DEFIANCE Design Documentation

Release 1.0

BPHK2023

Aug 16, 2024

CONTENTS

CHAPTER

ONE

DESIGN DOCUMENTATION

1.1 Overview

1.1.1 Basic Components

The goal of this module is to support the integration of reinforcement learning (RL) components into network scenarios to simulate their deployment and the communication between them. Typical RL tasks include agents, actions, observations and rewards as their main components. In a network, these components are often placed on different nodes. For example, collecting observations and training an agent often happen at different locations in the network. To associate these RL components with `Nodes`, the abstraction of user applications is used. The following applications inherit from a general `RlApplication`:

- `ObservationApplication`: observes part of the network state and communicates the collected data (i.e. observations or data used to calculate observations) to one or more agents
- `RewardApplication`: collects data to calculate a reward and communicates it to one or more agents
- `AgentApplication`: represents the training and/or inference agent in the network.
- `ActionApplication`: executes an action that was inferred by an agent and thereby changes a part of the network state

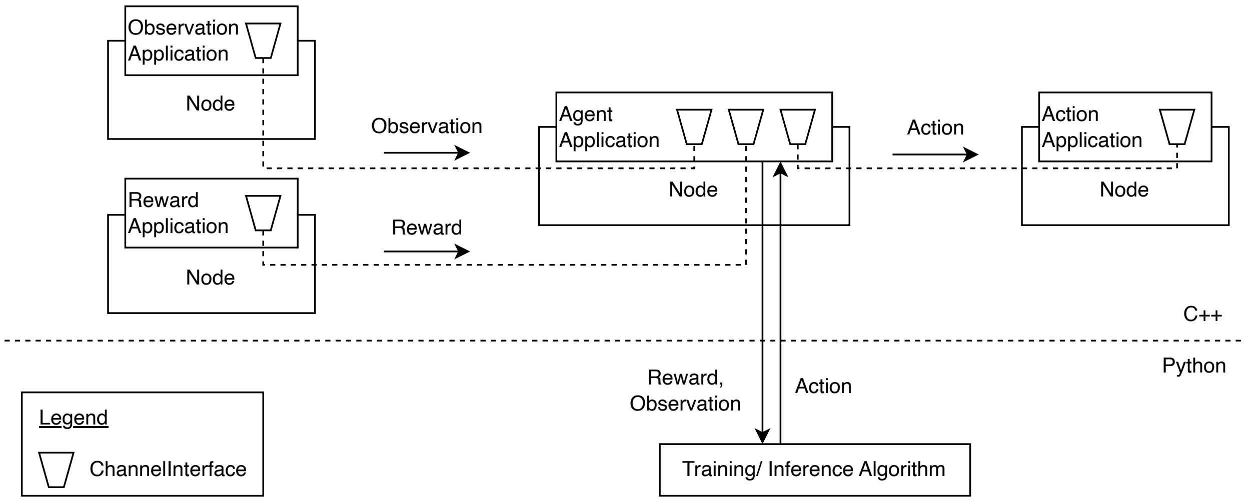


Fig. 1: Basic interaction of `RlApplications`

A commonly used standard for implementing RL environments is the Gymnasium standard [Gymnasium], which is based on Python. With RLLib (Ray) [RLLib] an extensive Python library for RL exists that uses this standard as an interface for single-agent training. As *ns-3* is implemented in C++, a connection with the mainly Python-based RL frameworks needs to be established. This module uses *ns3-ai* [ns3-ai] for the inter-process communication.

1.1.2 Design Criteria

Possible use cases this module is designed for are the following:

- Simulation of communication overhead between RL components
- Simulating how calculation and/or communication delays influence the performance of an RL approach via configurable delays
- Testing and evaluating tradeoffs between different RL deployments, e.g., distributed deployment on several nodes vs. centralized deployment on a single node

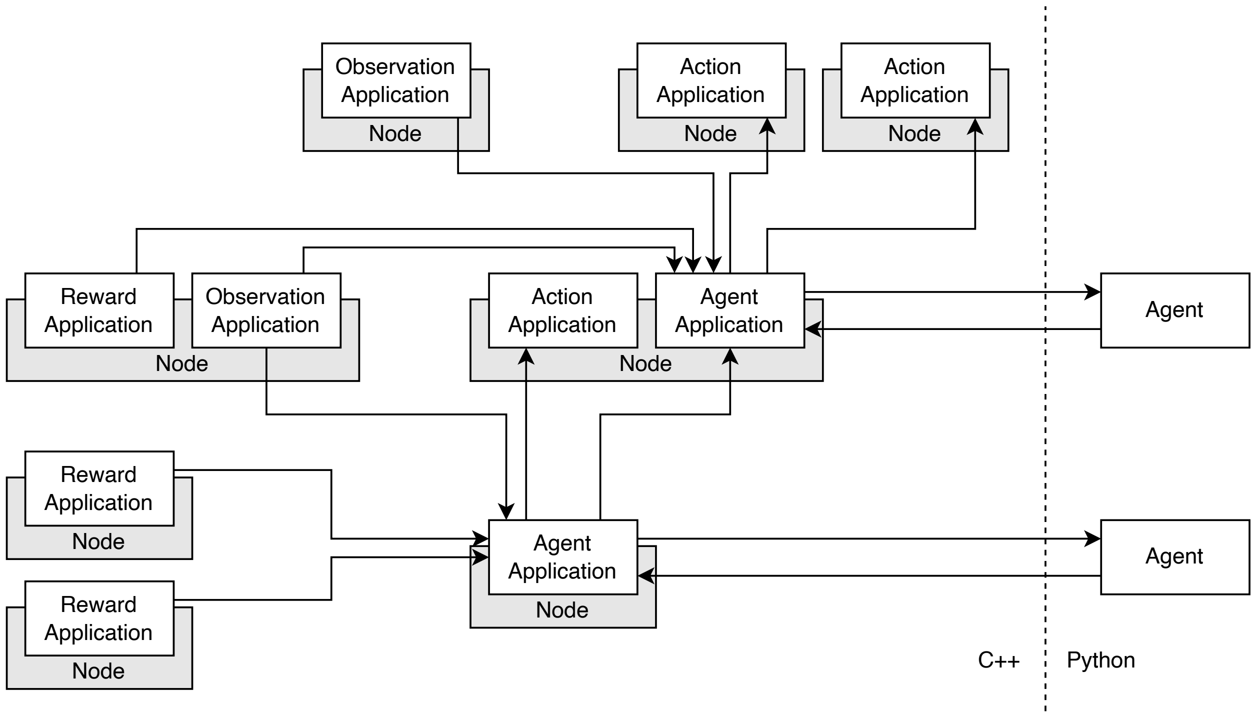


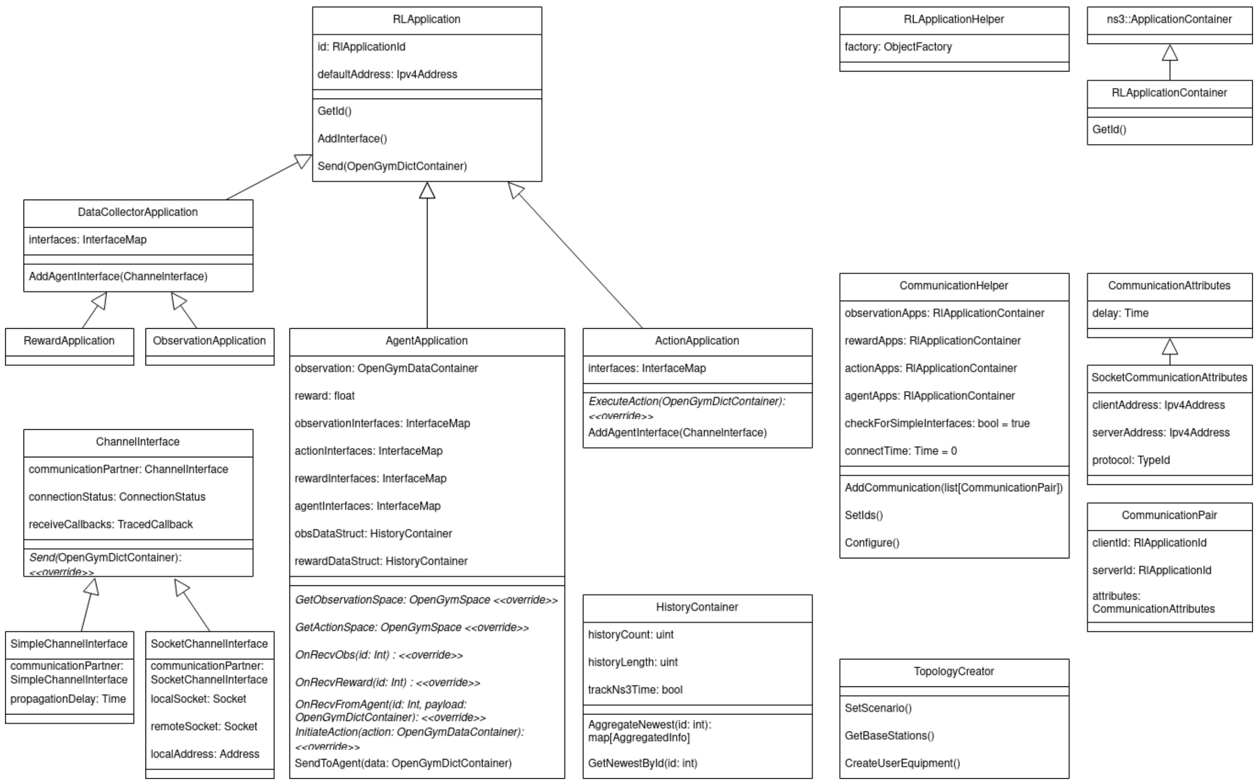
Fig. 2: Example scenario setup that should be supported by the framework

To make these generalized use cases possible, the following main requirements have been considered:

1. Support integration with existing *ns-3* scenarios with as few assumptions about the scenario as possible (even complex scenarios such as *Example scenario setup that should be supported by the framework* should be supported)
2. Support single-agent and multi-agent reinforcement learning (MARL)
3. Support communication between RL components via simulated network traffic

1.1.3 Class diagram

The following class diagram includes all classes provided by DEFIANCE. You can also find member variables and class methods that are particularly important.


RLApplication
id: RlApplicationId
defaultAddress: Ipv4Address
GetId()
AddInterface()
Send(OpenGymDictContainer)
RLApplicationHelper
factory: ObjectFactory
ns3::ApplicationContainer
RLApplicationContainer
GetId()
DataCollectorApplication
interfaces: InterfaceMap
AddAgentInterface(ChannelInterface)
RewardApplication
ObservationApplication
AgentApplication
observation: OpenGymDataContainer
reward: float
observationInterfaces: InterfaceMap
actionInterfaces: InterfaceMap
rewardInterfaces: InterfaceMap
agentInterfaces: InterfaceMap
obsDataStruct: HistoryContainer
rewardDataStruct: HistoryContainer
GetObservationSpace: OpenGymSpace <<override>>
GetActionSpace: OpenGymSpace <<override>>
OnRecvObs(id: Int) : <<override>>
OnRecvReward(id: Int) : <<override>>
OnRecvFromAgent(id: Int, payload: OpenGymDictContainer): <<override>>
InitiateAction(action: OpenGymDataContainer): <<override>>
SendToAgent(data: OpenGymDictContainer)
ActionApplication
interfaces: InterfaceMap
ExecuteAction(OpenGymDictContainer): <<override>>
AddAgentInterface(ChannelInterface)
CommunicationHelper
observationApps: RlApplicationContainer
rewardApps: RlApplicationContainer
actionApps: RlApplicationContainer
agentApps: RlApplicationContainer
checkForSimpleInterfaces: bool = true
connectTime: Time = 0
AddCommunication(list[CommunicationPair])
SetIds()
Configure()
CommunicationAttributes
delay: Time
SocketCommunicationAttributes
clientAddress: Ipv4Address
serverAddress: Ipv4Address
protocol: TypeId
CommunicationPair
clientId: RlApplicationId
serverId: RlApplicationId
attributes: CommunicationAttributes
ChannelInterface
communicationPartner: ChannelInterface
connectionStatus: ConnectionStatus
receiveCallbacks: TracedCallback
Send(OpenGymDictContainer): <<override>>
SimpleChannelInterface
communicationPartner: SimpleChannelInterface
propagationDelay: Time
SocketChannelInterface
communicationPartner: SocketChannelInterface
localSocket: Socket
remoteSocket: Socket
localAddress: Address
HistoryContainer
historyCount: uint
historyLength: uint
trackNs3Time: bool
AggregateNewest(id: int): map[AggregatedInfo]
GetNewestById(id: int)
TopologyCreator
SetScenario()
GetBaseStations()
CreateUserEquipment()

1.2 Customization

This module provides a framework to simulate different RL components by different `RlApplications`. The main tasks that the framework performs for the user in order to make it well usable are the following:

- provide frameworks for prototypical `RlApplications`,
- provide helper functionality to support creation of `RlApplications` and their installation on `Nodes`,
- enable typical communication between `RlApplications`, and
- handle the interaction between `RlApplications` and the Python-based training/inference processes in compliance with the typical RL workflow.

In addition to these tasks performed by the framework, some aspects of the `RlApplications` strongly depend on the specific RL task and solution approach that is to be implemented. Therefore, custom code provided by the user of the framework has to be integrated into the `RlApplications`. Typically, this mainly concerns the following aspects of `RlApplications`:

- Data collection: How are observations and rewards collected/calculated exactly?
- Communication between `RlApplications`: When and to whom are messages sent?
- Behavior of agents: At what frequency does the agent step? What triggers a step?
- Execution of actions: What happens exactly when a specific action occurs?

A typical example of necessary customization is an `ObservationApplication` which should be registered at a specific *ns-3* trace source to provide it with the necessary data. The according trace source and its signature have to be configurable as they depend on the specific scenario. Additionally it should be configurable to which `AgentApplications` the collected data is sent.

One option to solve this task are callbacks: The user could create functions outside the according `RlApplication` with a distinct interface. Those could then be registered as callbacks in the according `RlApplication`. Whenever user-specific code is required, the `RlApplication` would then call these callbacks. Similarly, the `RlApplication` could provide a method with a distinct interface. The user then has to register this method at a trace source to provide the `RlApplication` with data. This option is not very flexible as all function signatures have to be fixed and known already when the `RlApplication` class is designed. Another drawback of this approach is that there is no defined location for the custom code of an `RlApplication`.

Therefore, an approach using inheritance was chosen: The `RlApplications` are designed as abstract classes from which the user has to inherit in order to add the scenario-specific code. This has the advantage that all code connected to an `RlApplication` is collected in a single class. Additionally, it guarantees that all necessary methods are implemented and usable defaults can be implemented for methods that may be customized.

1.3 ChannelInterface

This framework is supposed to allow communication between `RlApplications` in a custom scenario. Therefore, it is the task of the framework user to set up the scenario and the communication channels between `Nodes`. This implies that the user has to provide the framework with an abstraction of a pre-configured channel over which data can be sent. Intuitively, this would be sockets. Nevertheless, the framework should prevent the user from the overhead of creating sockets. That is why the framework uses IP addresses and the type of protocol as data the user has to provide. Using this data, sockets can be created and connected to each other.

`RlApplications` should handle the interfaces of their communication channels transparently, e.g. independent from the protocol type. Additionally, direct communication without simulated network traffic should be possible. To this end, the `ChannelInterface` class was introduced as a generalized interface used in `RlApplications`. It is subclassed by the `SocketChannelInterface` class, which is responsible for creating sockets when provided with the necessary information (IP addresses and protocol type). The `SimpleChannelInterface` provides the `RlApplications` with the same interface while maintaining a direct reference to another `SimpleChannelInterface` to allow communication with a fixed delay (which might also be 0).

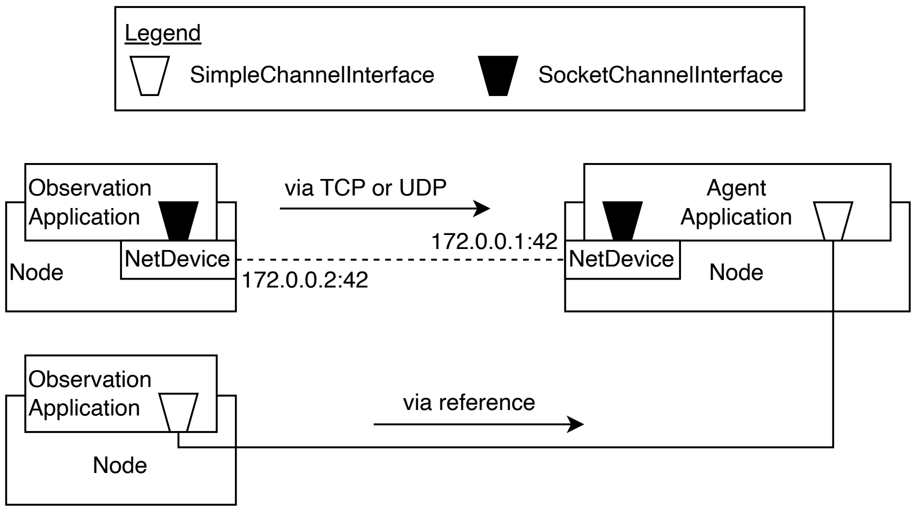


Fig. 3: Communication via `SimpleChannelInterface` and `SocketChannelInterface`

It should be noted that the framework should support multiple connections over `ChannelInterfaces` between a single pair of `RlApplications` to allow using different communication channels.

Simulating communication between `RlApplications` over simulated network channels includes the chance that a channel is broken and that therefore no communication is possible. This has to be handled by the underlying protocols or the user of the framework, since the user is responsible for the whole setup and configuration of the concrete network scenario.

1.4 Design of RlApplications

1.4.1 RlApplication

The `RlApplication` generalizes functionality that is equal among all applications provided by this module. This includes IDs to identify specific `RlApplication`, functionality to send and to handle `ChannelInterfaces`. In this way a generalized interface for all possible RL applications is established which can be used by all classes handling all kinds of RL applications, like the `CommunicationHelper` introduced in *Helper*.

In theory, multiple `RlApplications` of the same type can be installed on the same `Node`. Nevertheless, this was not tested yet since in most cases tasks of the same type (e.g. collecting observations) do not have to be separated into different applications when performed on the same `Node`.

1.4.2 AgentApplication

Basic Concept

The `AgentApplication` represents an RL agent (which is trained with e.g. RLLib) within the network. It has a scenario-specific observation and action space. Currently, the framework is tested only with fixed observation and action spaces (and not with parametric action spaces).

Interaction with other RlApplications

The `AgentApplication` may receive observations and rewards from one or multiple `ObservationApplications` resp. `RewardApplications`. To support as many use cases as possible, it is also supported to receive any data from `ObservationApplications` resp. `RewardApplications`, which is not immediatly used as observations or rewards but from which observations and rewards are derived by custom calculations. Therefore, the data transmitted from `ObservationApplications` to `AgentApplications` (which is called observation in the following) does not necessarily fit into the observation space of the agent. Likewise, an `AgentApplication` can send actions (or any data derived from it's actions) to one or multiple `ActionApplications`.

Additionally to the common RL interactions, this framework also supports transmitting arbitrary messages between `AgentApplications`. This provides users of this framework with the chance to implement a protocol for agent communication. Furthermore, it is the basis for exchanging model updates or policies between agents.

Interaction with Python-based learning process

The `AgentApplication` is intended to interact with the Python-based training/inference processes over the `OpenGymMultiAgentInterface`. This is primarily done by the `AgentApplication::InferAction` method(s), which call(s) `OpenGymMultiAgentInterface::NotifyCurrentState`. This interaction can happen timer-based (i.e. in fixed time intervals) or event-based (e.g. depending on how many observations were received). To have always access to the current observation and reward, which shall be sent to the Python side, the `AgentApplication` stores an `m_observation` and `m_reward` object.

Receiving, storing and calculating observations resp. rewards

To allow the `AgentApplication` to arbitrarily calculate observations and rewards based on the messages received from `ObservationApplications` and `RewardApplications`, these received messages have to be stored in the `AgentApplication`. For this purpose a new data structure, called `HistoryContainer` was designed. Each `AgentApplication` maintains one `HistoryContainer` for observations (`m_obsDataStruct`) and one for rewards (`m_rewardDataStruct`). `m_obsDataStruct` stores one deque for each connected `ObservationApplication` in which the newest `m_maxObservationHistoryLength` observations received from this `ObservationApplication` are stored. Additionally, `m_obsDataStruct` contains another deque, which stores the newest observations received independent from the `ObservationApplication`. `m_rewardDataStruct` is used equivalently. In this way, the user can specify how much observation and reward data is stored in the `AgentApplication` and use it arbitrarily.

Besides storing the received data, it is necessary to inform the `AgentApplication` when an observation or a reward is received. The user can then specify the behavior of the `AgentApplication` in response to such a message. For example, the `AgentApplication` could wait for 10 observations before inferring the next action. This is done by registering the abstract methods `AgentApplication::OnRecvObs` and `AgentApplication::OnRecvReward` at the according `ChannelInterfaces`.

This framework is intended to make communications between RL components more realistic. Nevertheless, it shall still support using global knowledge (e.g. knowledge available on other `Nodes`) to calculate rewards and observations. Particularly, global knowledge can be helpful to calculate rewards during offline training. If such global knowledge (i.e. data available without delay or communication overhead) shall be used, it can just be accessed when rewards and/or observations are calculated within the `AgentApplication` or data can be transmitted via `SimpleChannelInterfaces`.

Execution of actions

After the `AgentApplication` called `OpenGymMultiAgentInterface::NotifyCurrentState`, it receives an action via `AgentApplication::InitiateAction` from the Python side. To simulate the computation delay of the agent, an `actionDelay` can be configured in `OpenGymMultiAgentInterface::NotifyCurrentState`. Then the `OpenGymMultiAgentInterface` delays calling `AgentApplication::InitiateAction` by the configured actionDelay. Per default, `AgentApplication::InitiateAction` sends the received action to all connected `ActionApplications`. Because data is transmitted via `OpenGymDictContainers` between `RlApplications`, the received action is wrapped into such a container under the key "default". This method is intended to be overwritten if different behaviour is needed. In this way, the action can for example be divided into partial actions that are sent to different `ActionApplications`. Alternatively, one could also specify in a part of the action to which `ActionApplications` the action shall be sent.

Inference agents vs. training agents

In many RL tasks different agents perform inference and training. Therefore, one could provide different `AgentApplication` classes for these two purposes. Nevertheless, a general `AgentApplication` class, which can perform both inference and training is also necessary to support e.g. online training. Consequently, the `AgentApplications` used for inference and training would only be specializations of this class, which provide less functionality. That is why it was decided to leave it to the user to use only the functionality which is needed in the current use case. When it is necessary to differentiate between inference and training agents, this can be done e.g. by a flag introduced in a user-defined inherited `RlApplication`.

1.4.3 DataCollectorApplication

The `DataCollectorApplication` is the base class which is inherited by `ObservationApplication` and `RewardApplication` since both provide similar functionality: They collect scenario-specific data, maintain `ChannelInterfaces` connected to `AgentApplications`, and provide functionality to send over these interfaces. To register the applications at scenario-specific trace sources the user has to define a custom `ObservationApplication::Observe` resp. `RewardApplication::Reward` method with a custom signature within the custom `ObservationApplication` resp. `RewardApplication`. To provide a place to connect this custom method with an existing trace source, the abstract `DataCollectorApplication::RegisterCallbacks` method was created. If necessary, the user may also register multiple custom `ObservationApplication::Observe` resp. `RewardApplication::Reward` methods within `DataCollectorApplication::RegisterCallbacks`. To ensure that the callbacks are registered before the simulation starts, `DataCollectorApplication::RegisterCallbacks` is called in the `DataCollectorApplication::Setup` method.

Each `ObservationApplication` resp. `RewardApplication` can send observations resp. rewards to one or multiple `AgentApplications` in order not to limit possible scenarios.

1.4.4 ActionApplication

The `ActionApplication` provides functionality to maintain `ChannelInterfaces` which are connected to `AgentApplications` and to receive actions (in the form of `OpenGymDictContainers`). The abstract method `ActionApplication::ExecuteActions` is designed to provide a place for the user-specific code that handles the different actions. This method is automatically called when data is received on the registered `ChannelInterfaces`. Therefore, it is connected to the according callbacks within the `ActionApplication::AddAgentInterface` method.

1.4.5 General Decisions

All `RlApplications` have to store multiple `ChannelInterfaces` that connect them to other `RlApplications`. Typically, all `ChannelInterfaces` connected to a specific remote `RlApplication` are used together. Furthermore, multiple `ChannelInterfaces` between a pair of `RlApplications` have to be supported to enable communication over different channels. Therefore, InterfaceMaps were introduced, which are essentially two-dimensional maps. The outer map is unordered and maps `applicationIds` to a second ordered map. The second map maps an ID to the `ChannelInterface`. This ID is unique within this map of `ChannelInterfaces` connected to a specific `RlApplication`. To ensure this uniqueness, the entries are stored in ascending order of the IDs. In this way, one can simply use the last entry to generate a new unique ID. Connecting two `RlApplications` over multiple `ChannelInterfaces` is an edge case. Therefore, all `RlApplication::Send` methods are implemented with signatures that allow to send to a specific `RlApplication`. Nevertheless, storing `ChannelInterfaces` with IDs makes it possible to also provide methods to sent over a certain `ChannelInterface`.

We did not consider that during inference the agent might not be able to compute another action. In reality, the computation either needs to be queued ("single threaded") or processed in parallel ("multi threaded"). The latter case is different than the current implementation, because the individual inference times increase with increased parallelism. For a detailed discussion as how to extend the framework with this feature, see *Framework expansion options*

In complex scenarios with many `ObservationApplications` and `AgentApplications` each `ObservationApplication` should possibly be able to communicate with each `AgentApplication`. In this case, it is not practicable to configure all communication connections before the simulation started. Therefore, it is necessary to support dynamically adding and removing `ChannelInterfaces` during simulation time, which is done by `RlApplication::AddInterface` and `RlApplication::DeleteInterface` methods.

In some cases, one has to configure something within an `RlApplication` based on the attributes which were set but before the application is started. One example for this is the initialization of data structures with a scenario-dependent length. To provide a central place for such intialization functionality which cannot be placed in the constructor, the `RlApplication::Setup` method was created.

1.5 Interface for Multi-Agent RL

Gymnasium is a commonly used environment interface for single-agent training, which is also supported by *ns3-ai* [ns3-ai]. For multi-agent training Ray implemented the MultiAgentEnv API [MultiAgentEnv]. Besides this API, there is also the PettingZoo API [Pettingzoo] proposed by the Farama Foundation. Besides the Agent Environment Cycle (AEC) API, which is the main API of PettingZoo, exists also a Parallel API. For both APIs, RLLib provides a wrapper to make them compatible with the MultiAgentEnv [PettingzooWrapper].

Since this framework is intended to support multi-agent RL, it had to be decided which API to use. For the chosen API, the *ns3-ai* interface then had to be extended to support multi-agent RL.

The basic idea of the AEC [AEC] is that agents step sequentially and not in parallel. This restriction is intended to create a better understandable and less error-prone model to prevent developers for example from race conditions.

To decide for an API, the following aspects were considered:

- The AEC API is a subset of the MultiAgentEnv API, meaning that everything implemented with AEC API is representable with MultiAgentEnv. Using the AEC API would therefore add no functionality, but could be less error-prone because of its restrictions.
- For every step of an agent, observations and rewards have to be transferred from C++ to Python and an action back from Python to C++. To avoid difficulties with synchronizing agents, the most simple model is sequentially stepping agents. If agents should step simultaneously this can then be simulated by not continuing the simulation time between their steps.
- Including the AEC API when training with RLLib means including a further dependency and the environment would have to be wrapped into a MultiAgentEnv.
- According to [PettingzooWrapper], AEC expects agents to work in a cooperative manner. However, this framework should support also conflicting agents.
- Documentation of RLLib is not as comprehensive as it should be in some places. Nevertheless, there are many code examples for RLLib online to look up.

For these reasons, it was decided to use the MultiAgentEnv API instead of the PettingZoo API, but apply the restriction of sequentially stepping agents when expanding *ns3-ai*.

This framework should support both single-agent and multi-agent RL. To provide a uniform interface without code duplication, this framework handles single-agent RL as a special case of multi-agent RL.

Communication between the Python-based training process and the simulation in C++ works over the `Ns3MultiAgentEnv` (in Python) and the `OpenGymMultiAgentInterface` (in C++), which were added to *ns3-ai*. The training/inference process is then initiated by the Python side using `Ns3MultiAgentEnv`. The Python

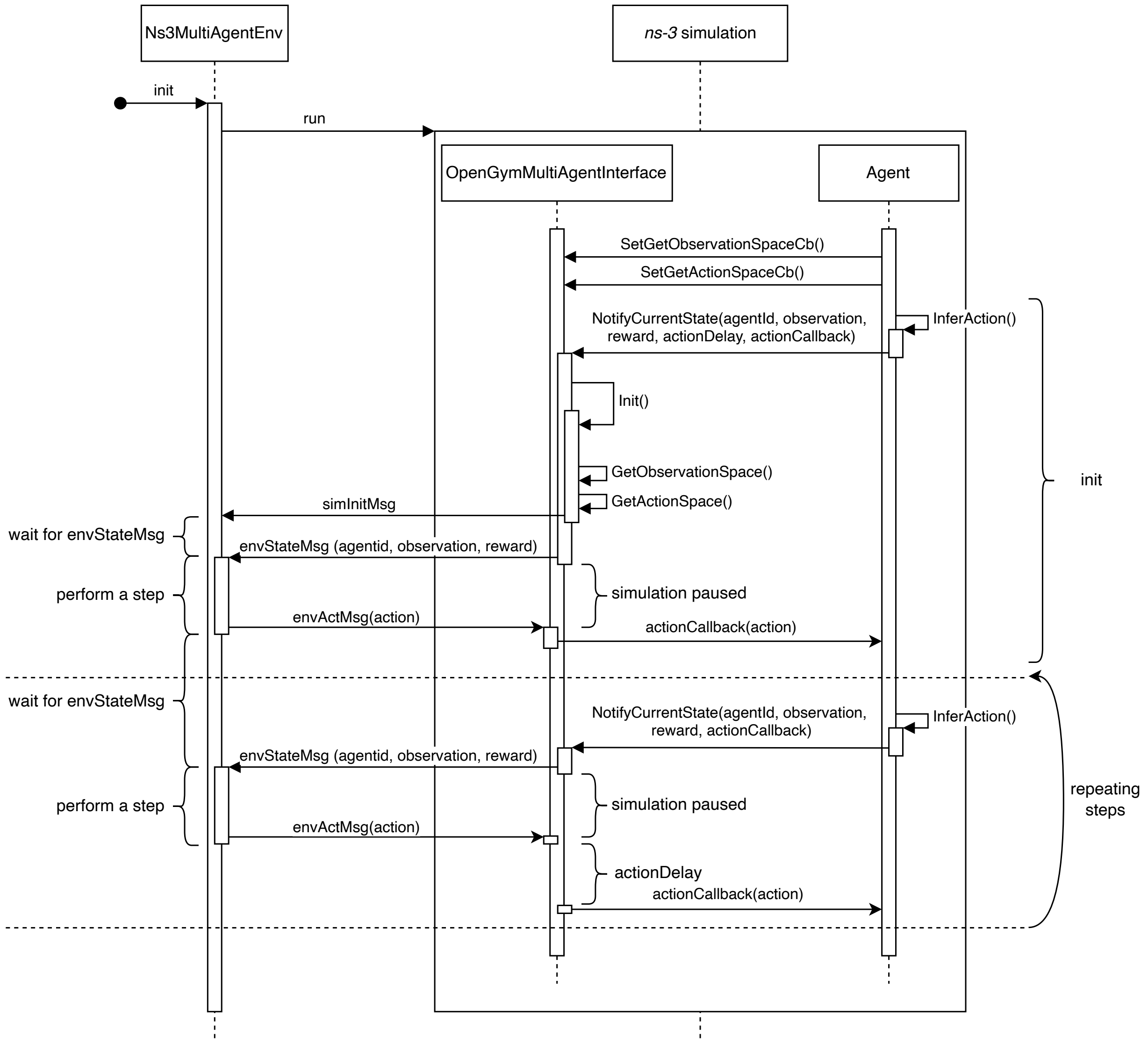


Fig. 4: Interaction between *ns-3* simulation (C++) and `Ns3MultiAgentEnv` (Python)

process starts the *ns-3* simulation process (implemented in C++) as a subprocess and waits for receiving observations and rewards from the C++ process. Whenever an agent decides to step (via the `AgentApplication::InferAction` method), the C++ process running the *ns-3* simulation switches back to the Python process via the `OpenGymMultiAgentInterface::NotifyCurrentState` method with the observation and the reward of the according agent. The Python process answers with an action for this agent. Only then, the simulation is resumed and the callback registered in `OpenGymMultiAgentInterface::NotifyCurrentState` is called with the action. Note the one to one relation between environment steps and calls to `AgentApplication::InferAction`. If the simulation does not call `AgentApplication::InferAction`, the environment won't step.

1.6 Helper

In a typical use case this framework has to be integrated into an existing *ns-3* scenario. In *ns-3*, the concept of helpers is commonly used to simplify the configuration and setup tasks the user has to perform.

In *ns-3.42* an `ApplicationHelper` was introduced, which is used to create and install applications of a specified type on `Node`s. To avoid repeating casts, which would lead to very cluttered code, an `RlApplicationHelper` was introduced by this framework which returns `RlApplicationContainer`s instead of `ApplicationContainer`s.

The main configuration task of this framework is the setup of all communication connections between `RlApplications`, e.g. the connection of all `ObservationApplication`s to their according `AgentApplication`s. For this purpose, the `CommunicationHelper` was created. The framework should allow all possible connections between pairs of `RlApplication`s without making any restricting assumptions. This is done by letting the user configure the communication relationships via an adjacency list. Thereby, it is even possible to configure multiple different connections, e.g. over different channels between two `RlApplication`s.

To allow the user to identify `RlApplication`s e.g. when passing them to this adjacency list, `RlApplicationId`s were introduced. They consist of a part identifying the `applicationType` (e.g. `ObservationApplication`) and an `applicationId` which is unique among all `RlApplication`s of this type. In this way, the `applicationType` can be identified when necessary and whenever the `applicationType` is clear, only the `applicationId` is used for identification. The `CommunicationHelper` is also used for creating these unique Ids. To do this, it needs to have access to all `RlApplication`s existing in a scenario. One option for this is to create all `RlApplications` within the `CommunicationHelper`. This requires the user to provide the `CommunicationHelper` with all `Node`s and the according:code:*applicationType*s to install on them. However, this would just move the identification problem to the level of the `Node`s. Additionally, this approach would conform less with the general idea that the user defines the location of applications by installing them on `Node`s. That is why, the tasks of creating/installing `RlApplication`s and configuring them and their communication relationships was split between the `RlApplicationHelper` and the `CommunicationHelper`. In this way, it is required that the user passes all `RlApplications` to the `CommunicationHelper`. Then the `RlApplicationId`s can be set by the `CommunicationHelper` via the `CommunicationHelper::SetIds` method.

Besides a pair of `RlApplicationId`s, the user has to specify in the adjacency list all attributes that are necessary to configure the connection between these `RlApplication`s. This is done via `CommunicationAttributes` as a compact format for all possible configuration data. If no information (i.e. `{}`) is provided by the user, the framework will establish `SimpleChannelInterface`s, so that as little configuration is required as possible. If `SocketCommunicationAttributes` are provided, the `CommunicationHelper` is responsible for creating the according `ChannelInterface`s and connecting them. The main goal when designing this configuration interface was to enable as many configurations as possible, while making as few configurations as possible necessary. That is why, e.g. a default protocol for `SocketCommunicationAttributes` and default IP addresses for each `RlApplication` (that is derived from the list of network interfaces of its `Node`) were implemented.

The `CommunicationHelper::Configure` method was introduced to make it possible to simultaneously call the `RlApplication::Setup` method on all `RlApplication`s at a time which is independent from e.g. the constructors, so that it can be done after setting the `RlApplicationId`s but before setting up the communication relationships. The methods `CommunicationHelper::Configure` and `CommunicationHelper::SetIds` could be called combinedly in a single method, so that the user does not have to call two methods. However, this was not done so far because both methods perform very different tasks.

1.7 Framework expansion options

- Create interface for sharing model updates or policies between agents. (already implemented to a large extent)
  - In some network infrastructures it is necessary to outsource training to a remote server, to share learned model updates, or to share policies between participants. To simulate resulting constraints and research possible opportunities it is required to realistically simulate the performance of shared updates and policies as well as their size. This feature addresses issues like:
    - How is performance affected when learning distributedly?
    - What burden does resulting communication pose on a network and can it be reduced?
  - The required communication functionality is already implemented to a large extent: On the *ns-3* side, in `AgentApplication::OnRecvFromAgent` logic to handle model weights, experience, and model update messages need to be handled by the agent. The message flow is depicted in *Interaction of inference agents, trainings server, and the ns3-ai message interface*.

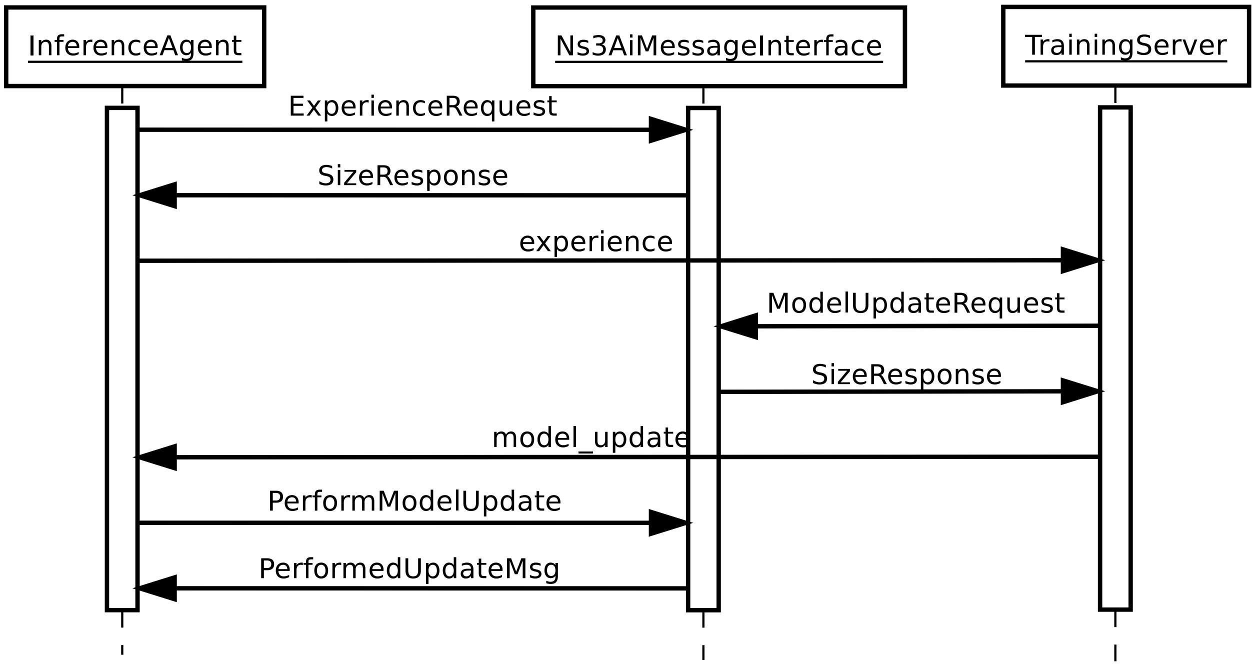


Fig. 5: Interaction of inference agents, trainings server, and the ns3-ai message interface

  This message flow is fully implemented; only the ns3-ai message handling on the Python side alongside the interaction with Ray is still missing.

- Support moving agents (and other `RlApplications`) to another `Node`. (not started)
  - In complex scenarios it might be required to change the `Node` from which the agent receives its observations or where it performs its actions. Currently, this would require installing `ObservationApplications` and `ActionApplications` on every possible `Node` and then switch between them when sending. Since this is prone to bugs at runtime and difficult to track especially for bigger scenarios, it would be more handy to move an existing application to a different `Node`. The same applies if agents shall switch the `Node` during simulation time. This would be possible via model updates if an `AgentApplication` was installed on

every possible `Node`. However, it would be much easier if it would be possible to move an application to another `Node`.

- Checkpointing (almost done)
  - To simulate inference without training or continue training of promising policies, it is required to implement Ray's checkpointing. We have already implemented inference runs. However, continuing training hasn't been tested yet.
- Multithreading vs. Singlethreading (not started)
  - What happens if multiple observations arrive while the agent is already inferring an action? In a realistic scenario with limited resources, the agent might only be capable of starting a limited amount of threads for inference. Then, increased parallelism increases the inference times for a job. Maybe the node is even single-threaded. To provide inference for all observations it would be required to buffer some of the observations. This feature would allow to simulate thereby introduced latency as well as additional limitations in regard to the buffer size. Scenarios could explore questions like: Which buffer strategies are sensible for overall performance if the buffer is full? How beneficial is it to provide more resources for the agent in order to allow multithreading? This would lead to quantifiable answers to complex optimization problems.

CHAPTER

TWO

USER DOCUMENTATION

2.1 Overview

The *ns-3* DEFIANCE module is a reinforcement learning (RL) framework for *ns-3*. It allows the simulation of distributed RL in networks. It can handle single-agent and multi-agent RL scenarios.

The user performs the following steps to carry out the RL experiment:

1. Implement the network topology and traffic using standard *ns-3* code.
2. To define how observations and reward signals are collected, how actions are executed in the environment and how the agents perform inference and training, subclass from the provided *RL-Applications*. Abstract classes for the different subtasks are provided via the *AgentApplication*, *ObservationApplication*, *RewardApplication* and *Action-Application*. These applications are installed in the simulation via the *RlApplicationHelper*.
3. Specify how data is exchanged between these components and specify the communication structure via channels. The lowest level of abstraction our framework proposes is using ChannelInterfaces for this. The framework also provides a *CommunicationHelper* class to simplify the communication setup.
4. Finally, use the utilities provided by *ns3-ai* to interact with the simulation as an RL environment.

Note: An in-depth documentation of the multi-agent interface we added to *ns3-ai* can be found here. This interface is also used by the DEFIANCE framework. In case you are interested in how the framework functions or think about extending DEFIANCE, we recommend to take a look at these docs or the *blog post <https://medium.com/@oliver.zimmermann/reinforcement-learning-in-ns3-part-1-698b9c30c0cd>* we wrote about it.

2.2 RL-Applications

2.2.1 RlApplication

All presented applications inherit from `RlApplication`. Therefore, all of them can access the following functionality.

RlApplicationId

A common use case is to identify the `RlApplication` information has been sent to or received from. To accomplish this, each `RlApplication` has a unique identifer, their `RlApplicationId`. This ID can be set by passing an `RlApplicationId` to `RlApplication::SetId`. `RlApplicationId` is a struct consisting of an `uint32_t applicationId` and an `ApplicationType applicationType`.

This introduces four different types of `RlApplication`:

- `OBSERVATION`
- `REWARD`
- `AGENT`
- `ACTION`

If the provided `CommunicationHelper` is not used, one must set all `RlApplicationIds` manually to be able to use inter-application communication.

2.2.2 AgentApplication

The `AgentApplication` is where inference is performed. This code is an example of a possible implementation followed by an explanation.

```cpp
class InferenceAgentApp : public AgentApplication
{
public:
    InferenceAgentApp()
        : AgentApplication()
    {
    };

    ~InferenceAgentApp() override{};

    static TypeId GetTypeId()
    {
        static TypeId tid = TypeId("ns3::InferenceAgentApp")
                                .SetParent<AgentApplication>()
                                .SetGroupName("defiance")
                                .AddConstructor<InferenceAgentApp>();
        return tid;
    }

    void Setup() override
    {
        AgentApplication::Setup();
        m_observation = GetResetObservation();
        m_reward = GetResetReward();
    }

    void OnRecvObs(uint id) override
    {
        m_observation = m_obsDataStruct.GetNewestByID(id)
                  ->data->Get("floatObs")
                  ->GetObject<OpenGymBoxContainer<float>>();
        InferAction();
    }
```

(continues on next page)

(continued from previous page)

```
34
35      void OnRecvReward(uint id) override
36      {
37          m_reward = m_rewardDataStruct.GetNewestByID(0)
38                      ->data->Get("reward")
39                      ->GetObject<OpenGymBoxContainer<float>>()
40                      ->GetValue(0);
41      }
42
43      Ptr<OpenGymDataContainer> GetResetObservation()
44      {
45          // This method returns the initial observation that is used after resetting␣
    ↪the environment.
46          uint32_t shape = 4;
47          std::vector<uint32_t> vShape = {shape};
48          auto obj =  CreateObject<OpenGymBoxContainer<float>>(vShape);
49          for(auto i = 0; i < shape; i++ ){obj->AddValue(0);}
50          return obj;
51      }
52
53      float GetResetReward()
54      {
55          // This method returns the initial reward that is used after resetting the␣
    ↪environment.
56          return 0.0;
57      }
58
59  private:
60      Ptr<OpenGymSpace> GetObservationSpace() override
61      {
62          uint32_t shape = 4;
63          std::vector<uint32_t> vShape = {shape};
64          std::string dtype = TypeNameGet<float>();
65
66          std::vector<float> low = {-4.8 * 2, -INFINITY, -0.418 * 2, -INFINITY};
67          std::vector<float> high = {4.8 * 2, INFINITY, 0.418 * 2, INFINITY};
68
69          return CreateObject<OpenGymBoxSpace>(low, high, vShape, dtype);
70      }
71
72      Ptr<OpenGymSpace> GetActionSpace() override
73      {
74          return MakeBoxSpace<int>(1, 0, 1);
75      }
76  };
```

To implement your own `AgentApplication` it is necessary to inherit from `AgentApplication` in order to access all features provided by our framework. This can be seen in line 1. The method `InferenceAgentApp::GetTypeId` (lines 11-18) is mandatory, as it is part of the *ns-3* library. Since our classes inherit from `ns3::Object` one has to provide this method to allow the usage of *ns-3*-factories and *ns-3*-pointers.

`InferenceAgentApp::Setup` is called at the beginning of the scenario and ensures that all required variables for inference are initialized. It is adviced to call the parent method (line 22) since it informs the MARL interface about the action and observation-space provided by `InferenceAgentApp::GetObservationSpace` and `InferenceAgentApp::GetActionSpace`. Aditionally the `InferenceAgentApp::Setup` method can be used to initialize `m_observation` and `m_reward` since this method should always be called before the first occurence of inference and thereby guarantees that no uninitialized variables will be used for inference.

`m_observation` and `m_reward` are two inherited variables from the `AgentApplication` class. `m_observation` is an `OpenGymDataContainer` that stores the observations used for inference. `m_reward` is simply a float value representing the current reward. Both of this variables are passed to the MARL interface when `AgentApplication::InferAction` is called (line 32).

`InferenceAgentApp::OnRecvObs` and `InferenceAgentApp::OnRecvReward` are called when the `AgentApplication` receives an observation or reward. The `id` is the ID of the `RlApplication` that sent the data. It can be used to retrieve the desired data from `m_obsDataStruct` or `m_rewardDataStruct` by calling `HistoryContainer::GetNewestByID(id)` (line 29). However, there is no restriction on how to update `m_observation` or whether `InferenceAgentApp::OnRecvObs` should be used at all.

Both of the data structures `m_obsDataStruct` and `m_rewardDataStruct` are instances of type `HistoryContainer`. Once a reward or observation is received, the `AgentApplication` ensures both are updated accordingly before calling `InferenceAgentApp::OnRecvObs` or `InferenceAgentApp::OnRecvReward`.

In line 32, the method `AgentApplication::InferAction` is called. As mentioned earlier passes this method all required parameters for inference to the MARL interface. Aditionally, a callback is passed on that sends the returned action from the Python side to an `ActionApplication`. Pass the `RlApplicationId::applicationId` to `AgentApplication::InferAction` as in `InferAction(id)` if the received action should only be send to an specific `ActionApplication`. Otherwise the action will be sent to all registered instances. It is not required to call this method in `InferenceAgentApp::OnRecvObs`. For example `AgentApplication::InferAction` could also be called in a method that is scheduled at equally spaced timesteps or after an *ns-3*-event. If preferred, it is even possible to call inference outside of the `ActionApplication`. Since `AgentApplication::InferAction` is by design protected within `AgentApplication`, this would require the usage of `OpenGymMultiAgentInterface::NotifyCurrentState` and thus thorough testing.

`InferenceAgentApp::OnRecvReward` is similar to `InferenceAgentApp::OnRecvObs` in terms of when it is called and its purpose. Both of these methods allow to aggregate over the data received by multiple `RlApplication` instances. One example could calculate the min of all rewards sent by `RewardApplication` in the method `InferenceAgentApp::OnRecvReward`.

`InferenceAgentApp::GetResetObservation` and `InferenceAgentApp::GetResetReward` are vital after a reset of the environment. Therefore, they must be implemented in the inheriting class. When setting up a scenario without training a Ray agent and no resets, they are optional, yet it is essential to initialize `m_observation` and `m_reward` at the beginning of the scenario (e.g. ll.23-24).

The last two important methods are `InferenceAgentApp::GetObservationSpace` and `InferenceAgentApp::GetActionSpace`. These methods are mandatory because they inform the MARL interface about the dimensions of the respective spaces. Information about the different spaces have to be provided in instances of `OpenGymSpace`. An exemplary creation of such spaces can be seen in line 62 to 69. These spaces as well as the `OpenGymDataContainer` are part of the *ns-3*-ai library. To reduce the overhead of creating an `OpenGymSpace` or `OpenGymDataContainer`, some useful functions are provided in `base-test.h`. An example usage of one of these functions can be seen in line 74.

Additional Features and Use-Cases

Configure History Containers

The length of `m_rewardDataStruct` and `m_obsDataStruct` can be changed by setting the attribute `MaxRewardHistoryLength` or `MaxObservationHistoryLength`.

It is also possible to save a timestamp, marking the time of arrival in `m_rewardDataStruct` and `m_obsDataStruct`. If this feature is required, set `ObservationTimestamping` or `RewardTimestamping` to true. More information is given in *Data History Container*.

Provide Extra Info

To pass extra info to the environment, override the method `AgentApplication::GetExtraInfo`

```
/* ... */
private:
    std::string m_importantMessage;
    std::map<std::string, std::string> GetExtraInfo() override
    {
        std::map<std::string, std::string> info;
        info["agent"] = m_importantMessage;
        return info;
    }
```

Action Delay

To simulate the time required to calculate inference, a delay can be set between receiving an action and performing the callback specified for action execution.

```
/* ... */
private:
    std::string m_importantMessage;
    Time GetActionDelay() override
    {
        return Seconds(1);
    }
```

Override initiateAction and initiateActionForApp

After inference took place, either of these methods is invoked with the returned action from the MARL interface. This method then sends the received message to either all registered `ActionApplication` or the one that matches `remoteAppId`. Overriding this method allows for example to only send over a specific `ChannelInterface`.

```
/* in your AgentApplication-class: */
protected:
    uint32_t interfaceToUse;
    void InitiateActionForApp(uint remoteAppId, Ptr<OpenGymDataContainer> action)
    {
        SendAction(MakeDictContainer("default", action), remoteAppId, interfaceToUse);
    }
```

OnRecvFromAgent

To specify how an `AgentApplication` should handle messages from another `AgentApplication`, override this method.

The method receives a `remoteAppId` matching the `RlApplicationId::applicationId` of the `AgentApplication` that send the data and the message itself as a `Ptr<OpenGymDictContainer>`. Here is an example for this:

```
/* in your AgentApplication-class: */
protected:
    uint32_t agentOfInterest;
    void OnRecvFromAgent(uint remoteAppId, Ptr<OpenGymDictContainer> payload)
    {
        if(remoteAppId == agentOfInterest)
        {
            message = payload->Get("parameter")
                     ->GetObject<OpenGymBoxContainer<float>>()
                     ->GetValue(0);
        }
    }
```

If desired, a new `HistoryContainer` can be added to the class which can be used to store and retrieve the received agent messages in a similar fashion as the observations and rewards.

2.2.3 ObservationApplication

The main purpose of the `ObservationApplication` is to send observations to the agent. Therefore, the class is equipped with the methods `ObservationApplication::RegisterCallbacks` and `ObservationApplication::Send`. To implement an `ObservationApplication`, create a child class that inherits from `ObservationApplication`. This also requires overriding `GetTypeId` in a similar fashion as seen earlier in the `AgentApplication` example.

ObservationApplication::RegisterCallbacks

This method allows registration of callbacks to trace sources. This ensures the `ObservationApplication` is always informed when a value that should be observed changes.

```
class YourImplementation : public ObservationApplication{
  public:
    /* ... */
    void
    RegisterCallbacks() override
    {
        DynamicCast<YourNode>(GetNode())->m_reportYourTrace.ConnectWithoutContext(
            MakeCallback(&YourImplementation::Observe, this));
    }
    void Observe(/*values provided by the trace source*/)
    {
        /* send observation or wait for more observation */
    }
}
```

Note: It can be tricky to access the required trace source inside the `ObservationApplication` class, especially if the trace source is not provided by *ns-3*. In this example, the costume trace source is accessed by inheriting the `Node` class and adding the trace source as a class member. All `ns3::Application` instances can access the node they are installed on with `GetNode`. Alternatively, trace sources can be accessed by a *ns-3* path. Look into the *ns-3* documentation for more information.

ObservationApplication::Send

Once the `ObservationApplication` is satisfied with the observations, it can send these observations to registered instances of `AgentApplication`. This functionality is offered by the base class. The observations have to be wrapped into an `OpenGymDictContainer`. If an observation should only be sent to a specific agent, pass the `RlApplicationId` to `ObservationApplication::Send`. Furthermore, the ID of the `ChannelInterface` can be provided. If not provided, the observation is sent to all registered instances.

```
class YourImplementation : public ObservationApplication{
  public:
    /* ... */
    void Observe(uint32_t value)
    {
        /*create OpenGymDataContainer */

        Send(/*OpenGymDictContainer*/);
        // or
        Send(/*OpenGymDictContainer*/, remoteId, interfaceId);
    }
}
```

2.2.4 RewardApplication

The `RewardApplication` is in its functionality similar to `ObservationApplication` since both classes inherit from the same base class. A reward should be sent to an instance of `AgentApplication` once a relevant event is triggered. To accomplish that the `RewardApplication::Send` is provided. It is required to wrap all reward information into an `OpenGymDictContainer`.

```
class YourImplementation : public RewardApplication{
  public:
    /* ... */
    void
    RegisterCallbacks() override
    {
        DynamicCast<YourNode>(GetNode())->m_reportYourTrace.ConnectWithoutContext(
            MakeCallback(&YourImplementation::ObserveReward, this));
    }
    void ObserveReward(/*values provided by the trace source*/)
    {
        /*create OpenGymDataContainer */

        Send(/*OpenGymDictContainer*/);
        // or
        Send(/*OpenGymDictContainer*/, remoteID, interfaceId);
    }
}
```

2.2.5 ActionApplication

The `ActionApplication` receives actions and executes them. Therefore, upon receiving an action from an `AgentApplication`, the virtual method `ActionApplication::ExecuteAction` is triggered. To specify what action should be performed, override the `ActionApplication::ExecuteAction` in a child class.

ActionApplication::ExecuteAction

In this method, two parameters are accessible. `remoteAppId` is the `RlApplicationId` of the `AgentApplication` that sent the action. `action` is an `OpenGymDictContainer` that contains the sent action out of the action space. An exemplary retrieval of the actual content of `action` is provided in line 19. `action->Get("default")` returns an `OpenGymDataContainer`. Therefore, it is necessary to dynamically cast this `OpenGymDataContainer` to the type that was sent by the `AgentApplication` (e.g. with `GetObject<OpenGymBoxContainer<int>>()`). If the content of `action` at key: `"default"` doesn't match the type passed to `GetObject`, a null pointer will be returned even if its only a mismatch in the provided data type for `OpenGymBoxContainer`.

Note: Make sure that the `OpenGymDictContainer action` actually contains the key passed by `action->Get("default")`. The `AgentApplication::InitiateAction` will always wrap the received action from the MARL interface into an `OpenGymDictContainer` with the key `"default"`. However, if this method was overridden in a child class, a different key is possible.

```
1  class YourActionApp : public ActionApplication
2  {
3  public:
4      YourActionApp(){};
5      ~YourActionApp() override{};
6
7      static TypeId GetTypeId()
8      {
9          static TypeId tid = TypeId("ns3::YourActionApp")
10                                 .SetParent<ActionApplication>()
11                                 .SetGroupName("defiance")
12                                 .AddConstructor<YourActionApp>();
13         return tid;
14     }
15
16     void ExecuteAction(uint32_t remoteAppId, Ptr<OpenGymDictContainer> action)␣
   ↪override
17     {
18         // auto m_objectActionIsPerformedOn = DynamicCast<objectActionIsPerformedOn>
   ↪(GetNode());
19         auto act = action->Get("default")->GetObject<OpenGymBoxContainer<int>>()->
   ↪GetValue(0);
20
21         m_objectActionIsPerformedOn->SetValue(acc);
22     }
23
24     void SetObservationApp(Ptr<ActionObject> object)
25     {
26         m_objectActionIsPerformedOn = object;
27     }
28
29 private:
```

(continues on next page)

(continued from previous page)

```
30      Ptr<ActionObject> objectActionIsPerformedOn;
31  };
```

To perform the action, the `ActionApplication` needs a reference to the object it perfoms the action on. One solution would be to pass it to the application as seen in line 24-27. Alternatively, the `ActionApplication` could access the node it is installed on.

2.2.6 Communication between RL-Applications

Add interfaces

To properly use the RL applications, connect them to one another via the *ChannelInterface*. The `RlApplication` interface provides the method `RlApplication::AddInterface` to register a `ChannelInterface`. Two applications can be connected over multiple instances of `ChannelInterface`, enabling potential multipath functionality. To index the different `ChannelInterface` between two applications, an `interfaceId` has to be provided. `RlApplicationId` in combination with the `interfaceId` represents an unique identifer for a connection between two instances of `RlApplication`.

`AddInterface` also sets up necessary callbacks for receiving messages.

```
//code to create your agent
RlApplicationHelper helper(TypeId::LookupByName("ns3::YourAgentClass"));
helper.SetAttribute("StartTime", TimeValue(Seconds(0)));
helper.SetAttribute("StopTime", TimeValue(Seconds(10)));
RlApplicationContainer agentApps = helper.Install(agentNode);

//code to create your observationApp
helper.SetTypeId("ns3::YourObservationApp");
RlApplicationContainer observationApps = helper.Install(obsNode);

RlApplicationId remoteIdObservationApp = DynamicCast<YourObservationApp>(observationApps.Get(0))->GetId();
Ptr<YourAgentClass> agent = DynamicCast<YourAgentClass>(agentApps.Get(0));
uint interfaceAtAgentId = agent->AddInterface(remoteIdObservationApp, ptrToChannelInterface);

RlApplicationId remoteAgentId = DynamicCast<YourAgentClass>(agentApps.Get(0))->GetId();
Ptr<YourObservationApp> obsApp = DynamicCast<YourObservationApp>(observationApps.Get(0));
uint interfaceAtObservationId = obsApp->AddInterface(remoteAgentId, ptrToChannelInterface);
```

Note that the functionality of this method is only provided for foreseen connections of the framework. For example it is necessary that an `AgentApplication` can exchange data with all other types of `RlApplications`. Therefore the call of `AgentApplication::AddInterface` will succeed as long as the provided `RlApplicationId::ApplicationType` matches any of the following:

- `OBSERVATION`
- `REWARD`
- `AGENT`
- `ACTION`

However, if one tries to add a `ChannelInterface` to an `ObservationApplication` that is connected to another `ObservationApplication`, the method would result in an error because the exchange between two `ObservationApplication` is deliberately excluded in the design of *ns3-defiance*.

When adding the `ChannelInterface`, the application can derive the `ApplicationType` from the `RlApplicationId`. This allows the application to properly handle the connection.

After registering the `ChannelInterface`, the `RlApplication` is ready to send.

Send

Call this method to send data over a registered `ChannelInterface`. Note that the different `RlApplications` often wrap the `RlApplication::Send` for general use cases. Therefore, refrain from using `RlApplication::Send` and use the respective appropiate method offered by each application instead. These methods often ensure additional necessary prerequisites for proper communication (e.g. registering callbacks).

Even though these wrapped methods differ in their functionality they are all called in a similar manner. There are always 3 arguments: `Ptr<OpenGymDictContainer> data, uint32_t appId, uint32_t interfaceIndex`. The first argument is required - the data that is supposed to be sent. The second argument is the `appId`. If provided, the data will only be sent to the `RlApplication` that has a matching `RlApplicationId::applicationId`. The third argument the `interfaceIndex` can be specified alongside the `RlApplicationId::applicationId`. This ensures that only a specific `ChannelInterface` is used. The index of an interface is returned by the `AddInterface` method. If the `interfaceIndex` is not set, all interfaces between the two applications are used. Similarly, if the `appId` is not set the data is sent to all registered applications of that type over all interfaces.

```
// method to send actions from agent to action app
uint interfaceIdActionApp = agentApp->AddInterface(remoteActionId,␣
↪ptrToChannelInterface);

Ptr<OpenGymDictContainer> action = /* create DictContainer */

//send to all
SendAction(action);
//send to specific application
SendAction(action, remoteActionId);
//send to specific application over specific channelInterface
SendAction(action, remoteActionId, interfaceIdActionApp);
```

AgentApplication Communication

The `AgentApplication` can communicate with applications of any `RlApplicationId::ApplicationType`:

- `OBSERVATION`
- `REWARD`
- `AGENT`
- `ACTION`

See *Add interfaces* for more information on how to set it up.

To fulfill its functionality, the `AgentApplication` is equipped with two methods - `SendAction` and `SendToAgent`. They are invoked as described in *Send*. `SendAction` only sends to applications of type `ACTION`, while `SendToAgent` only sends to applications of type `AGENT`.

RewardApplication Communication and ObservationApplication Communication

Both applications only allow communication to applications of type `AGENT`. See *Add interfaces* on how to add interfaces.

The interface of `RewardApplication` and `ObservationApplication` offers a `Send` method (through their parent class `DataCollectorApplication`) that works as described in *Send*. The passed data should be used by the agent to determine the reward or update its observation.

ActionApplication Communication

The `ActionApplication` only allows applications of type `AGENT` to be added. See *Add interfaces* on how to add interfaces.

It doesn't wrap the `Send` method because it is not supposed to send, but only receive.

2.3 Data History Container

The data history container is used for storage of data in the `AgentApplication`, specifically the latest observations received from `ObservationApplications` and latest rewards received from `RewardApplications`. When creating the history container, specify how much data it should store before deleting old data. It is possile to also specify whether the *ns-3* simulation time should be tracked with every data entry. If the usage of another history container is desired somewhere else, create a new instance of `HistoryContainer`. This can be useful for e.g. inter-agent communication.

The data container generally accepts every form of `OpenGymDictContainers`, but when the included aggregation functions like average, minimum or maximum over the last `n` entries are used, the aggregation functions will assume `OpenGymDictContainers` with `OpenGymBoxContainers` inside for them to work.

By way of the observation history container: It has an individual queue for each `ObservationApplication` that is connected to the `AgentApplication`. It also consists of a queue that contains all observations across `ObservationApplications`. The same applies to the reward history container, but with `RewardApplications`.

In order to add data to the history container, call the method `ns3::HistoryContainer::Push(ns3::Ptr<ns3::OpenGymDic` `data, uint id)`, which will add the data to the queue specified through `id`. This doesn't need to be done manually though, as the `AgentApplication` will automatically add the data to the history container when received from the `ObservationApplications` and `RewardApplications`. In order to do the same with agent messages, define a new history container and fill it accordingly in a method derived from `AgentApplication::OnRecvFromAgent`.

In order to get data from the history container, call the method `HistoryContainer::GetNewestByID(uint id, uint n)`, which will return the data from the queue specified through `id`. If necessary, use `n` to specify the number of entries to retrieve. If the newest data across all queues is needed, call the method `HistoryContainer::GetNewestOfCombinedHistory(uint n)`, which will return the latest `n` entries across all queues. Note that this might not retrieve evenly distributed numbers of entries from the queues, but rather the overall newest entries because different queues might be filled at different rates.

To get the average, minimum or maximum over the last `n` entries, call the method `HistoryContainer::AggregateNewest(uint id, uint n)`, which will return the average of the last `n` entries from the queue specified through `id`. This way, we can access the average, minimum or maximum of the last `n` entries for each key of the `OpenGymDictContainer`.

It makes sense to retreive the data from the history container in the `AgentApplication` after the `AgentApplication` has received data from the `ObservationApplications` and `RewardApplications`. Thus, the methods `void OnRecvObs(uint id) override` and `void OnRecvRew(uint id) override` are the right place to retrieve the latest observations and rewards, respectively, or to do other calculations.

For example, retrieve the newest observation from the history container with ID 0 like this:

```
void OnRecvObs(uint id) override
{
    auto obs = m_obsDataStruct.GetNewestByID(0)
                    ->data->Get("floatObs")
                    ->GetObject<OpenGymBoxContainer<float>>();
    m_observation = obs;
}
```

The following code creates a mapping of `AggregatedInfo` of the last 10 entries for each key of the `OpenGymDictContainer`, providing access to average, minimum and maximum values:

```
auto agg = m_obsDataStruct.AggregateNewest(0, 10);
auto min = agg["floatObs"].GetMin();
auto max = agg["floatObs"].GetMax();
auto avg = agg["floatObs"].GetAvg();
```

2.4 ChannelInterface

The channel interface is an abstraction for the communication between RL applications. It is used to send data as described in *Send*.

2.4.1 Overview

The channel interface is an abstracted communication channel for `RLApplications`. It provides an asynchronous, non-blocking API and uses callback mechanisms similar to the *ns-3* Socket API. It also handles serialization and deserialization of outgoing and incoming data.

The channel interface is designed to simplify communication between `RLApplications` and eliminate the overhead of creating and connecting sockets for each application. Read more about it in the *Design Documentation*. Additionally, the channel interface is extendable, allowing to create custom channel interfaces for other communication protocols.

The recommended way to connect and use channel interfaces is with the `CommunicationHelper` which handles the creation and connection process between `RLApplications`. Find more information in the Helper Documentation.

The channel interface sends and receives `OpenGymBoxContainer`. This makes it primarily suited for sharing observations, rewards, and actions between `RLApplications`, adhering to the OpenAI Gym API. However, due to the versatility of the `OpenGymBoxContainer`, it can be used to share arbitrary data between applications.

We provide two different channel interface implementations. Please note that **different** channel interface implementations are not interconnectable.

2.4.2 Usage

First, create two `ChannelInterfaces`, one for each `RLApplication` that will communicate with each other. Connect the two `ChannelInterface` objects using the `ChannelInterface::Connect` method. Afterwards, send data to the remote `RLApplication` using `ChannelInterface::Send`.

To handle received data, add a callback function to the channel interface with the `ChannelInterface::ConnectAddRecvCallback` method. This callback function will be called when new data arrives, with the deserialized data as an `OpenGymBoxContainer`.

If a callback function is no longer needed, remove it using the `ChannelInterface::ConnectRemoveRecvCallback` method. Add as many callback functions as needed. They will be called in the order they were added.

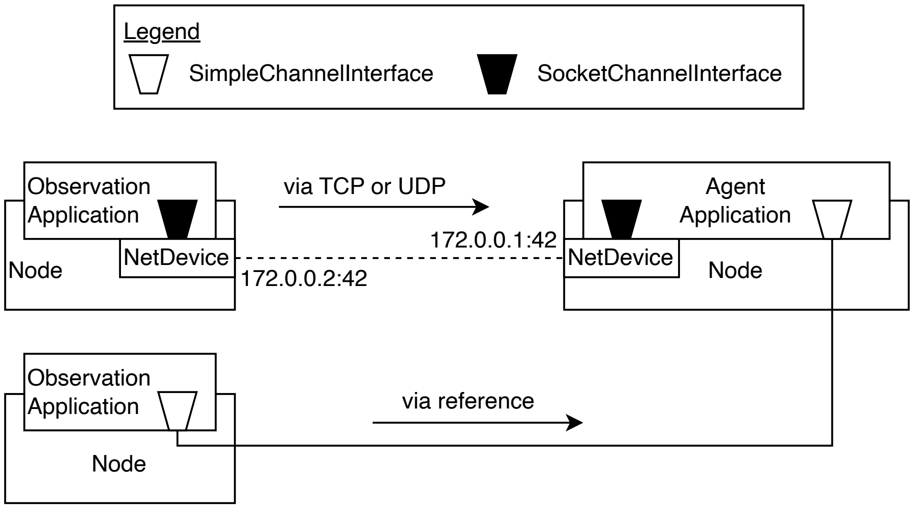


Fig. 1: Communication via `SimpleChannelInterface` and `SocketChannelInterface`

Disconnect the two `ChannelInterface` objects with the `ChannelInterface::Disconnect` method. For that, provide the specific callback function that shall be remove.

Check the connection status of the channel interface using `ChannelInterface::GetConnectionStatus`. It returns an element of the following enum:

```
enum ConnectionStatus
{
    DISCONNECTED,
    CONNECTING,
    CONNECTED,
};
```

2.4.3 SimpleChannelInterface

The `SimpleChannelInterface` simulates communication between `RLApplications` without using the underlying network simulation. It is primarily intended for debugging or simulating communication without the overhead of a full network simulation. It does not provide a realistic simulation of network communication and should not be used for performance evaluation. However, set a network delay to simulate network latency if needed.

Here is an example of how to use the `SimpleChannelInterface`:

```
// the simple interface does not need any configuration or parameters
auto interfaceSimple0 = CreateObject<SimpleChannelInterface>();
auto interfaceSimple1 = CreateObject<SimpleChannelInterface>();

// create a callback function which prints the contents of the OpenGymDictContainer
auto recvCallback = Callback<void, Ptr<OpenGymDictContainer>>(
    [](Ptr<OpenGymDictContainer> msg) { NS_LOG_INFO(msg->Get("box")); });

// add the callback function to the channel interfaces, both should just print the␣
↪received data
interfaceSimple0->AddRecvCallback(recvCallback);
interfaceSimple1->AddRecvCallback(recvCallback);
```

(continues on next page)

(continued from previous page)

```
// add a simple network delay of 0.1 seconds
interfaceSimple0->SetPropagationDelay(Seconds(0.1));
interfaceSimple1->SetPropagationDelay(Seconds(0.1));


// connect the two channel interfaces with each other in the simulation after 0.1␣
↪seconds
Simulator::Schedule(Seconds(0.1),
                    &SimpleChannelInterface::Connect,
                    interfaceSimple0,
                    interfaceSimple1);

/* helper method to creates a OpenGymDictContainer
   with a OpenGymBoxContainer named "box" with a float value */
Ptr<OpenGymDictContainer>
CreateTestMessage(float value)
{
    Ptr<OpenGymDictContainer> msg = Create<OpenGymDictContainer>();
    Ptr<OpenGymBoxContainer<float>> box = Create<OpenGymBoxContainer<float>>();
    box->AddValue(value);
    msg->Add("box", box);
    return msg;
}

// send the OpenGymDictContainer from interfaceSimple0 to interfaceSimple1 after 1␣
↪second */
Simulator::Schedule(Seconds(1),
                    &SimpleChannelInterface::Send,
                    interfaceSimple0,
                    CreateTestMessage(0));
```

This example creates two `SimpleChannelInterface` objects and connects them. After 1 second, it sends a message from one interface to the other. Due to the 0.1 second network delay, the message is printed by the receiving interface after 1.1 seconds.

2.4.4 SocketChannelInterface

The `SocketChannelInterface` uses sockets to communicate between `RLApplications`. It utilizes *ns-3* sockets under the hood and is the recommended way to simulate realistic network communication.

The network scenario and topology should ensure that the `RLApplications` can communicate with each other, for example, via the Internet or a local network. The channel interface itself does not handle the network communication; it only provides the API for communication.

If other communication methods are required, create a custom channel interface and implement it accordingly.

Here is an example of how to use the `SocketChannelInterface`:

```
// create nodes
NodeContainer nodes;
nodes.Create(2);

// create a point-to-point helper
PointToPointHelper p2p;
p2p.SetDeviceAttribute("DataRate", StringValue("5Mbps"));
p2p.SetChannelAttribute("Delay", StringValue("2ms"));
```

(continues on next page)

(continued from previous page)

```
// create devices and install them on nodes
NetDeviceContainer devices;
devices.Add(p2p.Install(nodes.Get(0), nodes.Get(1)));

// assign IP addresses
InternetStackHelper internet;
internet.Install(nodes);

Ipv4AddressHelper address;
address.SetBase("10.1.1.0", "255.255.255.0");
Ipv4InterfaceContainer interfaces = address.Assign(devices);

// get the protocol type id for the protocol we want to use
auto tcpProtocol = TcpSocketFactory::GetTypeId();

// create the interfaces with the nodes and the protocol
Ptr<SocketChannelInterface> interfaceTcp0_1 =
    CreateObject<SocketChannelInterface>(nodes.Get(0), interfaces.GetAddress(0),␣
↪tcpProtocol);
Ptr<SocketChannelInterface> interfaceTcp1_0 =
    CreateObject<SocketChannelInterface>(nodes.Get(1), interfaces.GetAddress(1),␣
↪tcpProtocol);

// create a callback function which prints the contents of the OpenGymDictContainer
auto recvCallback = Callback<void, Ptr<OpenGymDictContainer>>(
    [](Ptr<OpenGymDictContainer> msg) { NS_LOG_INFO(msg->Get("box")); });

// add the callback function to the channel interfaces, both should just print the␣
↪received data
interfaceTcp0_1->AddRecvCallback(recvCallback);
interfaceTcp1_0->AddRecvCallback(recvCallback);

// connect the two channel interfaces with each other in the simulation after 0.1␣
↪seconds
Simulator::Schedule(Seconds(0.1),
                    &SocketChannelInterface::Connect,
                    interfaceTcp0_1A,
                    interfaceTcp1_0);

/* helper method to creates a OpenGymDictContainer
   with a OpenGymBoxContainer named "box" with a float value */
Ptr<OpenGymDictContainer>
CreateTestMessage(float value)
{
    Ptr<OpenGymDictContainer> msg = Create<OpenGymDictContainer>();
    Ptr<OpenGymBoxContainer<float>> box = Create<OpenGymBoxContainer<float>>();
    box->AddValue(value);
    msg->Add("box", box);
    return msg;
}

// send the OpenGymDictContainer from interfaceTcp0_1 to interfaceTcp1_0 after 1␣
↪seconds */
Simulator::Schedule(Seconds(1),
                    &SocketChannelInterface::Send,
                    interfaceUdp0_1,
```

(continues on next page)

(continued from previous page)

```
                    CreateTestMessage(1));
```

This example creates two `SocketChannelInterface` and connects them. After 1 second, it sends a message from one interface to the other and prints the received message after approximately 1.02 seconds (because of the 20ms network delay).

2.4.5 Custom Channel Interface

If necessary, implement and use a custom channel interface to use alternative communication protocols or methods for communication between `RLApplications`.

To create a custom channel interface, inherit from the abstract base class `ChannelInterface` and implement its corresponding methods.

2.5 Helper Classes

2.5.1 RlApplicationHelper

As previously introduced, the DEFIANCE framework is mainly structured around user specific `RlApplications`. They are derived from their specific base classes (e.g. `AgentApplication`) and communicate relevant information with one another during the simulation.

To simplify the creation of their instances, the `RlApplicationHelper` class is provided. As with the typical helper classes already present in *ns-3*, it makes the creation of the applications more intuitive.

The following example demonstrates how the `RlApplicationHelper` can be used.

```
RlApplicationHelper helper(TypeId::LookupByName("ns3::MyObservationApp"));

// the helper allows to set attributes for the applications
// this is persistent for all the applications that will be created afterwards
helper.SetAttribute("StartTime", TimeValue(Seconds(0)));
helper.SetAttribute("StopTime", TimeValue(Seconds(10)));

RlApplicationContainer observationApps = helper.Install(myNodes1);

helper.SetTypeId("ns3::MyRewardApp");
RlApplicationContainer rewardApps = helper.Install(myNodes2);

helper.SetTypeId("ns3::MyActionApp");
RlApplicationContainer actionApps = helper.Install(myNodes3);

helper.SetTypeId("ns3::MyAgentApp");
RlApplicationContainer agentApps = helper.Install(myNodes4);
```

This example shows the main features of the `RlApplicationHelper`. First of all, it wraps the created application instances in an `RlApplicationContainer`. This container can be used like the standard *ns-3* `ApplicationContainer` to access or iterate over the applications but does not require to cast the applications each time that DEFIANCE-specific functionality is required. Secondly, the helper allows to set attributes for the applications. This enables work with the `TypeId` system, which makes it easy to set default arguments and to work with command line arguments. In the example above, the helper is used to create different types of applications but sets the same start and stop time for all of them.

Note: The `RlApplicationHelper` is not limited to the applications that are provided by the DEFIANCE framework. It can be used with any application that is derived from the `RlApplication` class.

2.5.2 CommunicationHelper

The natural extension to the `RlApplicationHelper` is the `CommunicationHelper`. It can work with `RlApplicationContainers` to create communication channels between the applications and configure them accordingly. The CommunicationHelper simplifies this procedure and reduces the risks of bugs.

First, create an instance of `CommunicationHelper` and set the different applications:

```
1  CommunicationHelper commHelper = CommunicationHelper();
2
3  commHelper.SetObservationApps(observationApps);
4  commHelper.SetAgentApps(agentApps);
5  commHelper.SetRewardApps(rewardApps);
6  commHelper.SetActionApps(actionApps);
7  commHelper.SetIds();
```

The different `Set` methods expect an object of type `RlApplicationContainer`. See chapter ApplicationHelper for more information on how to create one. After the helper received all `RlApplicationContainers`, the IDs of these applications need to be assigned (line 7). The IDs are used to identify the instances of `RlApplication` and are required for the next step.

Once the IDs are assigned, the actual connection can be configured. This can be done by passing a vector of type `CommunicationPair` to the `CommunicationHelper`. To create an instance of `CommunicationPair`, the IDs of the two `RlApplications` and a `CommunicationAttributes` object have to be provided. The `CommunicationAttributes` object describes the type of connection. If no argument is passed, a `SimpleChannelInterface` is created. To create a socket connection via TCP or UDP, a `SocketCommunicationAttributes` object with `TypeId protocol` set accordingly can be passed as `CommunicationAttributes`. The following code is a simple example that creates `CommunicationPairs` of different types.

```
1  // UDP
2  CommunicationPair actionCommPair = {
3      actionApps.GetId(0),
4      agentApps.GetId(0),
5      SocketCommunicationAttributes{"7.0.0.2", "1.0.0.2", UdpSocketFactory::GetTypeId()}
   ↪};
6
7  //TCP
8  CommunicationPair observationCommPair = {
9          observationApps.GetId(0),
10         agentApps.GetId(0),
11         SocketCommunicationAttributes{"7.0.0.2", "1.0.0.2",␣
   ↪TcpSocketFactory::GetTypeId()}};
12
13 //SIMPLE
14 CommunicationPair actionCommPair = {actionApps.GetId(0),
15                                     agentApps.GetId(0),
16                                     {}};
```

The method `GetId(i)` allows to retrieve the `RlApplicationId` by passing the index `i` to the `RlApplicationContainer` (as used in e.g. line 3–4). When creating `SocketCommunicationAttributes`, the passed IP addresses have to match the addresses of the node the application is installed on.

Once these `CommunicationPairs` are created, collect them in a vector and pass it to `Communication-Helper::AddCommunication` as a parameter. Finally, the configuration can be finished by calling `Configure` on the `CommunicationHelper`. Now all channel interfaces are created accordingly, ready for sending and receiving data. An explanation of the `Configure` method can be found in section *Helper* of the design documentation.

CHAPTER

THREE

REFERENCES

BIBLIOGRAPHY

[Pettingzoo] "PettingZoo Documentation"

[PettingzooWrapper] "MultiAgentEnv wrapper for PettingZoo APIs"

[AEC] Terry, Jordan, et al. "Pettingzoo: Gym for multi-agent reinforcement learning." Advances in Neural Information Processing Systems 34 (2021): 15032-15043.

[Gymnasium] "Gymnasium Documentation"

[RLLib] "RLLib Documentation"

[ns3-ai] Yin, Hao, et al. "ns3-ai: Fostering artificial intelligence algorithms for networking research." Proceedings of the 2020 Workshop on ns-3. 2020.

[MultiAgentEnv] "MultiAgentEnv API"