialization for scenarios involving large data objects because it minimizes unnecessary data copies, optimizes data transfers, and reduces the overall serialization and deserialization overhead. It is optimized for zero-copy pickling of contiguous arrays. This increases the throughput of serializing buffer-like objects, such as NumPy. The benchmark shown in Figure 2.3 illustrates the serialization and deserialization throughput for a 5 MB NumPy Array. It compares the performance between ROS2 NumPy package[73], in-band serialization and out-of-band serialization. The ROS2 NumPy package serializes NumPy arrays by embedding them into a message data type supported by ROS2, such as PointCloud or Image. The results demonstrate that, compared to the ROS2 NumPy package and in-band serialization, out-of-band serialization achieves a serialization throughput that is 14,045x and 49.2x higher, and a deserialization throughput that is 2.96x and 20.2x greater, respectively.

HPRM implements adaptive serialization, which dynamically adjusts the serialization method based on the data type. Data types such as lists, byte arrays, and integers continue to be serialized using the in-band approach, while data types like NumPy arrays and Tensors utilize out-of-band serialization for optimal performance. Additionally, we have developed a recursive serializer for data structures storing different types of objects, including dictionary and list. This serializer separates objects like NumPy arrays from other objects that can be transmitted between reactors using regular serialization. The serialized bytes are then retrieved from the in-memory object store by the receiver and combined with the object received over the network to reconstruct the data structure.

Eager Protocol & Real-Time Sockets

To minimize the latency for transmitting small payloads, such as object references, metadata, and vehicle controls, we've implemented an eager protocol [74]. It pre-allocates fixed-size buffering space (64KB) for the message, reducing the handshake latency or wait time involved for the other federate to allocate memory for a new message. Also, the Nagle algorithm [75], enabled by default, bundles short TCP messages together to avoid network traffic. As a result, it was delaying small messages. A socket option has been added to HPRM for disabling it.

2.5 Evaluation

Mean Latency

Figure 2.4 illustrates the comparison of average latency for broadcast and gather operations on objects with varying sizes using 4 nodes. This experiment utilizes ROS2 Humble, its variant with shared memory, and both the centralized and decentralized coordination strategy of HPRM. The x-axis represents the object size in megabytes (MB), ranging from 1 to 50 MB, while the y-axis indicates the latency in milliseconds, displayed on a logarithmic scale. The term latency refers to the duration required for one node to send a payload to another and for it to be received. This process may include sending and processing coordination-related messages, serialization and deserialization, and transferring data over the network.

It is important to note that ROS2 Humble (Shared Memory) refers to using Python's pickle for serialization and the shared memory module IPC. This approach allows ROS2 to pass object references between processes, a workaround commonly utilized by robotics

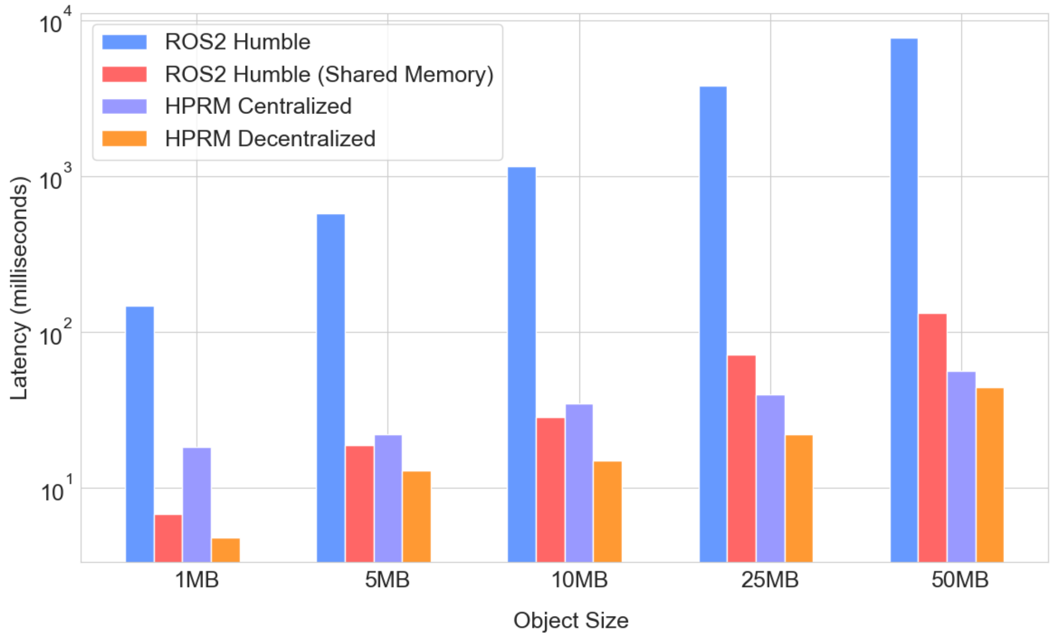

Figure 2.4: Mean latency of broadcast and gather on 4 nodes with different object sizes

developers to leverage the benefits of shared memory. However, we argue that such a method is inefficient and requires users to consider the message size and type, as well as to manage the object read and write access manually. HPRM abstracts away these burdens for the users.

From the graph, we can observe that as the object size increases, the latency for both ROS2 and HPRM also increases. However, ROS2's latency grows at a higher rate than HPRM's. Specifically, for 10MB objects, the typical size of large camera images, ROS2's mean latency hits 1,161 ms, while HPRM's mean latency is around 15 ms—about 77x faster. For the largest object size of 50 MB, ROS2's average latency is at 7,723 ms, whereas HPRM's average latency is 44.6 ms, which is 173x faster. We also observed that the latency of decentralized coordination is lower than that of centralized coordination, as it prioritizes availability and incurs less synchronization overhead. Nevertheless, as the size of the object grows, the impact of synchronization overhead on the mean latency diminishes. We conclude that HPRM with decentralized coordination consistently shows the lowest latency across all object sizes in the plot, outperforming ROS2 Humble and its variant with shared memory.

Applications

To show the improvements in performance, we validated HPRM and ROS2 Humble on running deep reinforcement learning agents in the CARLA autonomous driving simulator.

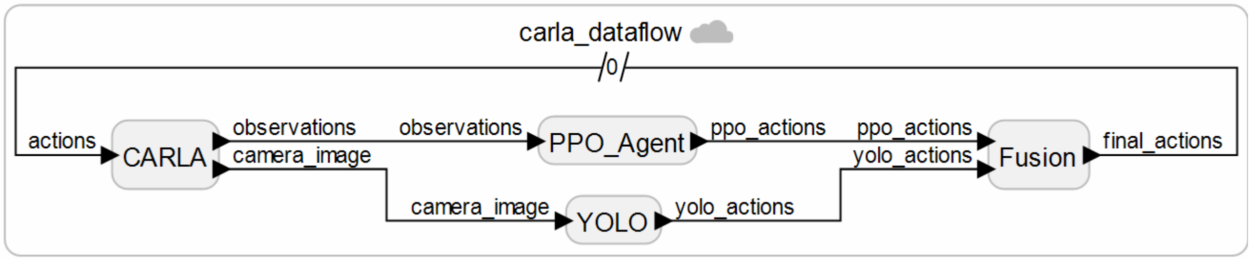

Figure 2.5: Dataflow diagram of the autonomous driving application

We designed a benchmark that simulates end-to-end urban driving, running ML models in parallel during inference. Specifically, we adapted a pre-trained Proximal Policy Optimization (PPO) agent developed by Zhang et al. [76] to run in parallel with You Only Look Once (YOLO) [77] for object detection.

The neural network architecture used by the RL expert employs six convolutional layers to encode the bird's-eye view (BEV) and two fully-connected (FC) layers to encode the measurement vector. Outputs of both encoders are concatenated and then processed by another two FC layers to produce a latent feature, which is then fed into a value head and a policy head, each with two FC hidden layers. In conjunction, we utilize YOLOv5 for object detection. The YOLOv5 architecture is structured into three main components: the backbone, neck, and head. The backbone, leverages Cross-Stage Partial networks, extracts features from input images. The neck, built upon a Path Aggregation Network, processes these features to produce enriched feature maps at various scales. Finally, the head component uses these feature maps to predict bounding boxes and class probabilities for detected objects.

Figure 2.5 is a data flow diagram automatically generated by the LF tools. The rendered BEV and RGB camera images from the CARLA simulator are passed separately to the PPO Agent reactor and the YOLO reactor. PPO Agent runs the policy and passes the policy action to Fusion reactor, while YOLO is executed in parallel and passes action based on object detection (e.g. STOP signs and traffic lights) to the Fusion reactor. The fusion reactor processes the two actions and determines the final actions. When actions are received by the CARLA reactor, the simulator applies those actions, advancing to the next frame. The publisher and subscriber implementation of the benchmark in ROS2 follows the same paradigm as in HPRM, replacing reactors with ROS nodes. For synchronization purposes, the fusion node in ROS2 blocks until it has received updated actions from both the PPO Agent and YOLO, after which it sends the final action to the CARLA node.

The throughput is measured after 100 warm-up steps, and then measured across 400 environment step frames. We found that running PPO policy inference in CPU and YOLO in GPU led to a slight performance increase due to full utilization of compute resources, and was implemented across the benchmark. The box plot in Figure 2.6 illustrates the frames per second (FPS) measured when running the CARLA benchmark of 500 environment step frames with HPRM and ROS2 Humble.

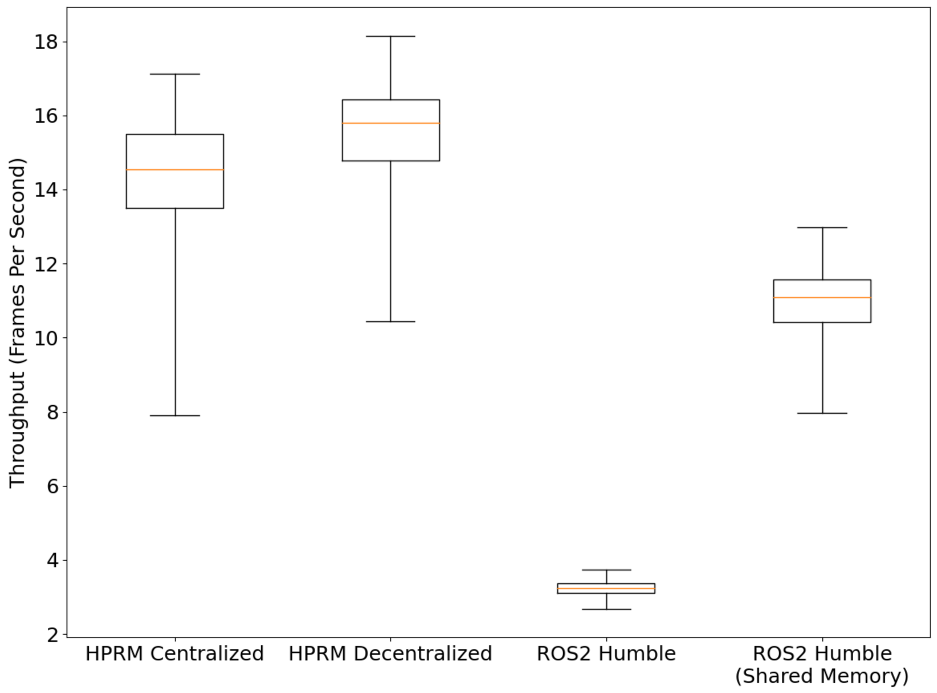

Figure 2.6: Performance of HPRM and ROS2 on the CARLA benchmark

From the box plot we can see that HPRM with decentralized and centralized coordination outperforms ROS2, even with shared memory, by 1.41x and 1.30x respectively. When comparing HPRM with the default ROS2 Humble, the throughput increased by 4.79x. HPRM with decentralized coordination has the best performance with the note that frame rates, slightly outperforming HPRM with centralized coordination. The optimizations implemented in this research significantly lowered the IPC overhead, as further demonstrated in real-world application.

It's worth noting that the performance gap between HPRM and ROS2 is more pronounced in the latency benchmark than in the real-world application. This can be attributed to the fact that our optimizations primarily focus on reducing I/O overhead. The mean latency benchmark consists of more I/O-bound tasks compared to the actual application, where a significant portion of the computation time is spent on running inference for ML models. As a result, the impact of our optimizations is more evident in the mean latency benchmark, whereas the performance difference in the real-world application is relatively smaller.

2.6 Conclusions

In this paper, we presented HPRM, a high-performance robotic middleware designed to address the challenges of nondeterminism and high communication latency in intelligent autonomous systems. Built on top of the deterministic coordination language Lingua Franca, HPRM leverages a centralized and decentralized coordination model to ensure predictable event processing across distributed nodes. We introduced several optimizations in HPRM, including an in-memory object store for efficient zero-copy transfer of large payloads, adaptive serialization to minimize serialization overhead based on data types, and an eager protocol with real-time sockets to reduce handshake latency.

Our performance evaluation demonstrated the significant benefits of HPRM compared to ROS2, a widely-used robotics middleware. Benchmark results showed that HPRM achieved up to 173x lower latency than ROS2 when transmitting large messages to multiple nodes. Furthermore, we validated the real-world applicability of HPRM by integrating it with the CARLA autonomous driving simulator and running deep reinforcement learning agents alongside object detection workloads. In this application, HPRM attained a 4.79x higher throughput than ROS2.

The deterministic coordination semantics of HPRM, combined with its optimized IPC mechanisms, enable efficient and predictable real-time communication for intelligent autonomous systems. By abstracting away the complexities of managing shared memory, object references, and messages, HPRM simplifies the development of high-performance, deterministic robotic applications.

Future work could explore the integration of HPRM with other programming languages and real-world robotic platforms to further validate its usability and performance. Additionally, investigating the scalability of HPRM in larger, more complex distributed robotic systems could provide valuable insights for further optimizations.

In conclusion, HPRM represents a significant step forward in the development of deterministic, high-performance robotic middleware. Its ability to efficiently handle large data payloads and ensure predictable event processing makes it a promising solution for the growing demands of intelligent autonomous systems.

Bibliography

[1] Martín Abadi et al. “TensorFlow: A System for Large-Scale Machine Learning”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA, USA: USENIX Association, 2016, pp. 265–283.

[2] Tianqi Chen et al. *MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems*. 2015. URL: `https://arxiv.org/abs/1512.01274`.

[3] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. URL: `https://doi.org/10.48550/arXiv.1912.01703`.

[4] Jeff Rasley et al. “Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters”. In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2020, pp. 3505–3506.

[5] David Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: *nature* 529.7587 (2016), pp. 484–489.

[6] OpenAI. *GPT-4 Technical Report*. 2023. URL: `https://arxiv.org/abs/2303.08774`.

[7] Li Chen et al. “Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization”. In: *Proceedings of the 2018 conference of the ACM special interest group on data communication*. 2018, pp. 191–205.

[8] Elia Kaufmann et al. “Champion-level drone racing using deep reinforcement learning”. In: *Nature* 620.7976 (2023), pp. 982–987.

[9] Ilge Akkaya et al. *Solving rubik’s cube with a robot hand*. 2019.

[10] Ameer Haj-Ali et al. *Deep Reinforcement Learning in System Optimization*. 2019. arXiv: `1908.01275`. URL: `http://arxiv.org/abs/1908.01275`.

[11] Ahmet Inci et al. *The architectural implications of distributed reinforcement learning on CPU-GPU systems*. 2020.

[12] Lasse Espeholt et al. *Seed rl: Scalable and efficient deep-rl with accelerated central inference*. 2019.

[13] Marten Lohstroh et al. “Toward a Lingua Franca for deterministic concurrent systems”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 20.4 (2021), pp. 1–27.

[14] Philipp Moritz et al. “Ray: A distributed framework for emerging {AI} applications”. In: *13th USENIX symposium on operating systems design and implementation (OSDI 18)*. 2018, pp. 561–577.

[15] Greg Brockman et al. *Openai gym.* 2016.

[16] Mohammad Reza Samsami and Hossein Alimadad. *Distributed deep reinforcement learning: An overview.* 2020.

[17] James Bradbury et al. *JAX: composable transformations of Python+ NumPy programs.* 2018.

[18] Adam Paszke et al. *Automatic differentiation in pytorch.* 2017.

[19] Xavi Puig et al. *Habitat 3.0: A Co-Habitat for Humans, Avatars and Robots.* 2023.

[20] Andrew Szot et al. “Habitat 2.0: Training Home Assistants to Rearrange their Habitat”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2021, pp. 1–17.

[21] Manolis Savva et al. “Habitat: A Platform for Embodied AI Research”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. 2019, pp. 1–17.

[22] Brennan Shacklett et al. “An extensible, data-oriented architecture for high-performance, many-world simulation”. In: *ACM Transactions on Graphics (TOG)* 42.4 (2023), pp. 1–13.

[23] Volodymyr Mnih et al. “Asynchronous methods for deep reinforcement learning”. In: *International conference on machine learning.* PMLR. 2016, pp. 1928–1937.

[24] John Schulman et al. *Proximal Policy Optimization Algorithms.* 2017. arXiv: `1707.06347`. URL: `http://arxiv.org/abs/1707.06347`.

[25] Dan Horgan et al. *Distributed Prioritized Experience Replay.* 2018. arXiv: `1803.00933`. URL: `http://arxiv.org/abs/1803.00933`.

[26] Fanyu Zeng and Chen Wang. “Visual navigation with asynchronous proximal policy optimization in artificial agents”. In: *Journal of Robotics* 2020 (2020), pp. 1–7.

[27] Lasse Espeholt et al. “Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures”. In: *International conference on machine learning.* PMLR. 2018, pp. 1407–1416.

[28] Hado Van Hasselt, Arthur Guez, and David Silver. *Deep Reinforcement Learning with Double Q-learning.* 2015.

[29] Thomas M Moerland et al. “Model-based reinforcement learning: A survey”. In: *Foundations and Trends® in Machine Learning* 16.1 (2023), pp. 1–118.

[30] Carl Hewitt. *Actor Model for Discretionary, Adaptive Concurrency.* 2010. arXiv: `1008.1459`. URL: `http://arxiv.org/abs/1008.1459`.

[31] Tim Rentsch. "Object oriented programming". In: *ACM Sigplan Notices* 17.9 (1982), pp. 51–57.

[32] Marten Lohstroh. "Reactors: A Deterministic Model of Concurrent Computation for Reactive Systems". PhD thesis. EECS Department, University of California, Berkeley, Dec. 2020. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-235.html`.

[33] Stephen A. Edwards and John Hui. "The Sparse Synchronous Model". In: *Forum on Specification and Design Languages (FDL)*. 2020, pp. 1–8.

[34] Albert Benveniste and Gérard Berry. "The Synchronous Approach to Reactive and Real-Time Systems". In: *Proceedings of the IEEE* 79.9 (1991), pp. 1270–1282.

[35] Matthew W Hoffman et al. *Acme: A research framework for distributed reinforcement learning.* 2020.

[36] Huanzhou Zhu et al. "MSRL: Distributed Reinforcement Learning with Dataflow Fragments". In: *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. Boston, MA: USENIX Association, July 2023, pp. 977–993. ISBN: 978-1-939133-35-9. URL: `https://www.usenix.org/conference/atc23/presentation/zhu-huanzhou`.

[37] Amir Yazdanbakhsh, Junchao Chen, and Yu Zheng. *Menger: Massively large-scale distributed reinforcement learning.* 2020.

[38] Eric Liang et al. "RLlib Flow: Distributed Reinforcement Learning is a Dataflow Problem". In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 5506–5517.

[39] Christian Menard et al. "High-performance deterministic concurrency using lingua franca". In: *ACM Transactions on Architecture and Code Optimization* 20.4 (2023), pp. 1–29.

[40] Alexander Schulz-Rosengarten et al. "Polyglot Modal Models through Lingua Franca". In: *Proceedings of Cyber-Physical Systems and Internet of Things Week 2023*. Berkeley, CA, USA: IEEE, 2023, pp. 337–342.

[41] Xiaojun Liu, Eleftherios Matsikoudis, and Edward A. Lee. "Modeling Timed Concurrent Systems". In: *CONCUR 2006 - Concurrency Theory*. Vol. LNCS 4137. Springer, 2006, pp. 1–15. DOI: `10.1007/11817949_1`.

[42] Leonardo de Moura et al. "The Lean theorem prover (system description)". In: *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*. Springer. 2015, pp. 378–388.

[43] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning.* 2013. arXiv: `1312.5602`. URL: `http://arxiv.org/abs/1312.5602`.

[44] Sam Gross. *Multithreaded Python without the GIL.* `https://docs.google.com/document/d/18CXhDb1ygxg-YXNBJNzfzZsDFosB5e6BfnXLlejd9l0/edit`. [Online; accessed 17-April-2024]. 2021.

[45] Kaiqing Zhang, Zhuoran Yang, and Tamer Başar. "Multi-agent reinforcement learning: A selective overview of theories and algorithms". In: *Handbook of reinforcement learning and control* 325.11 (2021), pp. 321–384.

[46] Afshin Oroojlooy and Davood Hajinezhad. "A review of cooperative multi-agent deep reinforcement learning". In: *Applied Intelligence* 53.11 (2023), pp. 13677–13722.

[47] Anurag Koul. *ma-gym: Collection of multi-agent environments based on OpenAI gym.* `https://github.com/koulanurag/ma-gym`. 2019.

[48] Joseph Sifakis. "Autonomous systems–an architectural characterization". In: *Models, Languages, and Tools for Concurrent and Distributed Programming: Essays Dedicated to Rocco De Nicola on the Occasion of His 65th Birthday* (2019), pp. 388–410.

[49] Hamed Dinari. "Inter-Process Communication (IPC) in Distributed Environments: An Investigation and Performance Analysis of Some Middleware Technologies." In: *International Journal of Modern Education & Computer Science* 12.2 (2020).

[50] Aditya Venkataraman and Kishore Kumar Jagadeesha. "Evaluation of inter-process communication mechanisms". In: *Architecture* 86 (2015), p. 64.

[51] Morgan Quigley et al. "ROS: an open-source Robot Operating System". In: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan. 2009, p. 5.

[52] Dipa Soni and Ashwin Makwana. "A survey on MQTT: a protocol of internet of things (IoT)". In: *International conference on telecommunication, power analysis and computing techniques (ICTPACT-2017)*. Vol. 20. 2017, pp. 173–177.

[53] Nicholas Hunt et al. "DDOS: taming nondeterminism in distributed systems". In: *ACM SIGPLAN Notices* 48.4 (2013), pp. 499–508.

[54] Tobias Blass et al. "Automatic latency management for ROS2: Benefits, challenges, and open problems". In: *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2021, pp. 264–277.

[55] Lennart Puck et al. "Performance evaluation of real-time ROS2 robotic control in a time-synchronized distributed network". In: *2021 IEEE 17th International Conference on Automation Science and Engineering (CASE)*. IEEE. 2021, pp. 1670–1676.

[56] Alexey Dosovitskiy et al. "CARLA: An open urban driving simulator". In: *Conference on robot learning*. PMLR. 2017, pp. 1–16.

[57] Paul Fitzpatrick et al. "A middle way for robotics middleware". In: *Journal of Software Engineering for Robotics* 5.2 (2014), pp. 42–49.

[58] *Extremely slow message creation for large arrays in Python.* `https://github.com/ros2/rosidl_python/issues/156`.

[59] *Publishing large data is 30x-100x slower than for rclcpp.* `https://github.com/ros2/rclpy/issues/763`.

[60] *Very slow publishing of large messages.* `https://github.com/ros2/ros2/issues/1242`.

[61] Tobias Kronauer et al. "Latency analysis of ROS2 multi-node systems". In: *2021 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI).* IEEE. 2021, pp. 1–7.

[62] Gerardo Pardo-Castellote. "Omg data-distribution service: Architectural overview". In: *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.* IEEE. 2003, pp. 200–206.

[63] *Memory leak in Subscription when using zero-copy with rmw_cyclonedds.* `https://github.com/ros2/rclpy/issues/833`.

[64] Wei Liu et al. "A robotic communication middleware combining high performance and high reliability". In: *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD).* IEEE. 2020, pp. 217–224.

[65] Wei Liu et al. "Zoro: A robotic middleware combining high performance and high reliability". In: *Journal of Parallel and Distributed Computing* 166 (2022), pp. 126–138.

[66] Yu-Ping Wang et al. "TZC: Efficient inter-process communication for robotics middleware with partial serialization". In: *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).* IEEE. 2019, pp. 7805–7812.

[67] Edward A Lee et al. "Consistency vs. availability in distributed cyber-physical systems". In: *ACM Transactions on Embedded Computing Systems* 22.5s (2023), pp. 1–24.

[68] K. Mani Chandy and Jayadev Misra. "Distributed simulation: A case study in design and verification of distributed programs". In: *IEEE Transactions on software engineering* 5 (1979), pp. 440–452.

[69] K. Mani Chandy, Victor Holmes, and Jayadev Misra. "Distributed simulation of networks". In: *Computer Networks (1976)* 3.2 (1979), pp. 105–113.

[70] Soroush Bateni et al. "Risk and Mitigation of Nondeterminism in Distributed Cyber-Physical Systems". In: *2023 21st ACM-IEEE International Symposium on Formal Methods and Models for System Design (MEMOCODE).* 2023, pp. 1–11.

[71] Travis E Oliphant et al. *Guide to numpy.* Vol. 1. Trelgol Publishing USA, 2006.

[72] Apache Arrow. *A cross-language development platform for in-memory data.* 2022.

[73] *ROS2 NumPy, Tooling for converting ROS messages to and from NumPy arrays.* `https://github.com/Box-Robotics/ros2_numpy`.

[74] Ron Brightwell and Keith Underwood. "Evaluation of an eager protocol optimization for MPI". In: *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting.* Springer. 2003, pp. 327–334.

[75] Greg Minshall et al. “Application performance pitfalls and TCP’s Nagle algorithm”. In: *ACM SIGMETRICS Performance Evaluation Review* 27.4 (2000), pp. 36–44.

[76] Zhejun Zhang et al. “End-to-End Urban Driving by Imitating a Reinforcement Learning Coach”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. 2021.

[77] Glenn Jocher. *YOLOv5 by Ultralytics*. Version 7.0. May 2020. DOI: `10.5281/zenodo.3908559`. URL: `https://github.com/ultralytics/yolov5`.