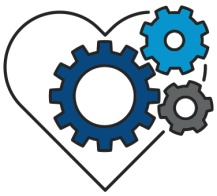
AML-IP Documentation

Release ..

eProsima

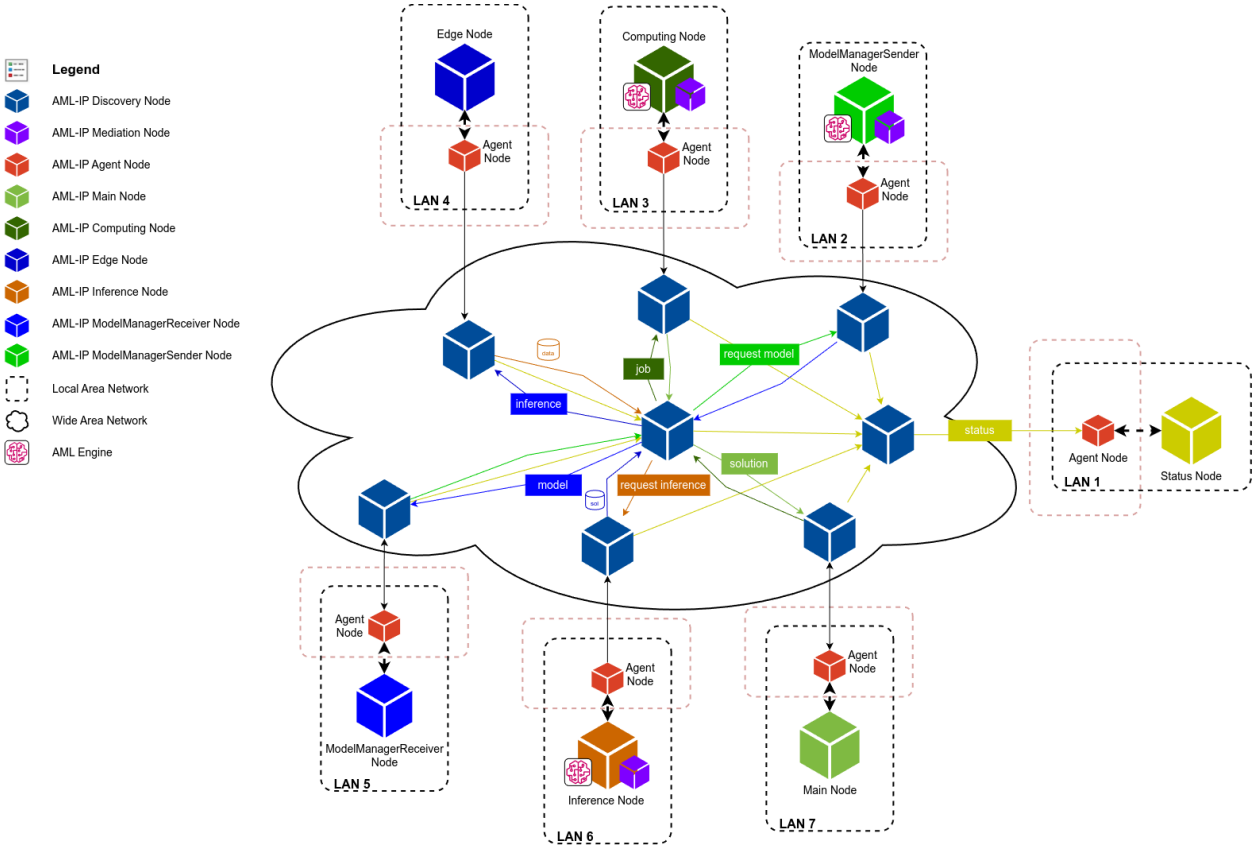
Apr 25, 2024

INTRODUCTION



eProsima AML-IP is a communications framework in charge of data exchange between Algebraic Machine Learning (AML) nodes through local or remote networks. It is designed to allow non-experts users to create and manage a cluster of AML nodes to exploit the distributed and concurrent learning capabilities of AML. Thus, AML-IP is a communication framework that makes the transport protocols abstracted from the user, creating a platform that communicates each node without requiring the user to be concerned about communication issues. It also allows the creation of complex distributed networks with one or multiple users working on the same problem.

This framework is developed as part of the ALMA project, which has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 952091. The aim of the EU-funded ALMA project is to leverage AML properties to develop a new generation of interactive, human-centric machine learning systems. These systems are expected to reduce bias and prevent discrimination, remember what they know when they are taught something new, facilitate trust and reliability and integrate complex ethical constraints into human-artificial intelligence systems. Furthermore, they are expected to promote distributed, collaborative learning.



Following are the main scenarios that the current *AML-IP* supports:

- *Monitor Network State Scenario*: Analyze the state of a network remotely.
- *Workload Distribution Scenario*: Distribute the machine learning training phase to multiple nodes to parallelize heavy computation.
- *Collaborative Learning Scenario*: Share models between nodes without having to share the private dataset with which the model was trained.
- *Distributed Inference Scenario*: Distribute large amounts of data to multiple nodes to perform inference in parallel.

Check section *Project Overview* to have a further explanation of the concepts and use cases of this project.

CHAPTER

ONE

CONTACTS AND COMMERCIAL SUPPORT

Find more about us at eProsima's webpage.

Support available at:

- Email: support@eprosima.com
- Phone: +34 91 804 34 48

CHAPTER

TWO

CONTRIBUTING TO THE DOCUMENTATION

AML-IP Documentation is an open source project, and as such all contributions, both in the form of feedback and content generation, are most welcomed. To make such contributions, please refer to the Contribution Guidelines hosted in our GitHub repository.

CHAPTER

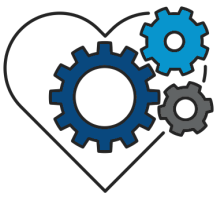
THREE

STRUCTURE OF THE DOCUMENTATION

This documentation is organized into the sections below.

- *Installation Manual*
- *Getting Started*
- *Demo Examples*
- *User Manual*
- *Developer Manual*
- *Release Notes*

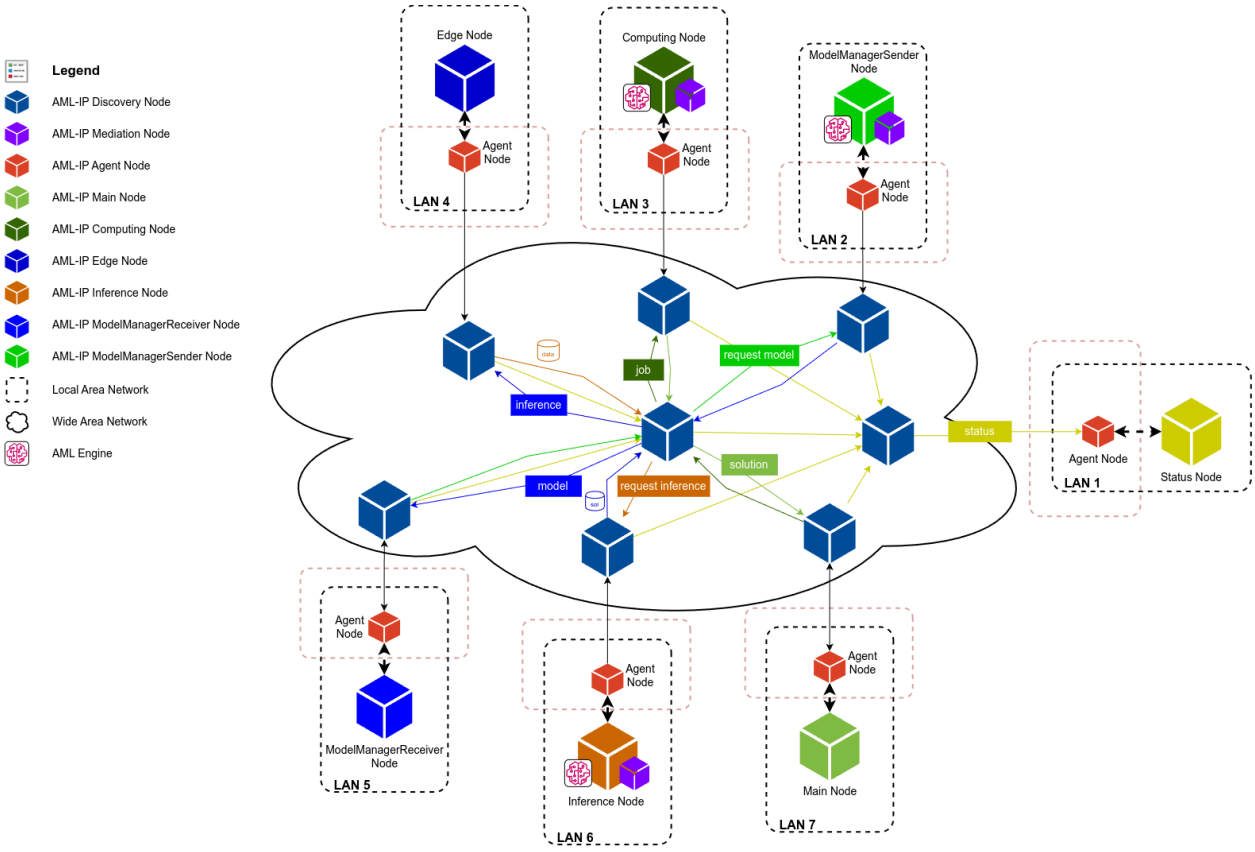
This project (ALMA: Human Centric Algebraic Machine Learning) has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 952091.



eProsima AML-IP is a communications framework in charge of data exchange between Algebraic Machine Learning (AML) nodes through local or remote networks. It is designed to allow non-experts users to create and manage a cluster of AML nodes to exploit the distributed and concurrent learning capabilities of AML. Thus, AML-IP is a communication framework that makes the transport protocols abstracted from the user, creating a platform that communicates each node without requiring the user to be concerned about communication issues. It also allows the creation of complex distributed networks with one or multiple users working on the same problem.

This framework is developed as part of the ALMA project, which has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 952091. The aim of the EU-funded ALMA

project is to leverage AML properties to develop a new generation of interactive, human-centric machine learning systems. These systems are expected to reduce bias and prevent discrimination, remember what they know when they are taught something new, facilitate trust and reliability and integrate complex ethical constraints into human-artificial intelligence systems. Furthermore, they are expected to promote distributed, collaborative learning.



3.1 Overview

Following are the main scenarios that the current *AML-IP* supports:

- *Monitor Network State Scenario*: Analyze the state of a network remotely.
- *Workload Distribution Scenario*: Distribute the machine learning training phase to multiple nodes to parallelize heavy computation.
- *Collaborative Learning Scenario*: Share models between nodes without having to share the private dataset with which the model was trained.
- *Distributed Inference Scenario*: Distribute large amounts of data to multiple nodes to perform inference in parallel.

Check section *Project Overview* to have a further explanation of the concepts and use cases of this project.

3.2 Contacts and Commercial support

Find more about us at eProsima's webpage.

Support available at:

- Email: support@eprosima.com
- Phone: +34 91 804 34 48

3.3 Contributing to the documentation

AML-IP Documentation is an open source project, and as such all contributions, both in the form of feedback and content generation, are most welcomed. To make such contributions, please refer to the Contribution Guidelines hosted in our GitHub repository.

3.4 Structure of the documentation

This documentation is organized into the sections below.

- *Installation Manual*
- *Getting Started*
- *Demo Examples*
- *User Manual*
- *Developer Manual*
- *Release Notes*

This project (ALMA: Human Centric Algebraic Machine Learning) has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 952091.

3.5 AML-IP on Windows

Warning: The current version of *AML-IP* does not have installers for Windows platforms. Please refer to the *Windows installation from sources* section to learn how to build *AML-IP* on Windows from sources.

3.6 AML-IP on Linux

Warning: The current version of *AML-IP* does not have installers for Linux platforms. Please refer to the *Linux installation from sources* section to learn how to build *AML-IP* on Linux from sources.

3.7 Docker image

A pre-compiled image of the *AML-IP* is not available at this stage. However, please find instructions on how to create your own Docker image here.

3.8 Project Overview

eProsima AML-IP is a communications framework in charge of data exchange between *AML* nodes through local or remote networks. It is designed to allow non-experts users to create and manage a cluster of AML nodes to exploit the distributed and concurrent learning capabilities of AML. Thus, AML-IP is a communication framework that abstracts the transport protocols from the user, creating a platform that communicates each node without requiring the user to be concerned about communication issues. It also allows the creation of complex distributed networks with one or multiple users working on the same problem.

3.8.1 AML

AML is a cutting edge *ML* technology based on algebraic representations of data. Unlike statistical learning, *AML* algorithms are robust regarding the statistical properties of the data and are parameter-free. This makes *AML* a great candidate in the future of ML, as it is far less sensitive to statistical characteristics of the training data, and can integrate unstructured and complex abstract information apart from the training data.

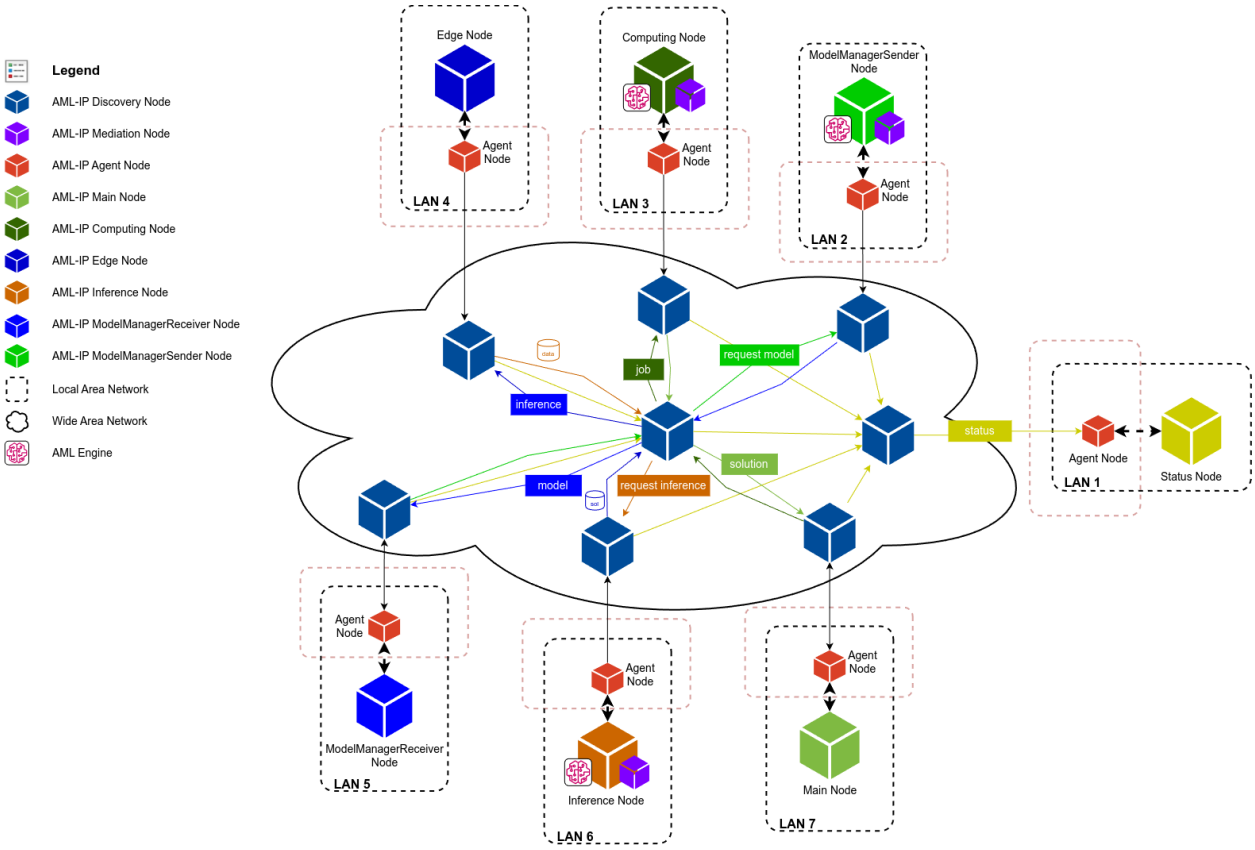
AML algorithm has several characteristics that makes it a great player in distributed learning. First, *AML* can be trained in parallel from different remote machines, and can merge the training information without losing information. It can also be shared and merged with other already trained models and share their learnt information without revealing the training data-set.

3.8.2 AML-IP

eProsima AML-IP is a framework based on different libraries and graphical and non-graphical tools that allow to create a network of nodes focused no different tasks of the *AML* environment. Every running part of the *AML-IP* is considered a **Node**. This is an **independent** and **distributed** software that could perform a specific **action**.

- *Independent* means that it is auto-sufficient and does not require the presence of any other node.
- *Distributed* means that can communicate with different nodes in the network, interacting and solving tasks collaboratively.
- *Action* is every part of the *AML* or any satellite action required in order to perform the correct execution of the algorithm or to support or facilitate the communication and managing of the different nodes.

These nodes are separated in different scenarios, that are explained more in detail in the *following section*.



3.8.3 Usage

AML-IP is a complex framework composed of different tools that run independently and out-of-the-box. But it also features some libraries that allow to instantiate *AML-IP* entities or *Nodes* whose behavior and functionality must be specified by the user. These libraries are presented in 2 main programming languages:

C++

This is the main programming language in *AML-IP*. C++ has been chosen because it is a very versatile and complete language that allows to easily implement complex concepts maintaining high performance. Also *Fast DDS* is mainly built in C++ and using the same programming language allows to easily interact without losing performance with the middleware layer.

There is a public *API* found in `AML-IP/amlip_cpp/include` with all the installed headers that can be used from the user side. The API, implementation and testing of this part of the code can be found mainly under sub-package `amlip_cpp`.

Python

This is the programming language though to be used by a final user. Python has been chosen as it is easier to work with state-of-the-art *ML* projects.

Nodes and classes that the user needs to instantiate in order to implement their own code are parsed from C++ by using SWIG tool, giving the user a Python API. The API, implementation and testing of this part of the code can be found mainly under sub-package `amlip_py`.

3.8.4 Architecture and Infrastructure

AML-IP is a software project based on different programming languages. It is a **public open-source** project focused to be used by the *ML* and scientific community. The whole project is hosted on a github repository, and can be found in the following url: AML-IP Github repository. The code project is divided in sub-packages that can be built, installed and tested independently.

AML-IP is a software project that does not rely on any specific hardware or Operating System, and does not require any physical infrastructure. The storage and CI is hosted by github.

3.8.5 Testing and CI

AML-IP project is exhaustively tested with unit, integration, manual and blackbox tests that can be found within the source code. As an open-source project, every person can and is welcome to contribute to the project by implementing, fixing, suggesting or issuing. Every new contribution to the project must be peer reviewed by a project member before being accepted and merged, and must fulfill all the tests and required *CI*.

3.9 Collaborative Learning

3.9.1 Background

This demo shows a *Collaborative Learning Scenario* and the *AML-IP* nodes involved: *Model Manager Receiver Node* and *Model Manager Sender Node*. With these 2 nodes implemented, the user can deploy as many nodes of each kind as desired and check the behavior of a simulated *AML-IP* network running. They are implemented in Python to prove the communication between the 2 implementations.

The purpose of the demo is to show how a *Sender* and a *Receiver* node can communicate. The *Receiver* node awaits model statistics from the *Sender*. Since the *Sender* doesn't have a real *AML Engine*, it sends the model statistics as a string. Upon receiving the statistics, the *Receiver* sends a model request, also as a string since it doesn't have an *AML Engine*. Then, the *Sender* converts the received model request to uppercase and sends it back as a model reply.

3.9.2 Prerequisites

Before running this demo, ensure that `AML-IP` is correctly installed using one of the following installation methods:

- *AML-IP on Linux*
- *AML-IP on Windows*
- *Docker image*

3.9.3 Building the demo

If the demo package is not compiled, please refer to *Build demos* or run the command below.

```
colcon build --packages-up-to amlip_demo_nodes
```

Once AML-IP packages are installed and built, import the libraries using the following command.

```
source install/setup.bash
```

3.9.4 Explaining the demo

In this section, we will delve into the details of the demo and how it works.

Model Manager Receiver Node

This is the Python code for the *Model Manager Receiver Node* application. It does not use real *AML Models*, but strings. It is implemented in Python using `amlip_py` API.

This code can be found here.

The next block includes the Python header files that allow the use of the AML-IP Python API.

```
from amlip_py.node.ModelManagerReceiverNode import ModelManagerReceiverNode,␣
↪ModelListener
from amlip_py.types.AmlipIdDataType import AmlipIdDataType
from amlip_py.types.ModelReplyDataType import ModelReplyDataType
from amlip_py.types.ModelRequestDataType import ModelRequestDataType
from amlip_py.types.ModelStatisticsDataType import ModelStatisticsDataType
```

Let's continue explaining the global variables.

`DOMAIN_ID` variable isolates the execution within a specific domain. Nodes with the same domain ID can communicate with each other.

```
DOMAIN_ID = 166
```

`waiter` is a `WaitHandler` that waits on a boolean value. Whenever this value is `True`, threads awake. Whenever it is `False`, threads wait.

```
waiter = BooleanWaitHandler(True, False)
```

The `CustomModelListener` class listens to *Model Statistics Data Type* and *Model Reply Data Type* messages received from a *Model Manager Sender Node*. This class is supposed to be implemented by the user in order to process the messages received from other nodes in the network.

```
class CustomModelListener(ModelListener):

    def statistics_received(
            self,
            statistics: ModelStatisticsDataType):

        print(f'Statistics received: {statistics.to_string()}')

        # Store the server id of the statistics
        self.server_id = statistics.server_id()

        waiter.open()

    def model_received(
            self,
            model: ModelReplyDataType) -> bool:

        print(f'Model reply received from server\n'
              f' solution: {model.to_string()}')

        return True
```

The *main* function orchestrates the execution of the Model Manager Receiver node. It creates an instance of the *ModelManagerReceiverNode* and starts its execution with the specified listener.

```
def main():
    """Execute main routine."""

    # Create request
    data = ModelRequestDataType('MobileNet V1')

    id = AmlipIdDataType('ModelManagerReceiver')
    id.set_id([15, 25, 35, 45])

    # Create node
    print('Starting Manual Test Model Manager Receiver Node Py execution. Creating Node..
↪.')
    model_receiver_node = ModelManagerReceiverNode(
        id=id,
```

(continues on next page)

(continued from previous page)

```
        data=data,
        domain=DOMAIN_ID)

    print(f'Node created: {model_receiver_node.get_id()}. '
          'Already processing models.')

    model_receiver_node.start(
        listener=CustomModelListener())
```

After starting the node, it waits for statistics to arrive from the *Model Manager Sender Node*.

```
    # Wait statistics
    waiter.wait()
```

Then, it requests a model from the *Model Manager Sender Node* using the received server ID.

```
    # Request model
    model_receiver_node.request_model(model_receiver_node.listener_.server_id)
```

Finally, the node stops.

```
    model_receiver_node.stop()
```

Model Manager Sender Node

This is the Python code for the *Model Manager Sender Node* application. It does not use real *AML Models*, but strings. It does not have a real *AML Engine* but instead the calculation is an *upper-case* conversion of the string received. It is implemented in Python using `amlip_py` API.

This code can be found here.

The following block includes the Python header files necessary for using the AML-IP Python API.

```
from amlip_py.node.ModelManagerSenderNode import ModelManagerSenderNode, ModelReplier
from amlip_py.types.AmlipIdDataType import AmlipIdDataType
from amlip_py.types.ModelReplyDataType import ModelReplyDataType
from amlip_py.types.ModelRequestDataType import ModelRequestDataType
```

Let's continue explaining the global variables.

`DOMAIN_ID` isolates the execution within a specific domain. Nodes with the same domain ID can communicate with each other.

```
DOMAIN_ID = 166
```

`waiter` is a `WaitHandler` that waits on a boolean value. Whenever this value is `True`, threads awake. Whenever it is `False`, threads wait.

```
waiter = BooleanWaitHandler(True, False)
```

The `CustomModelReplier` class listens to *Model Request Data Type* request messages received from a *Model Manager Receiver Node*. This class is supposed to be implemented by the user in order to process the messages.

```
class CustomModelReplier(ModelReplier):

    def fetch_model(
            self,
            request: ModelRequestDataType) -> ModelReplyDataType:

        reply = ModelReplyDataType(request.to_string().upper())

        print(f'Model request received from client\n'
              f' request: {request.to_string()}\n'
              f' reply: {reply.to_string()}')

        waiter.open()

        return reply
```

The *main* function orchestrates the execution of the Model Manager Sender node. It creates an instance of *ModelManagerSenderNode*.

```
def main():
    """Execute main routine."""

    id = AmlipIdDataType('ModelManagerSender')
    id.set_id([10, 20, 30, 40])

    # Create node
    print('Starting Manual Test Model Manager Sender Node Py execution. Creating Node...
↪')
    model_sender_node = ModelManagerSenderNode(
        id=id,
        domain=DOMAIN_ID)
```

After starting the node, it publishes statistics using the `publish_statistics()` function, which fills a *Model Statistics Data Type* and publishes it.

```
    model_sender_node.publish_statistics(
        'ModelManagerSenderStatistics',
        'hello world')
```

Then we start the node execution, passing the previously defined `CustomModelReplier()` class, which is responsible for managing the request received.

```
    model_sender_node.start(
        listener=CustomModelReplier())
```

Waits for the response model to be sent to the *Model Manager Receiver Node*.

```
    # Wait for the solution to be sent
    waiter.wait()
```

Finally, it stops and closes the node.

```
    model_sender_node.stop()
```

3.9.5 Running the demo

This demo runs the implemented nodes in `amlip_demo_nodes/amlip_collaborative_learning_demo`.

Run Model Manager Receiver Node

Run the following command:

```
# Source colcon installation
source install/setup.bash

# To execute Model Manager Receiver Node
cd ~/AML-IP-ws/src/AML-IP/amlip_demo_nodes/amlip_collaborative_learning_demo/amlip_
↪collaborative_learning_demo
python3 model_receiver_custom.py
```

The expected output is the following:

```
Starting Manual Test Model Manager Receiver Node Py execution. Creating Node...
Node created: ModelManagerReceiver.0f.19.23.2d. Already processing models.
Model reply received from server
solution: MOBILENET V1
Finishing Manual Test Model Manager Receiver Node Py execution.
```

Run Model Manager Sender Node

Run the following command to answer before closing:

```
# Source colcon installation
source install/setup.bash

# To execute Model Manager Sender Node
cd ~/AML-IP-ws/src/AML-IP/amlip_demo_nodes/amlip_collaborative_learning_demo/amlip_
↪collaborative_learning_demo
python3 model_sender_custom.py
```

This execution expects an output similar to the one shown below:

```
Starting Manual Test Model Manager Sender Node Py execution. Creating Node...
Node created: ModelManagerSender.0a.14.1e.28. Already processing models.
Model request received from client
model: MobileNet V1
solution: MOBILENET V1
Finishing Manual Test Model Manager Sender Node Py execution.
```

3.10 TensorFlow Inference

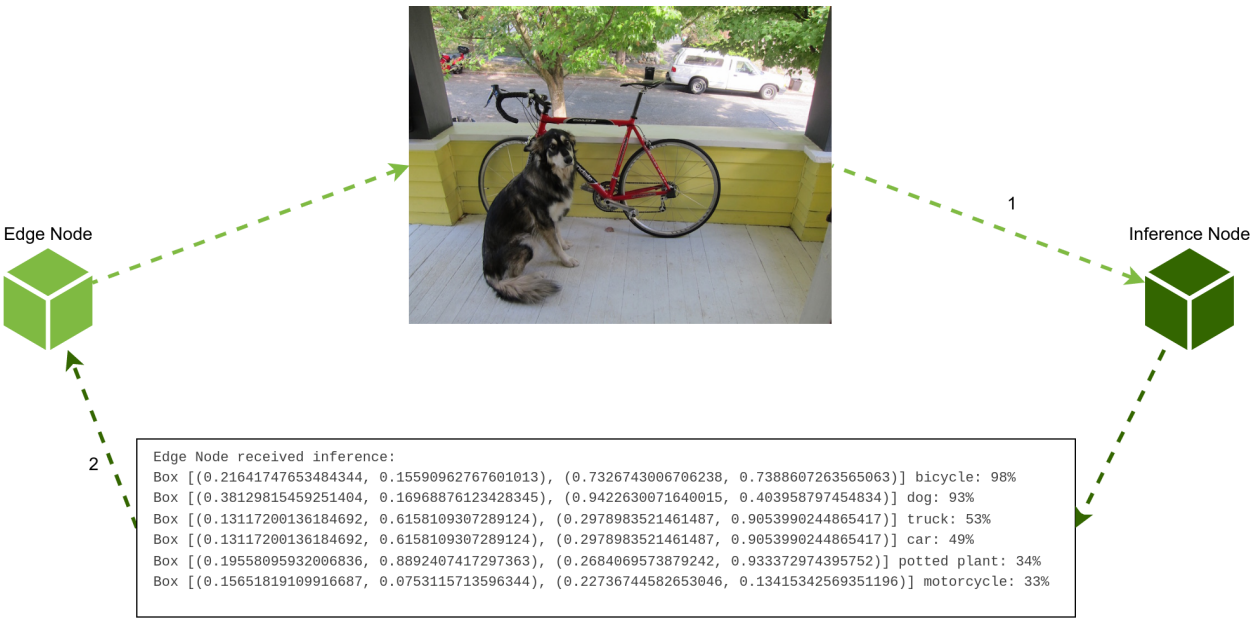
3.10.1 Background

Inference refers to the process of using a trained model to make predictions or draw conclusions based on input data. It involves applying the learned knowledge and statistical relationships encoded in the model to new, unseen data. The inference of an image involves passing the image through a trained AI model to obtain a classification based on the learned knowledge and patterns within the model.

This demo shows how to implement 2 types of nodes, *Inference Node* and *Edge Node*, to perform TensorFlow inference on a given image. With these 2 nodes implemented, the user can deploy as many nodes of each kind as desired and check the behavior of a simulated *AML-IP* network running.

The demo that is presented here follows the schema of the figure below:

- TensorFlow is an end-to-end machine learning platform with pre-trained models.
- Edge Node simulates an *Edge Node*. It is implemented in Python using `amlip_py` API.
- Inference Node simulates an *Inference Node*. It is implemented in Python using `amlip_py` API.



3.10.2 Prerequisites

First of all, check that `amlip_tensorflow_inference_demo` sub-package is correctly installed. If it is not, please refer to *Build demos*.

The demo requires the following tools to be installed in the system:

```
sudo apt install -y  swig alsa-utils libopencv-dev
pip3 install -U pyttsx3 opencv-python
curl https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh -o Miniconda3-
↪latest-Linux-x86_64.sh
bash Miniconda3-latest-Linux-x86_64.sh
# For changes to take effect, close and re-open your current shell.
conda create --name tf python=3.9
conda install -c conda-forge cudatoolkit=11.8.0
mkdir -p $CONDA_PREFIX/etc/conda/activate.d
echo 'CUDNN_PATH=$(dirname $(python3 -c "import nvidia.cudnn;print(nvidia.cudnn.__file__)
↪"))' >> $CONDA_PREFIX/etc/conda/activate.d/env_vars.sh
echo 'export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$CONDA_PREFIX/lib/:$CUDNN_PATH/lib' >>
↪$CONDA_PREFIX/etc/conda/activate.d/env_vars.sh
source $CONDA_PREFIX/etc/conda/activate.d/env_vars.sh
```

Ensure that you have TensorFlow and TensorFlow Hub installed in your Python environment before proceeding. You can install them using pip by executing the following commands:

```
pip3 install tensorflow tensorflow-hub tensorflow-object-detection-api nvidia-cudnn-
↪cu11==8.6.0.163 protobuf==3.20.*
```

Additionally, it is required to obtain the TensorFlow model from TensorFlow Hub, follow the steps below:

```
cd ~/AML-IP-ws/src/AML-IP/amlip_demo_nodes/amlip_tensorflow_inference_demo/resource/
↪tensorflow/models/
wget -O centernet_hourglass_512x512_kpts_1.tar.gz https://tfhub.dev/tensorflow/centernet/
↪hourglass_512x512_kpts/1?tf-hub-format=compressed
```

(continues on next page)

(continued from previous page)

```
mkdir centernet_hourglass_512x512_kpts_1
tar -xvf centernet_hourglass_512x512_kpts_1.tar.gz -C ./centernet_hourglass_512x512_kpts_
↪1
```

3.10.3 Building the demo

To build the demo, build the packages with Colcon:

```
colcon build --packages-up-to amlip_demo_nodes
```

Once AML-IP packages are installed and built, import the libraries using the following command.

```
source install/setup.bash
```

3.10.4 Explaining the demo

In this section, we will explore and explain the demo in detail.

Edge Node

Edge Node serves as the entity responsible for sending the data to be inferred to the Inference Node. The Edge Node is typically located at the edge of a network or closer to the data source, such as a sensor or a device generating the data.

This is the Python code for the Edge Node application. This code can be found here.

The next block includes the Python header files that allow the use of the AML-IP Python API.

```
from amlip_py.node.AsyncEdgeNode import AsyncEdgeNode, InferenceListenerLambda
from amlip_py.types.InferenceDataType import InferenceDataType
```

Let's continue explaining the global variables. The `waiter` allows the node to wait for the inference. `DOMAIN_ID` allows the execution to be isolated because only DomainParticipants with the same Domain Id would be able to communicate to each other.

```
# Variable to wait to the inference
waiter = BooleanWaitHandler(True, False)

# Domain ID
DOMAIN_ID = 166
```

The definition of the `inference_received` function prints the details of the received inference.

```
def inference_received(
        inference,
        task_id,
        server_id):
    print(f'Edge Node received inference from {server_id}')
    print(f'Edge Node received inference {inference.to_string()}')
    waiter.open()
```

We define the `main` function.

```
def main():
```

First, we create an instance of `AsyncEdgeNode`. The first thing the constructor gets is the given name. Then a listener, which is an `InferenceListenerLambda` object is created with the function `inference_received` declared above. This function is called each we receive an inference. And also we specified the domain equal to the DOMAIN_ID variable.

```
    node = AsyncEdgeNode(
        'AMLAsyncEdgeNode',
        listener=InferenceListenerLambda(inference_received),
        domain=DOMAIN_ID)
```

The next code block loads the image using `cv2.imread` based on the specified image_path. It converts the size information and the image into bytes and combines the two to send them to the Inference node.

```
    current_path = os.path.abspath(__file__)
    image_path = current_path.split('amlip_tensorflow_inference_demo', -1)[0]\
        + 'amlip_tensorflow_inference_demo/resource/tensorflow/models/research\
/object_detection/test_images/dog.jpg'
    img = cv2.imread(image_path)
    width = img.shape[1]
    height = img.shape[0]

    # Convert size to bytes
    str_size = str(width) + ' ' + str(height) + ' | '
    bytes_size = bytes(str_size, 'utf-8')
    # Convert image to bytes
    img_bytes = base64.b64encode(img)
    # Size + images
    img_size_bytes = bytes_size + img_bytes
```

After that, the `request_inference` method is called to request the inference of the image.

```
    task_id = node.request_inference(InferenceDataType(img_size_bytes))
```

Finally, the program waits for the inference solution using `waiter.wait`.

```
    waiter.wait()
```

Once the solution is received, the execution finish.

Inference Node

The Inference Node is responsible for making the inferences or predictions on the data it receives using a TensorFlow model. The Inference Node is typically a server or a computing resource equipped with high-performance hardware optimized for executing machine learning models efficiently.

This is the Python code for the Inference Node application. This code can be found here.

The next block includes the Python header files that allow the use of the AML-IP Python API.

```
from amlip_py.node.AsyncInferenceNode import AsyncInferenceNode, InferenceReplierLambda
from amlip_py.types.InferenceSolutionDataType import InferenceSolutionDataType
```

Let's continue explaining the global variables. `DOMAIN_ID` allows the execution to be isolated because only DomainParticipants with the same Domain Id would be able to communicate to each other. `tolerance` sets a limit to ignore detections with a probability less than the tolerance.

```
# Domain ID
DOMAIN_ID = 166

# Not take into account detections with less probability than tolerance
tolerance = 25
```

It loads the model from TensorFlow based on the specified path.

```
current_path = os.path.abspath(__file__)
# Initialise model
path = current_path.split('amlip_tensorflow_inference_demo', -1)[0]\
                        + 'amlip_tensorflow_inference_demo/resource/\
tensorflow/models/centernet_hourglass_512x512_kpts_1'
dataset = current_path.split('amlip_tensorflow_inference_demo', -1)[0]\
                            + 'amlip_tensorflow_inference_demo/resource/\
tensorflow/models/research/object_detection/data/mscoco_label_map.pbtxt'

print('Model Handle at TensorFlow Hub: {}'.format(path))
print('loading model...')
hub_model = hub.load(path)
```

The `process_inference` function is responsible for computing the inference when data is received. Inference is performed using the input data and the loaded model. Note that detected objects are filtered based on the specified tolerance.

```
def process_inference(
        inference,
        task_id,
        client_id):
    # Size | Image
    height, width = (inference.to_string().split(' | ', 1)[0]).split()
    image_str = inference.to_string().split(' | ', 1)[1]
    # Convert string to bytes
    img_bytes = base64.b64decode(image_str)
    # Convert bytes to image
    image = np.frombuffer((img_bytes), dtype=np.uint8).reshape((int(width), int(height),␣
↪3))
    string_inference = ''
    image_np = np.array(image).reshape((1, int(width), int(height), 3))
    results = hub_model(image_np)
    result = {key: value.numpy() for key, value in results.items()}
    category_index = label_map_util.create_category_index_from_labelmap(dataset,
                                                                         use_display_
↪name=True)
    classes = (result['detection_classes'][0]).astype(int)
    scores = result['detection_scores'][0]
    for i in range(result['detection_boxes'][0].shape[0]):
        if (round(100*scores[i]) > tolerance):
            boxes = result['detection_boxes'][0]
            box = tuple(boxes[i].tolist())
```

(continues on next page)

(continued from previous page)

```
            ymin, xmin, ymax, xmax = box
            string_inference = string_inference + \
                'Box [({}, {}), ({}, {})] {}: {}% \n' \
                .format(xmin, ymin, xmax, ymax, category_index[classes[i]]['name'],
                        round(100*scores[i]))
    print('Inference ready!')
    print('sending inference: ' + string_inference)
    return InferenceSolutionDataType(string_inference)
```

We define the `main` function.

```
def main():
```

We create an instance of `AsyncInferenceNode`. The first thing the constructor gets is the name `AMLInferenceNode`. Then the listener which is an `InferenceReplierLambda(process_inference)`. This means calling the `process_inference` function to perform the inference requests. And also we specified the domain equal to the DOMAIN_ID variable.

```
    node = AsyncInferenceNode(
        'AMLInferenceNode',
        listener=InferenceReplierLambda(process_inference),
        domain=DOMAIN_ID)
```

This starts the inference node. It will start listening for incoming inference requests and call the `process_inference` function to handle them.

```
    node.run()
```

Finally, waits for a SIGINT signal `Ctrl+C` to stop the node and close it.

```
    def handler(signum, frame):
        pass
    signal.signal(signal.SIGINT, handler)
    signal.pause()

    node.stop()
```

3.10.5 Run demo

This demo explains the implemented nodes in *amlip_demo_nodes/amlip_tensorflow_inference_demo*.

Run Edge Node

In the first terminal, run the Edge Node with the following command:

```
# Source colcon installation
source install/setup.bash

# To execute Edge Node to send an image to inferred
```

(continues on next page)

(continued from previous page)

```
cd ~/AML-IP-ws/src/amlip/amlip_demo_nodes/amlip_tensorflow_inference_demo/amlip_
↪tensorflow_inference_demo
python3 edge_node_async.py
```

Take into account that this node will wait until there is an *Inference Node* running and available in the same *LAN* in order to process the inference. The expected output is the following:

```
Edge Node AMLEdgeNode.fb.d4.38.13 ready.
Edge Node AMLEdgeNode.fb.d4.38.13 sending data.

Edge Node received inference from AMLInferenceNode.b8.34.4d.a3
Edge Node received inference:
Box [(0.15590962767601013, 0.21641747653484344), (0.7388607263565063, 0.
↪7326743006706238)] bicycle: 97%
Box [(0.16968876123428345, 0.38129815459251404), (0.403958797454834, 0.
↪9422630071640015)] dog: 92%
Box [(0.6158109307289124, 0.13117200136184692), (0.9053990244865417, 0.
↪2978983521461487)] truck: 53%
Box [(0.6158109307289124, 0.13117200136184692), (0.9053990244865417, 0.
↪2978983521461487)] car: 48%
Box [(0.8892407417297363, 0.19558095932006836), (0.933372974395752, 0.2684069573879242)]␣
↪potted plant: 34%
Box [(0.0753115713596344, 0.15651819109916687), (0.13415342569351196, 0.
↪22736744582653046)] motorcycle: 32%

Edge Node AMLEdgeNode.fb.d4.38.13 closing.
```

Run Inference Node

In the second terminal, run the following command to process the inference:

```
# Source colcon installation
source install/setup.bash

# To execute Inference Node with pre-trained model from TensorFlow
cd ~/AML-IP-ws/src/AML-IP/amlip_demo_nodes/amlip_tensorflow_inference_demo/amlip_
↪tensorflow_inference_demo
python3 inference_node_async.py
```

The execution expects an output similar to the one shown below:

```
2023-02-14 14:50:42.711797: I tensorflow/core/platform/cpu_feature_guard.cc:193] This␣
↪TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use␣
↪the following CPU instructions in performance-critical operations:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler␣
↪flags.
Inference Node AMLInferenceNode.b8.34.4d.a3 ready.
Model Handle at TensorFlow Hub: /home/user/AML-IP-ws/src/AML-IP/amlip_demo_nodes/amlip_
↪tensorflow_inference_demo/resource/tensorflow/models/centernet_hourglass_512x512_kpts_1
loading model...
WARNING:absl:Importing a function (__inference_batchnorm_layer_call_and_return_
↪conditional_losses_42408) with ops with unsaved custom gradients. Will likely fail if␣
↪a gradient is requested.
```

(continues on next page)

(continued from previous page)

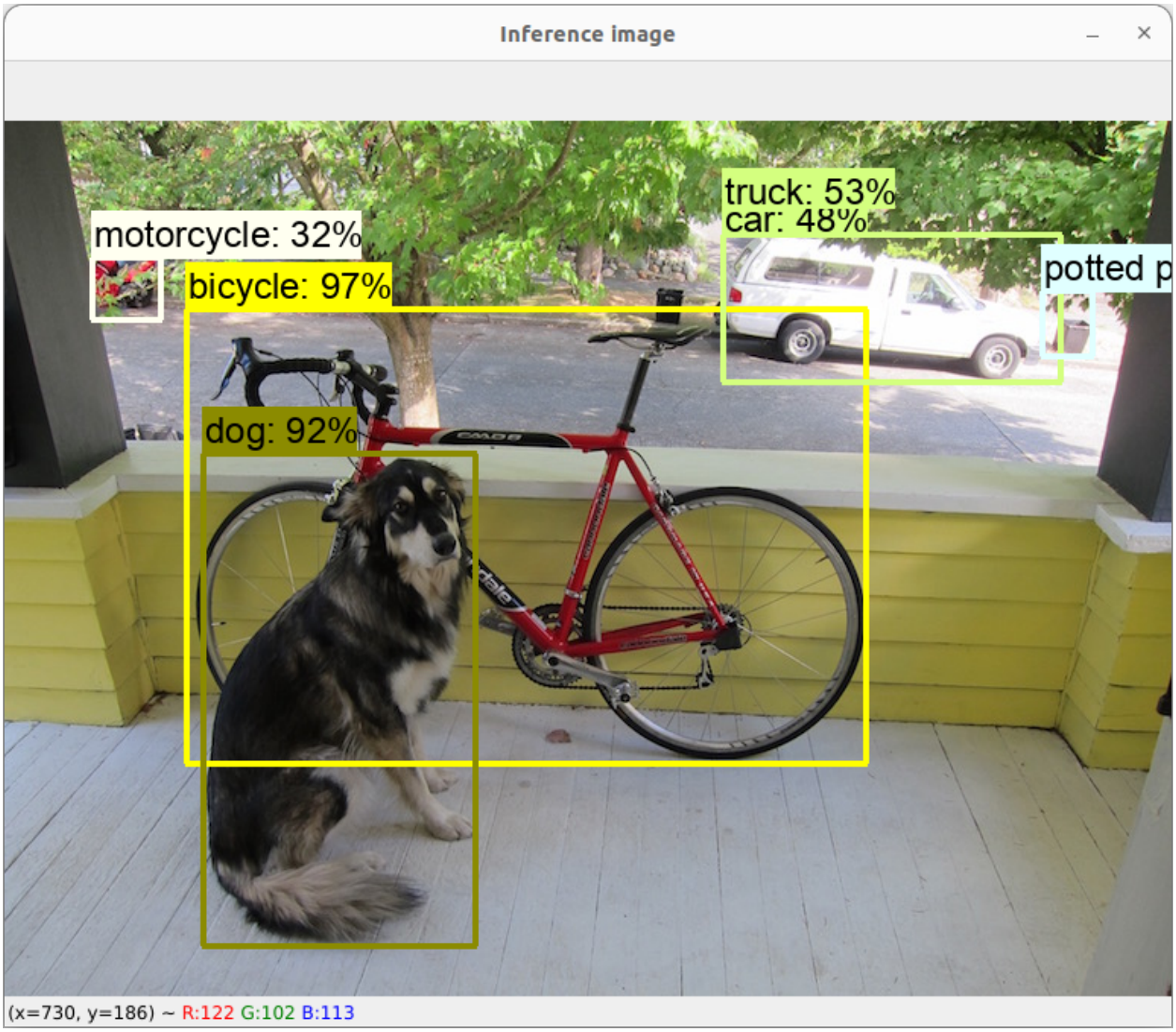
```
WARNING:absl:Importing a function (__inference_batchnorm_layer_call_and_return_
↪conditional_losses_209416) with ops with unsaved custom gradients. Will likely fail if␣
↪a gradient is requested.
WARNING:absl:Importing a function (__inference_batchnorm_layer_call_and_return_
↪conditional_losses_220336) with ops with unsaved custom gradients. Will likely fail if␣
↪a gradient is requested.
...
WARNING:absl:Importing a function (__inference_batchnorm_layer_call_and_return_
↪conditional_losses_55827) with ops with unsaved custom gradients. Will likely fail if␣
↪a gradient is requested.
WARNING:absl:Importing a function (__inference_batchnorm_layer_call_and_return_
↪conditional_losses_56488) with ops with unsaved custom gradients. Will likely fail if␣
↪a gradient is requested.
model loaded!
Selected model:tensorflow
2023-02-14 14:51:14.165305: W tensorflow/core/grappler/optimizers/loop_optimizer.cc:907]␣
↪Skipping loop optimization for Merge node with control input: StatefulPartitionedCall/
↪cond/then/_918/cond/Assert_2/AssertGuard/branch_executed/_1123
inference ready!
sending inference:
Box [(0.15590962767601013, 0.21641747653484344), (0.7388607263565063, 0.
↪7326743006706238)] bicycle: 97%
Box [(0.16968876123428345, 0.38129815459251404), (0.403958797454834, 0.
↪9422630071640015)] dog: 92%
Box [(0.6158109307289124, 0.13117200136184692), (0.9053990244865417, 0.
↪2978983521461487)] truck: 53%
Box [(0.6158109307289124, 0.13117200136184692), (0.9053990244865417, 0.
↪2978983521461487)] car: 48%
Box [(0.8892407417297363, 0.19558095932006836), (0.933372974395752, 0.2684069573879242)]␣
↪potted plant: 34%
Box [(0.0753115713596344, 0.15651819109916687), (0.13415342569351196, 0.
↪22736744582653046)] motorcycle: 32%

Inference sent to client AMLEdgeNode.fb.d4.38.13.
```

Warning: If you encounter an output similar to the next one, follow the set of instructions outlined *below*:

```
terminate called after throwing an instance of 'Swig::DirectorMethodException'
    what():  SWIG director method error. In method 'process_inference':␣
↪AttributeError: module 'tensorflow' has no attribute 'gfile'
Aborted (core dumped)
```

Next steps

Based on the information acquired, we have successfully generated the next image:



3.10.6 Run multiple nodes of each kind

One of the advantages inherent to this architecture lies in its ability to support multiple models operating concurrently across multiple *Inference Node*, while simultaneously requesting inferences from *Edge Node* in parallel. This architectural design fosters a highly efficient and scalable system, enabling the execution of diverse inference tasks in a distributed manner.

3.10.7 How to use your own model

To use your own model, simply download it and load it by passing the path to the function:

```
hub_model = hub.load(your_model_path)
```

3.10.8 Troubleshooting

TensorFlow using old API

Please be aware that Simple TensorFlow Serving is currently not compatible with TensorFlow 2.0 due to its reliance on the older API. It is important to note that in TensorFlow 2.0, the *gfile* package has been relocated under the *tf.io* module. Therefore, if you intend to utilize TensorFlow 2.0, please take into consideration this change in the package structure and update your code accordingly. Check following issue for further information.

To update the code, please follow these steps:

1. Locate the file *label_map_util.py*. (default path: `.local/lib/python3.10/site-packages/object_detection/utils/label_map_util.py`)
2. Navigate to line 132 within the file.
3. Replace *tf.gfile.GFile* with *tf.io.gfile.GFile*.

3.10.9 Next Steps

Now you can develop more functionalities in your application. See also *this tutorial* which explains how to take the image from a ROSbot 2R Camera.

3.11 TensorFlow Inference using ROSbot 2R

3.11.1 Background

This document provides detailed instructions on deploying an Edge Node on a *ROSbot 2R* from Husarion. The Edge Node will capture images using the Orbbec Astra camera and sends them to the Inference Node deployed on a laptop to perform TensorFlow inference on the given image. If the TensorFlow inference detects the presence of a person with a probability of 80% or higher, the robot will turn.

In this demo we will use the simulation of *ROSbot 2R* - an autonomous mobile robot by Husarion, designed for learning ROS and for research and development purposes. It is an affordable platform that can serve as a base for a variety of robotic applications, including inspection robots and custom service robots. The robot features a solid aluminum frame and is equipped with a Raspberry Pi 4 with 4GB of RAM, distance sensors, an RPLIDAR A2 laser scanner, and an RGB-D Orbbec Astra camera.

The demo that is presented here follows the scheme of the figure below:

Note: This tutorial assumes the reader has already reviewed *previous tutorial*, understands how Edge and Inference Nodes work and what the installation requirements are.

3.11.2 Prerequisites

Build *amlip-demos:inference-tensorflow* Docker Image from the workspace where the *Dockerfile* is located. In order to do so, execute the following:

```
cd ~/AML-IP-ws/src/AML-IP/amlip_demo_nodes/amlip_tensorflow_inference_rosbot2r_demo
docker build -t amlip-demos:inference-tensorflow -f Dockerfile .
```

Note: Use –no-cache argument to restart build.

3.11.3 ROSbot 2R Deployment

The Docker Compose used for the demo is `compose.yaml`. You can find it here.

The Docker Compose launches the following containers:

- **astra**: allows the usage of Orbbec 3D cameras with ROS Humble. It publishes the images captured by the camera to the `/camera/color/image_raw` topic. Edge Node can then subscribe to this topic to receive and process the camera images.

```
astra:
    image: husarion/astra:humble
    network_mode: host
    ipc: host
    devices:
    - /dev/bus/usb/
    volumes:
    - ./astra_params.yaml:/ros2_ws/install/astra_camera/share/astra_camera/params/astra_
↪mini_params.yaml
    privileged: true
    command: ros2 launch astra_camera astra_mini.launch.py
```

- **rosbot**: starts all base functionalities for ROSbot 2R. It subscribes to the `/cmd_vel` topic. This topic is used to control the movement of a robot by publishing velocity commands to it.

```
rosbot:
    image: husarion/rosbot:humble
    network_mode: host
    ipc: host
    privileged: true
    command: ros2 launch rosbot_bringup bringup.launch.py mecanum:=False
```

- **microros**: communicates with all firmware functionalities.

```
microros:
    image: husarion/micro-ros-agent:humble
    network_mode: host
    ipc: host
    devices:
    - ${SERIAL_PORT:?err}
    environment:
    - SERIAL_PORT
```

(continues on next page)

(continued from previous page)

```
    privileged: true
    command: ros2 run micro_ros_agent micro_ros_agent serial -D $SERIAL_PORT serial -b␣
↪576000 # -v6
```

- **edge**: is responsible for starting up the execution of the AML-IP Edge Node explained below.

```
edge:
    image: amlip-demos:inference-tensorflow
    network_mode: host
    ipc: host
    privileged: true
    command: bash -c "sleep 5 && source ./install/setup.bash && python3 ./src/amlip/
↪amlip_demo_nodes/amlip_tensorflow_inference_rosbot2r_demo/amlip_tensorflow_inference_
↪rosbot2r_demo/edge_node_async.py"
```

The following diagram illustrates the flow of the explained code:

3.11.4 Working with AML-IP

Through this section, we will delve into the details of the demo, examining the underlying concepts and processes involved.

Edge Node

Edge Node serves as the entity responsible for sending the data to be inferred to the Inference Node. The Edge Node is typically located at the edge of a network or closer to the data source, such as a sensor or a device generating the data. In this specific scenario, the data source is the camera of the robot.

The Python code for the Edge Node is explained in the *previous tutorial*, so here we will focus on the additional features added to this demo. You can find the complete code here.

The next block includes the Python header files that allow the use of the AML-IP Python API and ROS 2.

```
from amlip_py.node.AsyncEdgeNode import AsyncEdgeNode, InferenceListenerLambda
from amlip_py.types.InferenceDataType import InferenceDataType

from cv_bridge import CvBridge      # Package to convert between ROS and OpenCV Images

from geometry_msgs.msg import Twist

from py_utils.wait.BooleanWaitHandler import BooleanWaitHandler

import rclpy
from rclpy.node import Node
from rclpy.qos import QoSDurabilityPolicy
from rclpy.qos import QoSHistoryPolicy
from rclpy.qos import QoSProfile
from rclpy.qos import QoSReliabilityPolicy

from sensor_msgs.msg import Image
```

Continuing, the `SubscriberImage` ROS 2 Node subscribes to the `/camera/color/image_raw` topic to receive images from a camera sensor. It converts the received ROS Image message to an OpenCV image using the `CvBridge` package. The image data is stored in the image attribute of the node.

```
class SubscriberImage(Node):

    def __init__(self):
        super().__init__('subscriber_image')
        custom_qos_profile = QoSProfile(
                            depth=4,
                            reliability=QoSReliabilityPolicy.BEST_EFFORT)

        self.subscription = self.create_subscription(
            Image,
            '/camera/color/image_raw',
            self.listener_callback,
            custom_qos_profile)
        # Used to convert between ROS and OpenCV images
        self.br = CvBridge()
        self.image = None
        self.image_arrive = False

    def listener_callback(self, msg):
        self.get_logger().info('I received an image!!')
        # Convert ROS Image message to OpenCV image
        self.image = self.br.imgmsg_to_cv2(msg)
        self.image_arrive = True
```

The `PublisherVel` ROS 2 Node publishes `Twist` messages to the `/cmd_vel` topic, which controls the velocity of the ROSbot 2R. In the provided code, the `turn` method is implemented to set linear and angular velocities, causing the robot to turn.

```
class PublisherVel(Node):

    def __init__(self):
```

(continues on next page)

(continued from previous page)

```
        super().__init__('publisher_velocity')
        custom_qos_profile = QoSProfile(
                            history=QoSHistoryPolicy.KEEP_ALL,
                            durability=QoSDurabilityPolicy.TRANSIENT_LOCAL,
                            reliability=QoSReliabilityPolicy.BEST_EFFORT)

        self.pub = self.create_publisher(
            Twist,
            '/cmd_vel',
            custom_qos_profile)

    def turn(self):
        msg = Twist()

        msg.linear.x = 0.1
        msg.linear.y = 0.0
        msg.linear.z = 0.0

        msg.angular.x = 0.0
        msg.angular.y = 0.0
        msg.angular.z = -1.0

        self.pub.publish(msg)
```

Then, the definition of `turn_rosbot` function initializes the `PublisherVel` Node and repeatedly calls the `turn` method to make the ROSbot turn.

```
def turn_rosbot():
    node = PublisherVel()

    print('Turn ROSbot')
    loop = 0
    while rclpy.ok() and loop < 13:
        node.turn()
        time.sleep(0.5)
        loop += 1

    node.destroy_node()
```

After that, the `check_data` function extracts labels and percentages from the inference string received from the Edge Node. It searches for the label `person` with a confidence percentage greater than or equal to `80%`. If a person is detected, it calls the `turn_rosbot` function to make the robot turn.

```
def check_data(str_inference):
    labels = re.findall(r'\b(\w+):', str_inference)
    percentages = re.findall(r'(\d+)%', str_inference)
    print('Labels: ')
    print(labels)
    print('Percentages: ')
    print(percentages)
    for i in range(len(labels)):
        if (labels[i] == 'person' and int(percentages[i]) >= 80):
```

(continues on next page)

(continued from previous page)

```
            print('Found a person!')
            turn_rosbot()
            return
    print('No person found :(')
```

The `main` function initializes the `SubscriberImage` Node to receive images from the ROSbot 2R camera and waits until an image arrives before proceeding. The received image is then used to create the `AsyncEdgeNode`. The image is encoded as bytes and sent to the Edge Node for inference using the `request_inference` method as in the previous demo.

```
def main():
    rclpy.init(args=None)

    minimal_subscriber = SubscriberImage()

    # Create Node
    node = AsyncEdgeNode(
        'AMLAsyncEdgeNode',
        listener=InferenceListenerLambda(inference_received),
        domain=DOMAIN_ID)

    while True:

        while rclpy.ok() and not minimal_subscriber.image_arrive:
            rclpy.spin_once(minimal_subscriber, timeout_sec=1)

        img = minimal_subscriber.image

        print(f'Async Edge Node {node.id()} ready.')

        width = img.shape[1]
        height = img.shape[0]

        # Convert size to bytes
        str_size = str(width) + ' ' + str(height) + ' | '
        bytes_size = bytes(str_size, 'utf-8')
        # Convert image to bytes
        img_bytes = base64.b64encode(img)
        # Size + images
        img_size_bytes = bytes_size + img_bytes

        print(f'Edge Node {node.id()} sending data.')

        task_id = node.request_inference(InferenceDataType(img_size_bytes))

        print(f'Request sent with task id: {task_id}. Waiting inference...')

        # Wait to received solution
        waiter.wait()

        minimal_subscriber.image_arrive = False
```

(continues on next page)

(continued from previous page)

```
    # Closing
    minimal_subscriber.destroy_node()
    rclpy.shutdown()

    print(f'Edge Node {node.id()} closing.')
```

Inference Node

The Inference Node is responsible for making the inferences or predictions on the data it receives using a TensorFlow model. The Inference Node is typically a server or a computing resource equipped with high-performance hardware optimized for executing machine learning models efficiently.

The Python code for the Inference Node is explained in the previous tutorial and can be found here.

3.11.5 Run demo

Bring up ROSbot 2R and run Edge Node

First, it is necessary to launch the docker compose `compose.yaml` that will activate the containers `astra`, `rosbot`, `microros` and `edge`.

Start the containers in a new *ROSbot* terminal, run the following command:

```
cd ~/AML-IP-ws/src/AML-IP/amlip_demo_nodes/amlip_tensorflow_inference_rosbot2r_demo
docker compose up
```

Run Inference Node

In a terminal on your laptop, run the following command in order to process the inference:

```
# Source colcon installation
source install/setup.bash

# To execute Inference Node with pre-trained model from TensorFlow
cd ~/AML-IP-ws/src/AML-IP/amlip_demo_nodes/amlip_tensorflow_inference_rosbot2r_demo/
↪amlip_tensorflow_inference_rosbot2r_demo
python3 inference_node_async.py
```

Teleoperate ROSbot 2R

Note that ROSbot 2R subscribes to the `/cmd_vel` topic. To teleoperate it, you can use the `teleop_twist_keyboard` node, which allows you to control the robot using keyboard inputs. Follow these steps:

```
# Source ROS 2 installation
source install/setup.bash

ros2 run teleop_twist_keyboard teleop_twist_keyboard
```

3.12 Workload Distribution

This demonstrator shows how to implement 2 kind of nodes: *Computing Node* and *Main Node*. With these 2 nodes implemented, the user can deploy as many nodes of each kind as desired and check the behavior of a simulated *AML-IP* network running. They are implemented one in Python and one in C++ to demonstrate as well how to instantiate each kind of node with different *APIs*, and to prove the communication between the 2 implementations.

3.12.1 Simulation

AML Mock

For this demo the actual *AML* Engine is not provided, and it is mocked. This *Mock* simulates a difficult calculation by converting a string to uppercase and randomly waiting between 1 and 5 seconds in doing so.

Main Node

This node simulates a *Main Node*. It does not use real *AML Jobs*, but strings. It is implemented in Python using `amlip_py` API. There are 2 different ways to run it, an automatic one and a manual one:

Automatic version

In this version, the python executable expects input arguments. For each argument, it will convert it to a string (`str`) and send it as a *Job*. Once the arguments run out, it will finish execution and destroy the Node.

Manual version

In this version the python program expects to receive keyboard input. For each keyboard input received, it will convert it to a string (`str`) and send it as a *Job*. When empty string given, it will finish execution and destroy the Node.

Computing Node

This node simulates a *Computing Node*. It does not use real *AML Jobs*, but strings. It does not have a real *AML Engine* but instead the calculation is an *upper-case* conversion of the string received. It is implemented in C++ using `amlip_cpp` API.

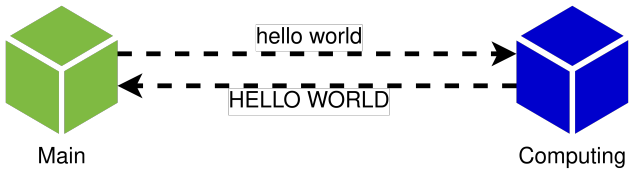
To run it, one integer argument is required. This will be the number of jobs this Node will answer to before finishing its execution and being destroyed.

3.12.2 Installation

First of all, check that `amlip_demo_nodes` sub-package is correctly installed. If it is not, please refer to *Build demos*.

3.12.3 Run demo

The demo that is presented here follows the schema of the figure below:



Run Main Node

Run the following command:

```
# Source colcon installation
source install/setup.bash

# To execute Main Node to send 2 jobs
python3 ./install/amlip_demo_nodes/bin/main_node.py first_job "second job"
```

Take into account that this node will wait until there are *Computing Nodes* running and available in the same *LAN* in order to solve the jobs. The expected output is the following:

```
Main Node AMLMainNode.aa.a5.47.fe ready.
Main Node AMLMainNode.aa.a5.47.fe sending task <first_job>.
# ... Waits for Computing Node
Main Node received solution from AMLComputingNode.d1.c3.86.0a for job <first_job> =>
↪<FIRST_JOB>.
Main Node AMLMainNode.aa.a5.47.fe sending task <second job>.
Main Node received solution from AMLComputingNode.d1.c3.86.0a for job <second job> =>
↪<SECOND JOB>.
Main Node AMLMainNode.aa.a5.47.fe closing.
```

Run Computing Node

Run the following command to answer 2 jobs before closing:

```
# Source colcon installation
source install/setup.bash

# To execute Computing Node to answer 2 jobs
./install/amlip_demo_nodes/bin/computing_node 2
```

Take into account that this node will wait until it has solved 2 different jobs. If there are more than 1 *Computing Node* running, one job is only solved by one of them. This execution expects an output similar to the one shown below:

```
Computing Node ID{AMLComputingNode.d1.c3.86.0a} computing 2 tasks.
# ... Waits for Main Node
 Received Job: <first_job>. Processing...
 Answering Solution: <FIRST_JOB>.
Computing Node ID{AMLComputingNode.d1.c3.86.0a} answered task. 1 remaining.
```

(continues on next page)

(continued from previous page)

```
 Received Job: <second job>. Processing...
 Answering Solution: <SECOND JOB>.
Computing Node ID{AMLComputingNode.d1.c3.86.0a} answered task. 0 remaining.
Computing Node ID{AMLComputingNode.d1.c3.86.0a} closing.
```

3.12.4 Bigger scenarios

There is no limit in the number of nodes of each kind that could run in the same network. However, take into account that these nodes are not meant to close nicely if they do not finish their tasks correctly, thus calculate the number of jobs sent in order for all nodes to close gently.

3.13 AML-IP Scenarios

The *AML-IP* framework is divided in different **scenarios** or **use cases** that allow it to exploit all the capabilities *AML* has to offer. These scenarios work independently of each other and make sense separately, but can be seamlessly combined to create a more complex network. Each of the scenarios rely on a different set of **Nodes** that perform the different actions required.

In order to know more about the *Node* concept and their kinds, please refer to *AML-IP Node* section.

3.13.1 Monitor Network State Scenario

This *Scenario* performs the monitoring action: knowing, analyzing and debugging an *AML* network. Each of the *AML-IP Nodes Publish* their current *Status* information and update it along their lifetimes. This scenario supports *subscription* to this *Topic* in order to receive such status information, that can be processed, stored, read, etc.

Status Data Type

The *Status* published by the nodes has the following information:

- **Node Id**: Uniquely identifies a Node. Check following *section*.
- **Current State**: The current state of the node, that can be `stopped`, `running` or `dropped`.
- **Node Kind**: Specifies which kind of node is, and so to which Scenario belongs.

Nodes

This scenario involves every Node, as all of them publish the *Status* information. However, the only Node Kind properly belonging to this Scenario is *Status Node*.

3.13.2 Workload Distribution Scenario

This *Scenario* performs the action of distributing a high computational effort *Task* in remote nodes, in order to parallelize the task and do not block any other important actions that may require to run in the same device. It uses the *MultiService over DDS* communication to publish those tasks in an efficient way.

The *Task* distributed is the training data-set of an *AML* model. This model is stored in a *Main Node* and the training data-set is divided in different *Jobs*, that are sent along with states of the model to *Computing Nodes* in order to perform this training in parallel, reducing the workload in the *Main Node* host, that may require to perform other actions at the same time.

Job Data Type

The **Job** Data Type represents a partial data-set and a model state. Internally, *Jobs* sent from a *Main Node* to a *Computing Node* are treated as byte arrays of arbitrary size. So far, the interaction with this class could be done from a `void*`, a byte array or a string. From Python API, the only way to interact with it is by *str* type.

Note: A more specific Data Type will be implemented in future releases for efficiency improvements.

Job Solution Data Type

The **Solution** Data Type represents an *Atomization* or new model state. The **Solution** sent from a *Computing Node* to a *Main Node* is treated as a bytes array of arbitrary size. So far, the interaction with this class could be done from a `void*`, a byte array or a string. From Python API, the only way to interact with it is by *str* type.

Note: A more specific Data Type will be implemented in future releases for efficiency improvements.

3.13.3 Collaborative Learning Scenario

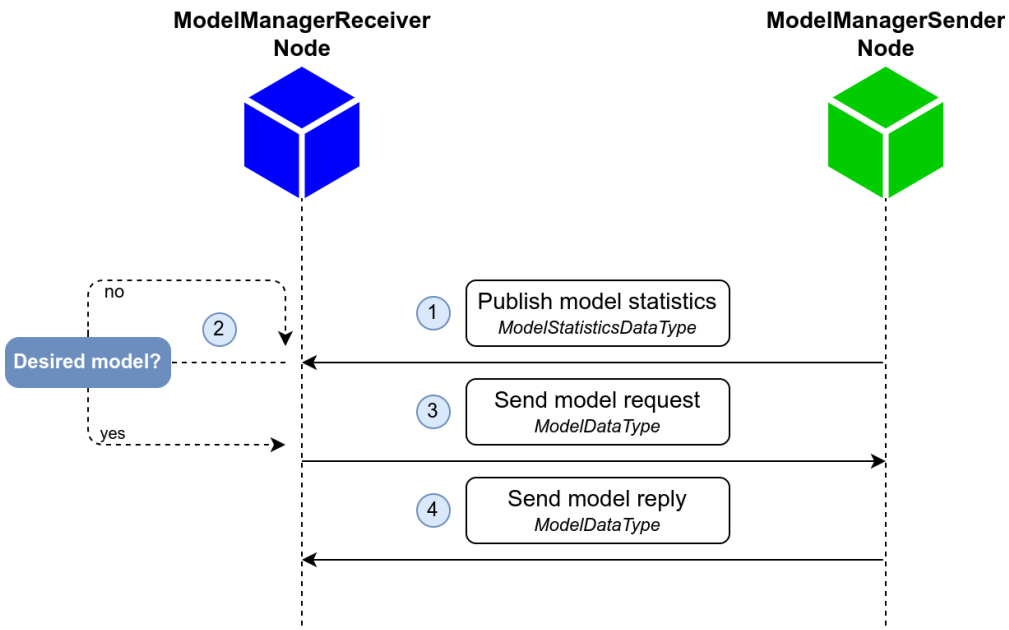
In this *Scenario*, *Model Manager Receiver* and *Model Manager Sender* nodes working on the same problem share their locally obtained models with each other, without having to share the private datasets with which they were trained on. This intends to lead towards a more complex and accurate model. It leverages the *RPC over DDS* communication protocol/paradigm in order to exchange all required information (model requests/replies) in an efficient way.

The Model Manager Sender Nodes publish *Model Statistics Data Type* 's while the Model Manager Receiver Nodes listen to them. When a *Model Manager Receiver Node* is interested in a model based on its *Model Statistics Data Type*, it sends a *Model Request Data Type* request to the Model Manager Sender Node that sent the *Model Statistics Data Type*. The *Model Manager Sender Node* will respond with a *Model Reply Data Type*.

Model Request Data Type

The **Model Request** Data Type represents a problem request. Internally, *requests* sent from a *Model Manager Receiver Node* to a *Model Manager Sender Node* are treated as byte arrays of arbitrary size. So far, the interaction with this class could be done from a `void*`, a byte array or a string.

Note: A more specific Data Type will be implemented in future releases for efficiency improvements.



Model Reply Data Type

The **Model Reply** Data Type represents a problem reply with the requested model. The *replies* sent from a *Model Manager Sender Node* to a *Model Manager Receiver Node* are treated as a bytes array of arbitrary size. So far, the interaction with this class could be done from a `void*`, a byte array or a string.

Note: A more specific Data Type will be implemented in future releases for efficiency improvements.

Model Statistics Data Type

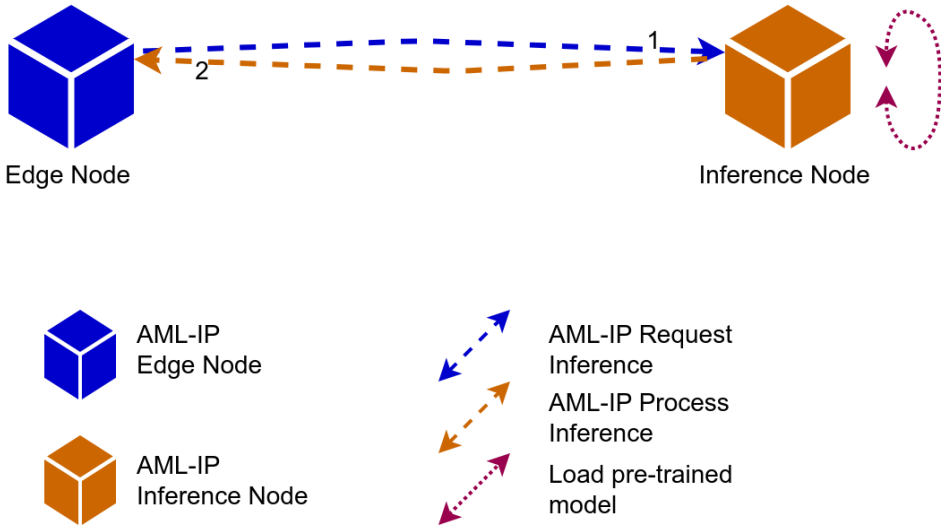
The **Statistics** Data Type represents the statistics of models, such as their number of parameters or the datasets they were trained on. The *messages* sent from a *Model Manager Sender Node* to a *Model Manager Receiver Node* are treated as a bytes array of arbitrary size. So far, the interaction with this class could be done from a `void*`, a byte array or a string.

Note: A more specific Data Type will be implemented in future releases for efficiency improvements.

3.13.4 Distributed Inference Scenario

This *Scenario* involves the action of distributing a large amount of *Data* to remote nodes to perform their inferences, in order to parallelize them and do not block any other important actions that may require to run in the same device. By performing this action, the system ensures seamless execution of multiple tasks, optimizing overall performance and resource utilization. It uses the *MultiService over DDS* to efficiently publish and distribute the data across remote nodes, ensuring a streamlined and effective process.

The inference is performed in a *Inference Node* and sent to an *Edge Node*.



Inference Data Type

The **Inference** Data Type represents a partial data-set. Internally, the data sent from an *Edge Node* to an *Inference Node* are treated as byte arrays of arbitrary size. So far, the interaction with this class could be done from a `void*`, a byte array or a string. From Python API, the way to interact with it is by *str* and *bytes* type.

Inference Solution Data Type

The **Inference Solution** Data Type represents the inference of the data sent by the *Edge Node*. The **Inference Solution** sent from an *Inference Node* to an *Edge Node* is treated as a bytes array of arbitrary size. So far, the interaction with this class could be done from a `void*`, a byte array or a string. From Python API, the way to interact with it is by *str* and *bytes* type.

Note: There is no real data type here, the data format inside is whatever the user wants it to be.

3.14 AML-IP Node

An *AML-IP* network is divided in independent stand-alone individuals named *Nodes*. A Node, understood as a software piece that performs one or multiple *Actions* in a auto-managing way, does not require external orchestration neither a central point of computation. These actions can be local actions such as calculations, data process, algorithm executions, etc., or communication actions as send messages, receive data, wait for data or specific status, etc. Each Node belongs to one and only one *Scenario*.

There are different ways to run or to work with a Node. Some of them are applications that can be executed and perform a fixed action. Others, however, require a user interaction as specifying the action such Node must perform depending on its status and the data received. In this last case, the Nodes are programming *Objects* that can be instantiated and customized regarding the action that must be performed.

Warning: *AML-IP* is a work in progress, and these interactions and usage can change along the process.

3.14.1 Nodes attributes

Node Id

Each Node in a network has a unique Id. This Id is created by the Node *name*, that is given to each node once it is created, and a randomly generated number. The name of a node must consist of less than 28 characters between `0-9 a-z A-Z _ .`

Examples of Nodes Ids are: `AMLIPNode.d2.24.9f.34`, `My_Custom_Node.58.cd.72.85`, `status.node_7.37.18.f7.05`.

Node Kind

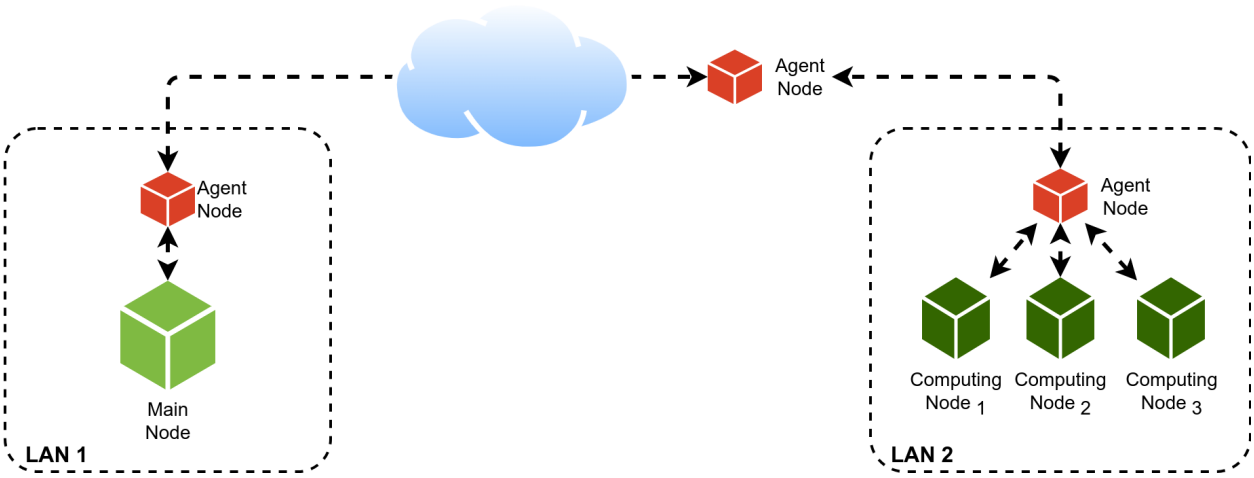
Each Node belongs to a specific *Kind*. The kind of the node identifies it, and makes it behave and perform different actions. There are no restrictions in the number of nodes of the same kind running in the same network. The kind of the nodes follows the Object Oriented Programming ideas, where every Node Kind represents a *class* and can inherit from other Node *classes* (e.g. every Node Kind inherits from `ParentNode`); and each running node in the network is an *instance* of the *class* of its Kind.

3.14.2 Node Kinds

Agent Node

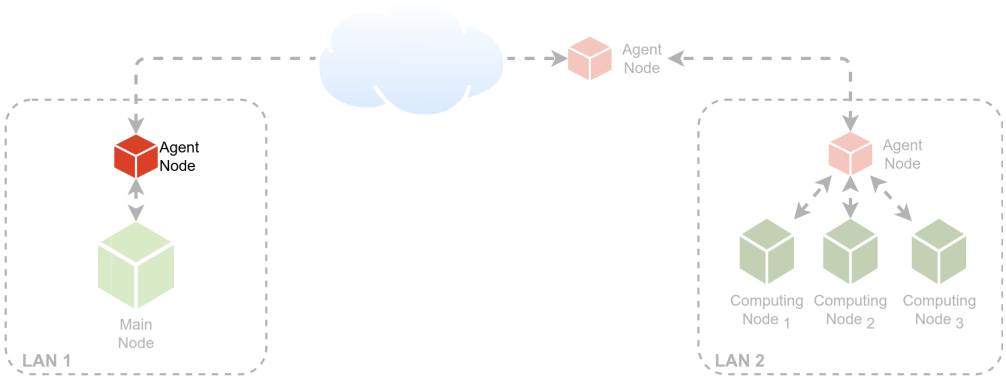
The Agent Node relays on the eProsima DDS Router. This tool is developed and maintained by *eProsima* which enables the connection of distributed DDS networks. DDS entities such as publishers and subscribers deployed in one geographic location and using a dedicated local network will be able to communicate with other DDS entities deployed in different geographic areas on their own dedicated local networks as if they were all on the same network.

This node is in charge of communicating a local node or AML-IP cluster with the rest of the network in WANs. It centralizes the WAN discovery and communication, i.e. it is the bridge for all the nodes in their LANs with the rest of the AML-IP components.



Client Node

This node acts as a communication client that connects to a Server Node.



Steps

- Create a new `eprosima::ddspipe::participants::types::Address` object with the address port, external address port, *IP* address and transport protocol.
- Instantiate the `ClientNode` creating an object of such class with a name, a connection address and a domain.
- Wait until `Ctrl+C`.

C++

```
// Create connection address
auto connection_address = eprosima::ddspipe::participants::types::Address(
    12121,
    12121,
    "localhost",
    eprosima::ddspipe::participants::types::TransportProtocol::udp);

// Create Client Node
eprosima::amlip::node::agent::ClientNode Client_node(
    "CppClientNode_Manual",
    { connection_address },
    100);

// Wait until Ctrl+C
eprosima::utils::event::SignalEventHandler<eprosima::utils::event::Signal::sigint>␣
↪sigint_handler;
sigint_handler.wait_for_event();
```

Python

```
# Create connection address
connection_address = Address(
    port=12121,
    external_port=12121,
    domain='localhost',
```

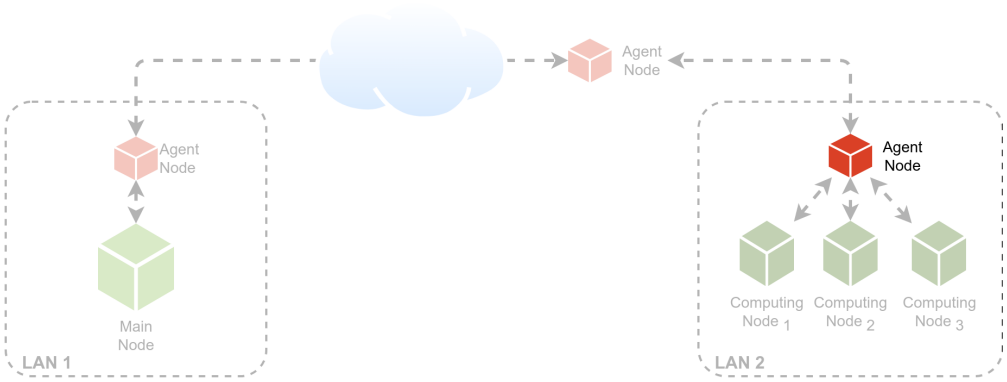
(continues on next page)

(continued from previous page)

```
    transport_protocol=TransportProtocol_udp)

# Create Client Node
ClientNode(
    name='PyTestClientNode',
    connection_addresses=[connection_address],
    domain=100)

# Wait until Ctrl+C
def handler(signum, frame):
    pass
signal.signal(signal.SIGINT, handler)
signal.pause()
```

Server Node

This node acts as a communication server, waiting for other Client Nodes to connect to it.



Steps

- Create a new `eprosima::ddspipe::participants::types::Address` object with the address port, external address port, *IP* address and transport protocol.
- Instantiate the `ServerNode` creating an object of such class with a name, a listening address and a domain.
- Wait until `Ctrl+C`.

C++

```
// Create listening address
auto listening_address = eprosima::ddspipe::participants::types::Address(
    12121,
    12121,
    "localhost",
    eprosima::ddspipe::participants::types::TransportProtocol::udp);

// Create Server Node
eprosima::amlip::node::agent::ServerNode Client_node(
```

(continues on next page)

(continued from previous page)

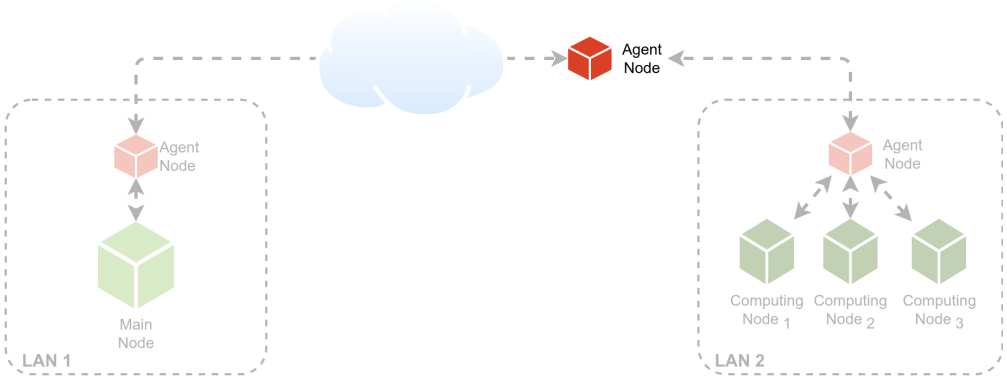
```
    "CppServerNode_Manual",
    { listening_address },
    200);

// Wait until Ctrl+C
eprosima::utils::event::SignalEventHandler<eprosima::utils::event::Signal::sigint>␣
↪sigint_handler;
sigint_handler.wait_for_event();
```

Python

```
# Create listening address
listening_address = Address(
    port=12121,
    external_port=12121,
    domain='localhost',
    transport_protocol=TransportProtocol_udp)

# Create Server Node
ServerNode(
    name='PyTestServerNode',
    listening_addresses=[listening_address],
    domain=200)

# Wait until Ctrl+C
def handler(signum, frame):
    pass
signal.signal(signal.SIGINT, handler)
signal.pause()
```

Repeater Node

A Repeater Node can be used to repeat messages between networks, that is, the message will be forwarded using the same network interface. This is useful to communicate across LANs.



Steps

- Create a new `eprosima::ddspipe::participants::types::Address` object with the address port, external address port, *IP* address and transport protocol.
- Instantiate the `RepeaterNode` creating an object of such class with a name, a listening address and a domain.
- Wait until `Ctrl+C`.

C++

```
// Create listening address
auto listening_address = eprosima::ddspipe::participants::types::Address(
    12121,
    12121,
    "localhost",
    eprosima::ddspipe::participants::types::TransportProtocol::udp);

// Create Repeater Node
eprosima::amlip::node::agent::RepeaterNode repeater_node(
    "CppRepeaterNode_Manual",
    { listening_address });

// Wait until Ctrl+C
eprosima::utils::event::SignalEventHandler<eprosima::utils::event::Signal::sigint>␣
↪sigint_handler;
sigint_handler.wait_for_event();
```

Python

```
# Create listening address
listening_address = Address(
    port=12121,
    external_port=12121,
    domain='localhost',
    transport_protocol=TransportProtocol_udp)

# Create Repeater Node
RepeaterNode(
    name='PyTestRepeaterNode',
    listening_addresses=[listening_address])

# Wait until Ctrl+C
def handler(signum, frame):
    pass
signal.signal(signal.SIGINT, handler)
signal.pause()
```

Status Node

This kind of node *Subscribe* to *Status Topic*. Thus it receives every *Status* data from all the other *Nodes* in the network. This node runs a *function* that will be executed with each message received. This is the main agent of *Monitor Network State Scenario*.

Example of Usage

This node kind does require **few interaction** with the user once it is running. User must start and stop this node as desired using methods `process_status_async` and `stop_processing`. Also, user must yield a callback (function) that will be executed with every *Status* message received. By destroying the node it stops if running, and every internal entity is correctly destroyed.

Steps

- Instantiate the Status Node creating an object of such class with a name.
- Start processing status data of the network calling `process_status_async`.
- Stop processing data calling `stop_processing`.

C++

```
// Create a new Status Node
auto node = eprosima::amlip::StatusNode("My_Status_Node");

// Process arrival data by printing it in stdout (defined by std::function)
node.process_status_async(
    []( const eprosima::amlip::types::StatusDataType& status ){ std::cout << status <<␣
↪std::endl; });

// Do other cool things here

// Stop processing data
node.stop_processing()
```

Python

```
# Create a new Status Node
node = StatusNode("My Status Node");

# Process arrival data by printing it in stdout (defined by lambda)
node.process_status_async(
    callback=lambda status: print(f'{status}'))

# Do other cool things here

# Stop processing data
node.stop_processing()
```

Main Node

This kind of Node performs the active (client) action of *Workload Distribution Scenario*. This node is able to send different *Jobs* serialized as *Job Data Type* and it receives a Solution once the task has been executed as *Job Solution Data Type*.

Synchronous

This node kind does require **active** interaction with the user to perform its action. This means that once a job is sent, the thread must wait for the solution to arrive before sending another task. Users can use method `request_job_solution` to send a new *Job*. The thread calling this method will wait until the whole process has finished and the *Solution* has arrived from the *Computing Node* in charge of this *Job*. By destroying the node every internal entity is correctly destroyed.

Steps

- Instantiate the Main Node creating an object of such class with a name.
- Create a new `JobDataType` from an array of bytes.
- Send a new *Job* synchronously and wait for the solution by calling `request_job_solution`.

C++

```
// Create a new Main Node
auto node = eprosima::amlip::MainNode("My_Main_Node");

// Create a new job to be executed remotely
auto new_job = eprosima::amlip::JobDataType("Some Job as byte array serialized from a␣
↪string");

// Send a Job to a remote Computing and waits for the answer
// This could be called with an id as well, and it will return the server id that send␣
↪the solution
auto solution = node.request_job_solution(new_job);
```

Python

```
# Create a new Main Node
node = MainNode("My_Main_Node")

# Create a new job to be executed remotely
new_job = JobDataType("Some Job as byte array serialized from a string")

# Send a Job to a remote Computing and waits for the answer
solution, server_id = node.request_job_solution(new_job)
```

Asynchronous

Users can use method `request_job_solution` to send a new *Job*. The thread calling this method must wait until the whole process has finished and the *Solution* has arrived from the *Computing Node* in charge of this data that will process it by the Listener or callback given, and return the Solution calculated in other thread. By destroying the node every internal entity is correctly destroyed.

Steps

- Instantiate the Asynchronous Main Node creating an object of such class with a name, a listener or callback and a domain.
- Create a new `JobDataType` from an array of bytes.
- Send a new *Job* synchronously and wait for the solution by calling `request_job_solution`.
- Wait for the solution.

Python

```
def solution_received(
        solution,
        task_id,
        server_id):
    print(f'Solution received from server: {server_id}\n'
        f' with id: {task_id}\n'
        f' solution: {solution.to_string()}')

def main():
    # Create a new Async Main Node
    node = AsyncMainNode(
        'MyAsyncMainNode',
        callback=solution_received,
        domain=100)

    # Create new data to be executed remotely
    data_str = '<Job Data In Py String Async [LAMBDA]>'
    job_data = JobDataType(data_str)

    # Send data to a remote Computing Node and waits for the solution
    task_id = main_node.request_job_solution(job_data)

    # User must wait to receive solution.
    # Out of scope, the node will be destroyed,
    # and thus the solution will not arrive.
```

Computing Node

This kind of Node performs the passive (server) action of *Workload Distribution Scenario*. This node waits for a *Job* serialized as *Job Data Type*, and once received it performs a calculation (implemented by the user) whose output is the solution as *Job Solution Data Type*.

Synchronous

This node kind does require **active** interaction with the user to perform its action. This means that once a job is sent, the thread must wait for the solution to arrive before sending another task. User can use method `request_job_solution` to send a new *Job*. The thread calling this method will wait until the whole process has finished and the *Solution* has arrived from the *Computing Node* in charge of this *Job*. By destroying the node every internal entity is correctly destroyed.

Steps

- Instantiate the Computing Node creating an object of such class with a name.
- Create a new `JobDataType` from an array of bytes.
- Send a new *Job* synchronously and wait for the solution by calling `request_job_solution`.

C++

```
// Create a new Computing Node
auto node = eprosima::amlip::ComputingNode("My_Computing_Node");

// Create a callback to process a job and return a solution
auto process_solution = []( const eprosima::amlip::types::JobDataType& ){
    eprosima::amlip::types::JobSolutionDataType solution;
    // Do some code that calculates the solution
    return solution;
};

// Wait for 1 task from any client and answer it with process_solution callback
node.process_job(process_solution);
```

Python

```
# Create a new Computing Node
node = ComputingNode("My_Computing_Node")

def process_solution():
    JobSolutionDataType solution;
    # Do some code that calculates the solution
    return solution

# Wait for 1 task from any client and answer it with process_solution callback
node.process_job(callback=process_solution)
```

Asynchronous

User can use method `request_job_solution` to send a new *Job* from *Main Node* to send new data. The thread calling this method will wait until the whole process has finished and the *Solution* has arrived from the *Computing Node* in charge of this *Job*. By destroying the node every internal entity is correctly destroyed.

Steps

- Instantiate the Asynchronous Computing Node creating an object of such class with a name, a listener or callback and a domain.
- Wait for tasks by calling `run`.

Python

```
def process_job(
        job,
        task_id,
        client_id):
    JobSolutionDataType solution;
    # Do some code that calculates the solution
    return solution

# Create a new Async Computing Node
node = AsyncComputingNode(
    'MyAsyncComputingNode',
    callback=process_job,
    domain=100)

node.run()

# Wait until Ctrl+C

node.stop()
```

Edge Node

This node is able to send data serialized as *Inference Data Type* and it receives an Inference as *Inference Solution Data Type*.

Synchronous

This node kind does require **active** interaction with the user to perform its action. Once the data is sent, the thread must wait for the inference to arrive before sending another data. Users can use method `request_inference` to send new data. The thread calling this method will wait until the whole process has finished and the *Inference* has arrived from the *Inference Node* in charge of this data. By destroying the node every internal entity is correctly destroyed.

Steps

- Instantiate the Edge Node creating an object of such class with a name.
- Create a new `InferenceDataType` from an array of bytes.
- Send a data synchronously and wait for the inference by calling `request_inference`.

C++

```
// Create a new Edge Node
auto node = eprosima::amlip::EdgeNode("My_Edge_Node");

// Create new data to be executed remotely
auto data = eprosima::amlip::InferenceDataType("Some data as byte array serialized from␣
↪a string or bytes");

// Send data to a remote Inference Node and waits for the inference
// This could be called with an id as well, and it will return the server id that send␣
↪the inference
auto solution = node.request_inference(data);
```

Python

```
# Create a new Edge Node
node = EdgeNode("My_Edge_Node")

# Create new data to be executed remotely
data = InferenceDataType("Some data as byte array serialized from a string or bytes")

# Send data to a remote Inference Node and waits for the inference
inference, server_id = node.request_inference(data)
```

Asynchronous

Users can use method `request_inference` to send new data. The thread calling this method must wait until the whole process has finished and the *Inference* has arrived from the *Inference Node* in charge of this data that will process it by the Listener or callback given, and return the Inference calculated in other thread. By destroying the node every internal entity is correctly destroyed.

Steps

- Instantiate the Asynchronous Edge Node creating an object of such class with a name, a listener or callback and a domain.
- Create a new `InferenceDataType` from an array of bytes.
- Send a data synchronously calling `request_inference`.
- Wait for the inference.

Python

```
def inference_received(
        inference,
        task_id,
        server_id):
    print(f'Data received from server: {server_id}\n'
          f' with id: {task_id}\n'
          f' inference: {inference.to_string()}')

def main():
    # Create a new Async Edge Node
    node = AsyncEdgeNode(
        "My_Async_Edge_Node",
        listener=InferenceListenerLambda(inference_received),
        domain=DOMAIN_ID)

    # Create new data to be executed remotely
    data = InferenceDataType("Some data as byte array serialized from a string or bytes")

    # Send data to a remote Inference Node and waits for the inference
    task_id = node.request_inference(data)

    # User must wait to receive solution.
    # Out of scope, the node will be destroyed,
    # and thus the solution will not arrive.
```

Inference Node

This node waits for data serialized as *Inference Data Type*, and once received it calculate the inference whose output is the inference solution as *Inference Solution Data Type* and send the result back.

Synchronous

This node kind does require **active** interaction with the user to perform its action. This means that calling *process_inference* will wait for receiving data, and will only finish when the result is sent back. User can use method `request_inference` from *Edge Node* to send new data. The thread calling this method will wait until the whole process has finished and the *Inference* has arrived from the *Inference Node* in charge of this data. By destroying the node every internal entity is correctly destroyed.

Steps

- Instantiate the Inference Node creating an object of such class with a name.
- Wait for the data by calling `process_inference`.
- Return the inference as an `InferenceSolutionDataType`.

C++

```
// Create a new Inference Node
auto node = eprosima::amlip::InferenceNode("My_Inference_Node");
```

(continues on next page)

(continued from previous page)

```
// Create a callback to process data and return the inference
auto engine_routine = []( const eprosima::amlip::types::InferenceDataType& dataset){
    eprosima::amlip::types::InferenceSolutionDataType inference;
    // Do some code that calculates the inference
    return inference;
};

// Wait for 1 task from any client and answer it with process_inference callback
node.process_inference(engine_routine);
```

Python

```
# Create a new Inference Node
node = InferenceNode("My_Inference_Node")

def engine_routine(dataset):
    # Do some code that calculates the inference
    return InferenceSolutionDataType(inference_solution)

# Wait for 1 task from any client and answer it with process_inference callback
node.process_inference(callback=lambda inference: engine_routine(dataset))
```

Asynchronous

User can use method `request_inference` from *Edge Node* to send new data. The thread calling this method must wait until the whole process has finished and the *Inference* has arrived from the *Inference Node* in charge of this data that will process it by the Listener or callback given, and return the Inference calculated in other thread. By destroying the node every internal entity is correctly destroyed.

Steps

- Instantiate the Asynchronous Inference Node creating an object of such class with a name, a listener or callback and a domain.
- Wait for the data by calling `run`.

Python

```
def process_inference(
        dataset,
        task_id,
        client_id):
    # Do some code that calculates the inference
    return inference_solution

def main():

    # Create a new Async Inference Node
    node = AsyncInferenceNode(
        "My_Async_Inference_Node",
        listener=InferenceReplierLambda(process_inference),
```

(continues on next page)

(continued from previous page)

```
        domain=100)

    inference_node.run()

    # Wait until Ctrl+C

    inference_node.stop()
```

Model Manager Receiver Node

This kind of Node performs the active (client) action of *Collaborative Learning Scenario*. This node receives statistics about models and sends a request if it is interested in a particular one. Then, waits for the arrival of the requested model, serialized as *Model Reply Data Type*.

Example of Usage

Steps

- Create the Id of the node.
- Create the data to request.
- Instantiate the ModelManagerReceiver Node creating an object of such class with the Id and data previously created.
- Start the execution of the node.
- Wait for statistics.
- Request the model.
- Stop the execution of the node.

C++

```
#include <cpp_utils/wait/BooleanWaitHandler.hpp>

#include <amlip_cpp/types/id/AmlipIdDataType.hpp>
#include <amlip_cpp/types/model/ModelRequestDataType.hpp>
#include <amlip_cpp/types/model/ModelReplyDataType.hpp>
#include <amlip_cpp/types/model/ModelStatisticsDataType.hpp>

#include <amlip_cpp/node/collaborative_learning/ModelManagerReceiverNode.hpp>


class CustomModelListener : public eprosima::amlip::node::ModelListener
{
public:

    CustomModelListener(
            const std::shared_ptr<eprosima::utils::event::BooleanWaitHandler>& waiter)
        : waiter_(waiter)
    {
```

(continues on next page)

(continued from previous page)

```
        // Do nothing
    }

    virtual void statistics_received (
            const eprosima::amlip::types::ModelStatisticsDataType statistics) override
    {
        // Save the server id
        server_id = statistics.server_id();

        waiter_->open();

        // Always request model
        return true;
    }

    virtual bool model_received (
            const eprosima::amlip::types::ModelReplyDataType model) override
    {
        std::cout << "Model received: " << model << " ." << std::endl;

        return true;
    }

    std::shared_ptr<eprosima::utils::event::BooleanWaitHandler> waiter_;
    eprosima::amlip::types::AmlipIdDataType server_id;
};

// Create the Id of the node
eprosima::amlip::types::AmlipIdDataType id({"ModelManagerSender"}, {10, 20, 30, 40});

// Create the data
eprosima::amlip::types::ModelRequestDataType data("MobileNet V1");

// Create ModelManagerReceiver Node
eprosima::amlip::node::ModelManagerReceiverNode model_receiver_node(id, data);

// Create waiter to wait for statistics
std::shared_ptr<eprosima::utils::event::BooleanWaitHandler> waiter =
    std::make_shared<eprosima::utils::event::BooleanWaitHandler>(false, true);

// Create listener to process statistics and replies
std::shared_ptr<CustomModelListener> listener =
    std::make_shared<CustomModelListener>(waiter);

// Start execution
model_receiver_node.start(listener);

// Wait for statistics
waiter->wait();

// ... do something ...
```

(continues on next page)

(continued from previous page)

```
// Request the model
model_receiver_node.request_model(listener->server_id);

// Stop execution
model_receiver_node.stop();
```

Python

```
from py_utils.wait.BooleanWaitHandler import BooleanWaitHandler

from amlip_py.types.AmlipIdDataType import AmlipIdDataType
from amlip_py.types.ModelRequestDataType import ModelRequestDataType
from amlip_py.types.ModelReplyDataType import ModelReplyDataType
from amlip_py.types.ModelStatisticsDataType import ModelStatisticsDataType

from amlip_py.node.ModelManagerReceiverNode import ModelManagerReceiverNode,␣
↪ModelListener

# Variable to wait for the statistics
waiter = BooleanWaitHandler(True, False)

class CustomModelListener(ModelListener):

    def statistics_received(
            self,
            statistics: ModelStatisticsDataType) -> bool:

        waiter.open()

        return True

    def model_received(
            self,
            model: ModelReplyDataType) -> bool:
        print(f'Model reply received from server\n'
              f' solution: {model.to_string()}')

        return True

# Create the data
data = ModelRequestDataType('MobileNet V1')

# Create the Id of the node
id = AmlipIdDataType('ModelManagerReceiver')
id.set_id([10, 20, 30, 40])

# Create a new ModelManagerReceiver Node
model_receiver_node = ModelManagerReceiverNode(
    id=id,
    data=data,
    domain=100)
```

(continues on next page)

(continued from previous page)

```
# Start execution
model_receiver_node.start(
    listener=CustomModelListener())

# Wait statistics
waiter.wait()

# ... do something ...

# Request the model
model_receiver_node.request_model(model_receiver_node.listener_.server_id)

# Stop execution
model_receiver_node.stop()
```

Model Manager Sender Node

This kind of Node performs the passive (server) action of *Collaborative Learning Scenario*. This node sends statistics about the models it manages. Then, waits for a Model request serialized as *Model Request Data Type*. Once received, a user-implemented callback (*fetch_model*) is executed, whose output should be the requested model in the form of a *Model Reply Data Type*.

Example of Usage

Steps

- Create the Id of the node.
- Create the statistics to be sent.
- Instantiate the ModelManagerSender Node creating an object of such class with the Id and statistics previously created.
- Start the execution of the node.
- Wait for a model request to arrive and be answered.
- Stop the execution of the node.

C++

```
#include <cpp_utils/wait/BooleanWaitHandler.hpp>

#include <amlip_cpp/types/id/AmlipIdDataType.hpp>
#include <amlip_cpp/types/model/ModelRequestDataType.hpp>
#include <amlip_cpp/types/model/ModelReplyDataType.hpp>
#include <amlip_cpp/types/model/ModelStatisticsDataType.hpp>

#include <amlip_cpp/node/collaborative_learning/ModelManagerSenderNode.hpp>


class CustomModelReplier : public eprosima::amlip::node::ModelReplier
{
```

(continues on next page)

(continued from previous page)

```
public:

    CustomModelReplier(
            const std::shared_ptr<eprosima::utils::event::BooleanWaitHandler>& waiter)
        : waiter_(waiter)
    {
        // Do nothing
    }

    virtual eprosima::amlip::types::ModelReplyDataType fetch_model (
            const eprosima::amlip::types::ModelRequestDataType data) override
    {
        std::cout << "Processing data: " << data << " . Processing data..." << std::endl;

        // Create new solution from data here
        eprosima::amlip::types::ModelReplyDataType solution("MOBILENET V1");

        std::cout << "Processed model: " << solution << " . Returning model..." <<␣
↪std::endl;

        waiter_->open();

        return solution;
    }

    std::shared_ptr<eprosima::utils::event::BooleanWaitHandler> waiter_;
};

// Create the Id of the node
eprosima::amlip::types::AmlipIdDataType id({"ModelManagerSender"}, {66, 66, 66, 66});

// Create ModelManagerSender Node
eprosima::amlip::node::ModelManagerSenderNode model_sender_node(id);

// Create statistics data
std::string data = "hello world";
model_sender_node.publish_statistics("v0", data);

// Create waiter
std::shared_ptr<eprosima::utils::event::BooleanWaitHandler> waiter =
    std::make_shared<eprosima::utils::event::BooleanWaitHandler>(false, true);

// Create listener to process requests and return replies
std::shared_ptr<CustomModelReplier> replier =
    std::make_shared<CustomModelReplier>(waiter);

// Start execution
model_sender_node.start(replier);

// Wait for the solution to be sent
waiter->wait();
```

(continues on next page)

(continued from previous page)

```
// Stop execution
model_sender_node.stop();
```

Python

```
from py_utils.wait.BooleanWaitHandler import BooleanWaitHandler

from amlip_py.types.AmlipIdDataType import AmlipIdDataType
from amlip_py.types.ModelRequestDataType import ModelRequestDataType
from amlip_py.types.ModelReplyDataType import ModelReplyDataType
from amlip_py.types.ModelStatisticsDataType import ModelStatisticsDataType

from amlip_py.node.ModelManagerSenderNode import ModelManagerSenderNode, ModelReplier


class CustomModelReplier(ModelReplier):

    def fetch_model(
            self,
            model: ModelRequestDataType) -> ModelReplyDataType:
        solution = ModelReplyDataType(model.to_string().upper())
        print(f'Model request received from client\n'
            f' model: {model.to_string()}\n'
            f' solution: {solution.to_string()}')

        waiter.open()

        return solution

# Create the Id of the node
id = AmlipIdDataType('ModelManagerSender')
id.set_id([66, 66, 66, 66])

# Create  a new ModelManagerSender Node
model_sender_node = ModelManagerSenderNode(
    id=id,
    domain=100)

model_sender_node.publish_statistics(
    'ModelManagerSenderStatistics',
    'hello world')

# Start execution
model_sender_node.start(
    listener=CustomModelReplier())

# Wait for the solution to be sent
waiter.wait()

# Stop execution
model_sender_node.stop()
```

3.15 AML-IP Tools

3.15.1 Agent Tool

This tool launches an *Agent Node*, which is the node in charge of communicating a local node or AML-IP cluster with the rest of the network in *WAN*s. It centralizes the *WAN* discovery and communication, i.e. it is the bridge for all the nodes in their *LAN*s with the rest of the AML-IP components.

Building the tool

If the tool package is not compiled, please refer to *Linux installation from sources using colcon* or run the command below.

```
colcon build --packages-up-to amlip_agent
```

Once AML-IP packages are installed and built, source the workspace using the following command.

```
source install/setup.bash
```

Application Arguments

The Agent Tool supports several input arguments:

Command	Option	Long option	Value	Default Value
Help	`-h`	`--help`		
Entity	`-e`	`--entity`	Agent Entity type.	`client`
Debug	`-d`	`--debug`	Enables the *AML-IP* logs so the execution can be followed by internal debugging information. Sets `Log Verbosity` to `info` and `Log Filter` to `AMLIP`.	
Log Verbosity	Set the verbosity level so only log messages with equal or higher importance level are shown.		`--log-verbosity`	`warning`
Log Filter	Set a regex string as filter.		`--log-filter`	`AMLIP`
Name	`-n`	`--name`	Readable File Path	`amlip_agent`
DDS Domain	`-d`	`--domain`	Configures the *Domain Id*.	`0`
Connection Address	`-c`	`--connection-address`	Address to connect.	`127.0.0.1`
Connection Port	`-p`	`--connection-port`	Address connection port.	`12121`
Listening Address	`-l`	`--listening-address`	Address where to listen.	`127.0.0.1`
Listening Port	`-q`	`--listening-port`	Address listening port.	`12121`
Transport	`-t`	`--transport`	Use only TCPv4 or UDPv4 transport.	`TCPv4`

Help Argument

It shows the usage information of the tool.

```
Usage: ./agent tool

General options:
-h, --help                                Produce help message.
-e, --entity <client|server|repeater> Agent Entity type (Default: client).
                                            Allowed options:
                                            • client -> Run an Agent Client Node.
                                            • server -> Run an Agent Server Node.
                                            • repeater -> Run an Agent Repeater Node.
```

(continues on next page)

(continued from previous page)

```
Debug options:
-d, --debug                             Set log verbosity to Info (Using this option with -
↪-log-filter and/or --log-verbosity will head to undefined behaviour).
    --log-filter                        Set a Regex Filter to filter by category the info␣
↪and warning log entries. (Default = "AMLIP").
    --log-verbosity <info|warning|error>
                                          Set a Log Verbosity Level higher or equal the␣
↪one given (Default: warning).

Client options:
-n, --name <name>                       Name (Default: amlip_agent).
-d, --domain <id>                       DDS domain ID (Default: 0).
-c, --connection-address <address>      Address to connect (Default: 127.0.0.1).
-p, --connection-port <num>             Address connection port (Default: 12121).
-t, --transport <tcp|udp>               Use only TCPv4 or UDPv4 transport. (Default:␣
↪TCPv4).

Server options:
-n, --name <name>                       Name (Default: amlip_agent).
-d, --domain <id>                       DDS domain ID (Default: 0).
-l, --listening-address <address>       Address where listen (Default: 127.0.0.1).
-q, --listening-port <num>              Address listening port (Default: 12121).
-t, --transport <tcp|udp>               Use only TCPv4 or UDPv4 transport. (Default:␣
↪TCPv4).

Repeater options:
-n, --name <name>                       Name (Default: amlip_agent).
-d, --domain <id>                       DDS domain ID (Default: 0).
-c, --connection-address <address>      Address to connect (Default: 127.0.0.1).
-l, --listening-address <address>       Address where listen (Default: 127.0.0.1).
-p, --connection-port <num>             Address connection port (Default: 12121).
-q, --listening-port <num>              Address listening port (Default: 12121).
-t, --transport <tcp|udp>               Use only TCPv4 or UDPv4 transport. (Default:␣
↪TCPv4).
```

Run tool

Source the following file to setup the AML-IP environment:

```
source <path-to-amlip-installation>/install/setup.bash
```

Launching an *Agent Client Node* instance is as easy as executing the following command:

```
amlip_agent -e client -c 87.111.115.111 -p 18000 -t tcp
```

To launch an *Agent Server Node*, execute:

```
amlip_agent -e server -l 87.111.115.111 -q 18000 -t tcp
```

To launch an *Agent Repeater Node*, execute:

```
amlip_agent -e repeater -l 87.111.115.111 -q 18000 -t tcp
```

Close tool

In order to stop the *Agent* tool, press `Ctrl + C` in the terminal where the process is running.

3.16 Linux installation from sources

3.16.1 Sub-packages

The *AML-IP* is constituted of several sub-packages. Depending on the use of those packages, some or all of them must be built. These are the packages of *AML-IP* and the dependencies between them:

Sub-package	Description	Depends on
`amlip_cpp`	Main C++ library with the implementation and API to create *AML-IP* Nodes.	
`amlip_swig`	Project to auto-generate a Python library from `amlip_cpp`.	`amlip_cpp`
`amlip_py`	Main Python library with API to create AML-IP Nodes.	`amlip_swig`
`amlip_docs`	Sphinx documentation project.	

3.16.2 Dependencies

These are the dependencies required in the system before building *AML-IP* from sources.

- *CMake, g++, pip, wget and git*
- *Asio and TinyXML2 libraries*
- *OpenSSL*
- *yaml-cpp*
- *SWIG*
- *Colcon* [optional]
- *Gtest* [for test only]
- *eProsima dependencies*

CMake, g++, pip, wget and git

These packages provide the tools required to install *AML-IP* and its dependencies from command line. Install CMake, g++, pip, wget and git using the package manager of the appropriate Linux distribution. For example, on Ubuntu use the command:

```
sudo apt install cmake g++ pip wget git
```

Asio and TinyXML2 libraries

Asio is a cross-platform C++ library for network and low-level I/O programming, which provides a consistent asynchronous model. TinyXML2 is a simple, small and efficient C++ XML parser. Install these libraries using the package manager of the appropriate Linux distribution. For example, on Ubuntu use the command:

```
sudo apt install libasio-dev libtinyxml2-dev
```

OpenSSL

OpenSSL is a robust toolkit for the TLS and SSL protocols and a general-purpose cryptography library. Install OpenSSL using the package manager of the appropriate Linux distribution. For example, on Ubuntu use the command:

```
sudo apt install libssl-dev
```

yaml-cpp

yaml-cpp is a YAML parser and emitter in C++ matching the YAML 1.2 spec, and is used by *DDS Router* application to parse the provided configuration files. Install yaml-cpp using the package manager of the appropriate Linux distribution. For example, on Ubuntu use the command:

```
sudo apt install libyaml-cpp-dev
```

SWIG

SWIG is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages. In this project, it is used to wrap `amlip_cpp` C++ API to generate a Python library. Install SWIG using the package manager of the appropriate Linux distribution. For example, on Ubuntu use the command:

```
sudo apt install swig
```

Colcon

Install the ROS 2 development tools (colcon and vcstool) by executing the following command:

```
pip3 install -U colcon-common-extensions vcstool
```

Note: If this fails due to an Environment Error, add the `--user` flag to the pip installation command.

Gtest

Gtest is a unit testing library for C++. For a detailed description of the Gtest installation process, please refer to the Gtest Installation Guide.

If using colcon, it is also possible to clone the Gtest Github repository into the *DDS Router* workspace and compile it as a dependency package. Add this new package to `.repos` file before importing the sources:

```
googletest-distribution:
    type: git
    url: https://github.com/google/googletest.git
    version: release-1.12.1
```

eProsima dependencies

These are the eProsima libraries required for building *AML-IP*:

- `foonathan_memory_vendor`, an STL compatible C++ memory allocation library.
- `fastcdr`, a C++ library that serializes according to the standard CDR serialization mechanism.
- `fastrtps`, the core library of eProsima Fast DDS library.
- `cmake_utils`, an eProsima utilities library for CMake.
- `cpp_utils`, an eProsima utilities library for C++.
- `ddsrouter_core`, the eProsima DDS Router library C++.

If they are already available in the system there is no need to build them again, just source them when building the *AML-IP*. If using CMake, add the already libraries installation paths to `LD_LIBRARY_PATH`. If using colcon, use the following command to source them:

```
source <eprosima-dependencies-installation-path>/install/setup.bash
```

3.16.3 Installation methods

There are two main possibilities to build *AML-IP* from sources in Linux. One of them uses CMake and the other colcon, an auto-build framework.

Note: Colcon version is advised for non advanced users as it is easier and neater.

Linux installation from sources using colcon

colcon is a command line tool based on CMake aimed at building sets of software packages in a tidy and easy way. The instructions for installing the *AML-IP* using colcon application from sources and its required dependencies are provided in this page.

Installation

Follow the instructions below to build *eProsima AML-IP*, after making sure all required dependencies are installed in your system (*Dependencies*).

Download eProsima dependencies

1. Create a `AML-IP` directory and download the `.repos` file that will be used to install *AML-IP* and its dependencies:

```
mkdir -p ~/AML-IP/src
cd ~/AML-IP
wget https://raw.githubusercontent.com/eProsima/AML-IP/main/amlip.repos
vcs import src < amlip.repos
```

Note: In case there are already some eProsima libraries installed in the system, it is not required to download and build every dependency in the `.repos` file, but just those projects that are not already in the system. Refer to section *eProsima dependencies* in order to check how to source those libraries.

Build packages

1. Build the packages:

```
colcon build --packages-up-to-regex amlip
```

Note: Not all the sub-packages of all the dependencies are required. In order to build only the packages required, use the colcon option `--packages-up-to <package-to-build>`. *e.g. the AML-IP C++ library is completely built using* `--packages-up-to amlip_cpp` For more details about the colcon available arguments, please refer to packages selection page of the colcon manual.

Note: Being based on CMake, it is possible to pass the CMake configuration options to the `colcon build` command. For more information on the specific syntax, please refer to the CMake specific arguments page of the colcon manual. For more details about the available CMake options, please refer to the *CMake options* section.

Run Tests

Tests are not automatically built within the *AML-IP* project. Use CMake option *BUILD_TESTS* when building the project in order to activate tests. This could also be done by a colcon.meta file to only activate tests in the desired packages.

1. Build the packages with tests:

```
colcon build --packages-select-regex amlip --cmake-args "-DBUILD_TESTS=ON"
```

2. Run tests. Use `--packages-select <package-name>` to only execute tests of a specific package:

```
colcon test --event-handlers=console_direct+ --packages-select amlip_cpp
```

Source installation

To source the installation of the previously built *AML-IP* (in order to use its tools or link against it), use the following command:

```
source install/setup.bash
```

Build demos

There is a demo sub-package that can be installed along with the project. In order to install this package use one of these 2 commands:

```
# To build all sub-packages
colcon build
```

```
# To only build demo package and its dependencies
colcon build --packages-up-to amlip_demo_nodes
```

Linux installation from sources using CMake

The instructions for installing the *AML-IP* using CMake from sources and its required dependencies are provided in this page. This section explains how to compile *AML-IP* with CMake, either *locally* or *globally*.

Local installation

1. Create a `AML-IP` directory where to download and build *AML-IP* and its dependencies:

   ```
   mkdir -p ~/AML-IP/src
   mkdir -p ~/AML-IP/build
   cd ~/AML-IP
   wget https://raw.githubusercontent.com/eProsima/AML-IP/main/amlip.repos
   vcs import src < amlip.repos
   ```

2. Compile all dependencies using CMake.

   - Foonathan memory

     ```
     cd ~/AML-IP
     mkdir build/foonathan_memory_vendor
     cd build/foonathan_memory_vendor
     cmake ~/AML-IP/src/foonathan_memory_vendor -DCMAKE_INSTALL_PREFIX=~/AML-
     ↪IP/install -DBUILD_SHARED_LIBS=ON
     cmake --build . --target install
     ```

   - Fast CDR

```
cd ~/AML-IP
mkdir build/fastcdr
cd build/fastcdr
cmake ~/AML-IP/src/fastcdr -DCMAKE_INSTALL_PREFIX=~/AML-IP/install
cmake --build . --target install
```

- Fast DDS

```
cd ~/AML-IP
mkdir build/fastdds
cd build/fastdds
cmake ~/AML-IP/src/fastdds -DCMAKE_INSTALL_PREFIX=~/AML-IP/install -
↪DCMAKE_PREFIX_PATH=~/AML-IP/install
cmake --build . --target install
```

- Dev Utils

```
# cmake_utils
cd ~/AML-IP
mkdir build/cmake_utils
cd build/cmake_utils
cmake ~/AML-IP/src/dev-utils/cmake_utils -DCMAKE_INSTALL_PREFIX=~/AML-IP/
↪install -DCMAKE_PREFIX_PATH=~/AML-IP/install
cmake --build . --target install

# cpp_utils
cd ~/AML-IP
mkdir build/cpp_utils
cd build/cpp_utils
cmake ~/AML-IP/src/dev-utils/cpp_utils -DCMAKE_INSTALL_PREFIX=~/AML-IP/
↪install -DCMAKE_PREFIX_PATH=~/AML-IP/install
cmake --build . --target install
```

- DDS Router

```
# ddsrouter_core
cd ~/AML-IP
mkdir build/ddsrouter_core
cd build/ddsrouter_core
cmake ~/AML-IP/src/ddsrouter/ddsrouter_core -DCMAKE_INSTALL_PREFIX=~/AML-
↪IP/install -DCMAKE_PREFIX_PATH=~/AML-IP/install
cmake --build . --target install
```

3. Once all dependencies are installed, install *AML-IP*:

```
# amlip_cpp
cd ~/AML-IP
mkdir build/amlip_cpp
cd build/amlip_cpp
cmake ~/AML-IP/src/amlip/amlip_cpp -DCMAKE_INSTALL_PREFIX=~/AML-IP/install -DCMAKE_
↪PREFIX_PATH=~/AML-IP/install
cmake --build . --target install
```

(continues on next page)

(continued from previous page)

```
# amlip_swig
cd ~/AML-IP
mkdir build/amlip_swig
cd build/amlip_swig
cmake ~/AML-IP/src/amlip/amlip_swig -DCMAKE_INSTALL_PREFIX=~/AML-IP/install -DCMAKE_
↪PREFIX_PATH=~/AML-IP/install
cmake --build . --target install

# amlip_py
cd ~/AML-IP
mkdir install/amlip_py
pip3 install --editable ~/AML-IP/src/amlip/amlip_py/ --target install/amlip_py
```

Note: By default, *AML-IP* does not compile tests. However, they can be activated by downloading and installing Gtest and building with CMake option `-DBUILD_TESTS=ON`.

Global installation

To install *AML-IP* system-wide instead of locally, remove all the flags that appear in the configuration steps of `Fast-CDR`, `Fast-DDS`, `DDS-Router`, and `AML-IP`, and change the first in the configuration step of `foonathan_memory_vendor` to the following:

```
-DCMAKE_INSTALL_PREFIX=/usr/local/ -DBUILD_SHARED_LIBS=ON
```

3.17 Windows installation from sources

3.17.1 Sub-packages

The *AML-IP* is constituted of several sub-packages. Depending on the use of those packages, some or all of them must be built. These are the packages of *AML-IP* and the dependencies between them:

Sub-package	Description	Depends on
`amlip_cpp`	Main C++ library with the implementation and API to create *AML-IP* Nodes.	
`amlip_swig`	Project to auto-generate a Python library from `amlip_cpp`.	`amlip_cpp`
`amlip_py`	Main Python library with API to create AML-IP Nodes.	`amlip_swig`
`amlip_docs`	Sphinx documentation project.	

3.17.2 Dependencies

The installation of *eProsima AML-IP* in a Windows environment from sources requires the following tools to be installed in the system:

- *Visual Studio*
- *Chocolatey*
- *CMake, pip3, wget and git*
- *Asio and TinyXML2 libraries*
- *OpenSSL*
- *yaml-cpp*
- *Colcon* [optional]
- *Gtest* [for test only]
- *eProsima dependencies*

Visual Studio

Visual Studio is required to have a C++ compiler in the system. For this purpose, make sure to check the `Desktop development with C++` option during the Visual Studio installation process.

If Visual Studio is already installed but the Visual C++ Redistributable packages are not, open Visual Studio and go to `Tools -> Get Tools and Features` and in the `Workloads` tab enable `Desktop development with C++`. Finally, click `Modify` at the bottom right.

Chocolatey

Chocolatey is a Windows package manager. It is needed to install some of *eProsima AML-IP*'s dependencies. Download and install it directly from the website.

CMake, pip3, wget and git

These packages provide the tools required to install *eProsima AML-IP* and its dependencies from command line. Download and install CMake, pip3, wget and git by following the instructions detailed in the respective websites. Once installed, add the path to the executables to the `PATH` from the *Edit the system environment variables* control panel.

Asio and TinyXML2 libraries

Asio is a cross-platform C++ library for network and low-level I/O programming, which provides a consistent asynchronous model. TinyXML2 is a simple, small and efficient C++ XML parser. They can be downloaded directly from the links below:

- Asio
- TinyXML2

After downloading these packages, open an administrative shell with *PowerShell* and execute the following command:

```
choco install -y -s <PATH_TO_DOWNLOADS> asio tinyxml2
```

where `<PATH_TO_DOWNLOADS>` is the folder into which the packages have been downloaded.

OpenSSL

OpenSSL is a robust toolkit for the TLS and SSL protocols and a general-purpose cryptography library. Download and install the latest OpenSSL version for Windows at this link. After installing, add the environment variable `OPENSSL_ROOT_DIR` pointing to the installation root directory.

For example:

```
OPENSSL_ROOT_DIR=C:\Program Files\OpenSSL-Win64
```

yaml-cpp

yaml-cpp is a YAML parser and emitter in C++ matching the YAML 1.2 spec, and is used by *DDS Router* application to parse the provided configuration files. From an administrative shell with *PowerShell*, execute the following commands in order to download and install yaml-cpp for Windows:

```
git clone --branch yaml-cpp-0.7.0 https://github.com/jbeder/yaml-cpp
cmake -DCMAKE_INSTALL_PREFIX='C:\Program Files\yamlcpp' -B build\yamlcpp yaml-cpp
cmake --build build\yamlcpp --target install    # If building in Debug mode, add --
↪config Debug
```

SWIG

SWIG is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages. In this project, it is used to wrap `amlip_cpp` C++ API to generate a Python library. Install SWIG using the package manager of the appropriate Linux distribution. For example, on Ubuntu use the command:

```
choco install swig --allow-downgrade --version=4.0.2.04082020
```

Colcon

colcon is a command line tool based on CMake aimed at building sets of software packages. Install the ROS 2 development tools (colcon and vcstool) by executing the following command:

```
pip3 install -U colcon-common-extensions vcstool
```

Note: If this fails due to an Environment Error, add the `--user` flag to the `pip3` installation command.

Gtest

Gtest is a unit testing library for C++. By default, *DDS Router* does not compile tests. It is possible to activate them with the opportune CMake options when calling colcon or CMake. For more details, please refer to the *CMake options* section.

Run the following commands on your workspace to install Gtest.

```
git clone https://github.com/google/googletest.git
cmake -DCMAKE_INSTALL_PREFIX='C:\Program Files\gtest' -Dgtest_force_shared_crt=ON -
↪DBUILD_GMOCK=ON ^
    -B build\gtest -A x64 -T host=x64 googletest
cmake --build build\gtest --config Release --target install
```

or refer to the Gtest Installation Guide for a detailed description of the Gtest installation process.

eProsima dependencies

These are the eProsima libraries required for building *AML-IP*:

- `foonathan_memory_vendor`, an STL compatible C++ memory allocation library.
- `fastcdr`, a C++ library that serializes according to the standard CDR serialization mechanism.
- `fastrtps`, the core library of eProsima Fast DDS library.
- `cmake_utils`, an eProsima utilities library for CMake.
- `cpp_utils`, an eProsima utilities library for C++.
- `ddsrouter_core`, the eProsima DDS Router library C++.

If it already exists in the system an installation of these libraries there is no need to build them again, just source them when building the *AML-IP*. If using CMake, add the already libraries installation paths to `PATH`. If using colcon, use the following command to source them:

```
source <eprosima-dependencies-installation-path>/install/setup.bash
```

3.17.3 Installation methods

There are two main possibilities to build *AML-IP* from sources in Windows. One of them uses CMake and the other colcon, an auto-build framework.

Note: Colcon version is advised for non advanced users as it is easier and neater.

Windows installation from sources using colcon

colcon is a command line tool based on CMake aimed at building sets of software packages in a tidy and easy way. The instructions for installing the *AML-IP* using colcon application from sources and its required dependencies are provided in this page.

Installation

This section explains how to install *AML-IP* using colcon.

Download eProsima dependencies

1. Create a `AML-IP` directory and download the `.repos` file that will be used to install *AML-IP* and its dependencies:

```
mkdir <path\to\user\workspace>\AML-IP
cd <path\to\user\workspace>\AML-IP
mkdir src
wget https://raw.githubusercontent.com/eProsima/AML-IP/main/amlip.repos
vcs import src < amlip.repos
```

Note: In case there are already some eProsima libraries installed in the system, it is not required to download and build every dependency in the `.repos` file, but just those projects that are not already in the system. Refer to section *eProsima dependencies* in order to check how to source those libraries.

Build packages

1. Build the packages:

```
colcon build --packages-up-to-regex amlip
```

Note: Not all the sub-packages of all the dependencies are required. In order to build only the packages required, use the colcon option `--packages-up-to <package-to-build>`. *e.g. the AML-IP C++ library is completely built using* `--packages-up-to amlip_cpp` For more details about the colcon available arguments, please refer to packages selection page of the colcon manual.

Note: Being based on CMake, it is possible to pass the CMake configuration options to the `colcon build` command. For more information on the specific syntax, please refer to the CMake specific arguments page of the colcon manual. For more details about the available CMake options, please refer to the *CMake options* section.

Run Tests

Tests are not automatically built within the *AML-IP* project. Use CMake option *BUILD_TESTS* when building the project in order to activate tests. This could also be done by a colcon.meta file to only activate tests in the desired packages.

1. Build the packages with tests:

```
colcon build --packages-select-regex amlip --cmake-args "-DBUILD_TESTS=ON"
```

2. Run tests. Use `--packages-select <package-name>` to only execute tests of a specific package:

```
colcon test --event-handlers=console_direct+ --packages-select amlip_cpp
```

Source installation

To source the installation of the *AML-IP* previously built in order to link it to another built or to use its tools, use the following command:

- Command prompt:

```
install/setup.bat
```

- PowerShell:

```
install/setup.ps1
```

Build demos

There is a demo sub-package that can be installed along with the project. In order to install this package use one of these 2 commands:

```
# To build all sub-packages
colcon build
```

```
# To only build demo package and its dependencies
colcon build --packages-up-to amlip_demo_nodes
```

Windows installation from sources using CMake

The instructions for installing the *AML-IP* using CMake from sources and its required dependencies are provided in this page. This section explains how to compile *AML-IP* with CMake, either *locally* or *globally*.

Local installation

1. Open a command prompt, and create a `AML-IP` directory where to download and build *AML-IP* and its dependencies:

   ```
   mkdir <path\to\user\workspace>\AML-IP
   mkdir <path\to\user\workspace>\AML-IP\src
   mkdir <path\to\user\workspace>\AML-IP\build
   cd <path\to\user\workspace>\AML-IP
   wget https://raw.githubusercontent.com/eProsima/AML-IP/main/amlip.repos
   vcs import src < amlip.repos
   ```

2. Compile all dependencies using CMake.

   - Foonathan memory

```
cd <path\to\user\workspace>\AML-IP
mkdir build\foonathan_memory_vendor
cd build\foonathan_memory_vendor
cmake <path\to\user\workspace>\AML-IP\src\foonathan_memory_vendor -
↪DCMAKE_INSTALL_PREFIX=<path\to\user\workspace>\AML-IP\install ^
    -DBUILD_SHARED_LIBS=ON
cmake --build . --config Release --target install
```

- Fast CDR

```
cd <path\to\user\workspace>\AML-IP
mkdir build\fastcdr
cd build\fastcdr
cmake <path\to\user\workspace>\AML-IP\src\fastcdr -DCMAKE_INSTALL_PREFIX=
↪<path\to\user\workspace>\AML-IP\install
cmake --build . --config Release --target install
```

- Fast DDS

```
cd <path\to\user\workspace>\AML-IP
mkdir build\fastdds
cd build\fastdds
cmake <path\to\user\workspace>\AML-IP\src\fastdds -DCMAKE_INSTALL_PREFIX=
↪<path\to\user\workspace>\AML-IP\install ^
    -DCMAKE_PREFIX_PATH=<path\to\user\workspace>\AML-IP\install
cmake --build . --config Release --target install
```

- Dev Utils

```
# cmake_utils
cd <path\to\user\workspace>\AML-IP
mkdir build\cmake_utils
cd build\cmake_utils
cmake <path\to\user\workspace>\AML-IP\src\dev-utils\cmake_utils -DCMAKE_INSTALL_
↪PREFIX=<path\to\user\workspace>\AML-IP\install ^
    -DCMAKE_PREFIX_PATH=<path\to\user\workspace>\AML-IP\install
cmake --build . --config Release --target install

# cpp_utils
cd <path\to\user\workspace>\AML-IP
mkdir build\cpp_utils
cd build\cpp_utils
cmake <path\to\user\workspace>\AML-IP\src\dev-utils\cpp_utils -DCMAKE_INSTALL_
↪PREFIX=<path\to\user\workspace>\AML-IP\install ^
    -DCMAKE_PREFIX_PATH=<path\to\user\workspace>\AML-IP\install
cmake --build . --config Release --target install
```

- DDS Router

```
# ddsrouter_core
cd <path\to\user\workspace>\AML-IP
mkdir build\ddsrouter_core
cd build\ddsrouter_core
```

(continues on next page)

(continued from previous page)

```
cmake <path\to\user\workspace>\AML-IP\src\ddsrouter\ddsrouter_core -
↪DCMAKE_INSTALL_PREFIX=<path\to\user\workspace>\AML-IP\install ^
    -DCMAKE_PREFIX_PATH=<path\to\user\workspace>\AML-IP\install
cmake --build . --config Release --target install
```

3. Once all dependencies are installed, install *AML-IP*:

```
# amlip_cpp
cd ~/AML-IP
mkdir build/amlip_cpp
cd build/amlip_cpp
cmake ~/AML-IP/src/amlip/amlip_cpp -DCMAKE_INSTALL_PREFIX=~/AML-IP/install -DCMAKE_
↪PREFIX_PATH=~/AML-IP/install
cmake --build . --target install

# amlip_swig
cd ~/AML-IP
mkdir build/amlip_swig
cd build/amlip_swig
cmake ~/AML-IP/src/amlip/amlip_swig -DCMAKE_INSTALL_PREFIX=~/AML-IP/install -DCMAKE_
↪PREFIX_PATH=~/AML-IP/install
cmake --build . --target install

# amlip_py
cd ~/AML-IP
mkdir install/amlip_py
pip3 install --editable ~/AML-IP/src/amlip/amlip_py/ --target install/amlip_py
```

Note: By default, *AML-IP* does not compile tests. However, they can be activated by downloading and installing Gtest and building with CMake option `-DBUILD_TESTS=ON`.

Global installation

To install *AML-IP* system-wide instead of locally, remove all the flags that appear in the configuration steps of `Fast-CDR`, `Fast-DDS`, `AML-IP`, and `AML-IP`, and change the first in the configuration step of `foonathan_memory_vendor` to the following:

```
-DCMAKE_INSTALL_PREFIX=/usr/local/ -DBUILD_SHARED_LIBS=ON
```

3.18 CMake options

eProsima AML-IP provides numerous CMake options for changing the behavior and configuration of *AML-IP* packages built with CMake. These options allow the developer to enable/disable certain *AML-IP* settings by defining these options to `ON`/`OFF` or a string value at the CMake execution.

Warning: These options are only for advanced developers who installed *eProsima AML-IP* from sources.

Option	Description	Possible values	Default
CMAKE_BUILD_TYPE	CMake optimization build type.	Release Debug	Release
BUILD_ALL	Build all *AML-IP* sub-packages. Setting to ON sets to ON BUILD_TOOL, BUILD_LIBRARY, and BUILD_DOCS.	OFF ON	OFF
BUILD_LIBRARY	Build the *AML-IP* libraries sub-packages. It is set to ON if BUILD_ALL is set to ON.	OFF ON	ON
BUILD_TOOL	Build the *AML-IP* tools sub-packages. It is set to ON if BUILD_ALL is set to ON.	OFF ON	ON
BUILD_DOCS	Build the *AML-IP* documentation sub-packages. It is set to ON if BUILD_ALL is set to ON.	OFF ON	OFF
BUILD_TESTS	Build the *AML-IP* application and documentation tests. Setting BUILD_TESTS to ON sets BUILD_ALL, BUILD_LIBRARY_TESTS, BUILD_TOOL_TESTS, and BUILD_DOCS_TESTS to ON.	OFF ON	OFF
BUILD_LIBRARY_TESTS	Build the *AML-IP* library tests. It is set to ON if BUILD_TESTS is set to ON.	OFF ON	OFF
BUILD_TOOL_TESTS	Build the *AML-IP* application tests. It is set to ON if BUILD_TESTS is set to ON.	OFF ON	OFF
BUILD_DOCS_TESTS	Build the *AML-IP* documentation tests. It is set to ON if BUILD_TESTS is set to ON.	OFF ON	OFF
BUILD_MANUAL_TESTS	Build the *AML-IP* manual tests only if BUILD_TESTS is set to ON.	OFF ON	OFF
LOG_INFO	Activate *AML-IP* execution logs. It is set to ON if CMAKE_BUILD_TYPE is set to Debug.	OFF ON	ON if Debug OFF otherwise
ASAN_BUILD	Activate address sanitizer build.	OFF ON	OFF
TSAN_BUILD	Activate thread sanitizer build.	OFF ON	OFF

3.19 Enabling technologies

This page describes the different technologies that support the development of the *AML-IP*. These focus on the communication between nodes, the protocols used to support such communication and the libraries and tools used to handle the different types of data to be transmitted.

3.19.1 DDS (Data Distribution Service)

DDS is a distributed dynamic real-time middleware protocol based on a specification defined by the *OMG*. It relies on the underlying *RTPS* wire protocol.

AML-IP framework relies on *DDS* communication protocol to connect and communicate each of its *Nodes*. *DDS* protocol support *publications* and *subscriptions* in different *Topics* in order to create a distributed network of entities where communication takes place peer-to-peer, avoiding centralized systems and creating an homogeneous and standalone network. *DDS* relies on *QoS* to configure different characteristics for each of the communication channels, allowing to create really dynamic and complex networks.

Fast DDS

AML-IP uses *eProsima Fast DDS*, a C++ open-source library that implements *DDS* specification. *eProsima Fast DDS* has all the features and characteristics needed to power *AML-IP* communications. A whole documentation for the *Fast DDS* project can be found in Fast DDS Documentation.

3.20 Internal Protocols

This page briefly describes the protocols developed on top of DDS that enable AML-IP node communications as well as the deployment of the different *scenarios* presented in the user manual of this documentation.

3.20.1 DDS Topics

All the *AML-IP* Topics in the DDS *AML-IP* network have a previous name mangling. So an *AML-IP* topic named , `some_topic` would actually be named `amlip::some_topic` in the underneath DDS network.

3.20.2 MultiService over DDS

A new communication protocol based on *DDS* has been designed in order to fulfill the necessity of distributing a task in a network. The idea is a Service protocol based on Client-Server communication where multiple servers could be available at the same time in the same network. This protocol creates an auto-regulated orchestration method where a task could be distributed to **one and only one** server that is publicly available, and each server receives no more than one task at a time.

3.20.3 RPC over DDS

The Remote Procedure Call based on *DDS* has been implemented to meet the need to distribute requests and replies across a network.

Unlike how it is implemented in ROS 2 (with Fast DDS as middleware), topic mangling is used for the communication between the servers and clients.

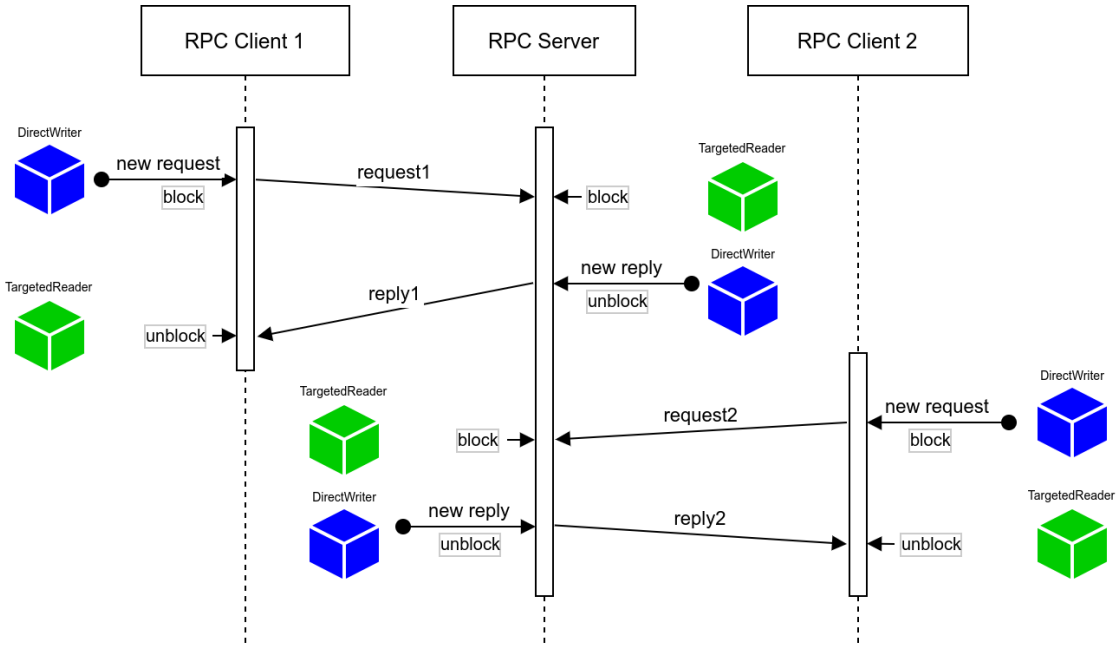
The following diagram illustrates the flow of the implementation:

3.20.4 DDS Entities properties

Every *AML-IP* entity within the *AML-IP* network is associated with predefined properties that encompass the entity's identification and metadata.

- The `fastdds.application.metadata` property is a `JSON` object that provides detailed information about the entity:
  - Internal: Specifies the name of the node.
  - Entity: expound the DDS entity.
  - Topic: define the topic name.
- The `fastdds.application.id` property uniquely identifies the DDS application to which the entity belongs, in this case `AML_IP`.

For a practical illustration, consider a *Writer* in a *TestNode* publishing on the */test* topic. The corresponding C++ code snippet for configuring the DataWriter QoS properties is as follows:




```
nlohmann::json property_value;

property_value["Internal"] = "TestNode";
property_value["Entity"] = "Writer";
property_value["Topic"] = "/test";

eprosima::fastdds::dds::DataWriterQos qos_request_availability_writer_ = default_request_
↪availability_writer_qos_();

qos_request_availability_writer_.properties().properties().emplace_back("fastdds.
↪application.metadata",
        property_value.dump(), true);

qos_request_availability_writer_.properties().properties().emplace_back("fastdds.
↪application.id",
        "AML_IP", true);
```

To retrieve the QoS, the following code can be used:

```
const std::string* application_id =
            eprosima::fastrtps::rtps::PropertyPolicyHelper::find_property(
        datareader_locked->get_qos().properties(), "fastdds.application.id");
```

3.21 Version v0.2.0

This release adds new **features**:

- Support Fast CDR v2.2.0.
- Implement asynchronous request model in `ModelManagerReceiver`.
- Add `fastdds.application.id` property to participants and endpoints.
- Add `fastdds.application.metadata` property to participants and endpoints.
- Add `MainNode` costructor with `domain` parameter in Python bindings.
- Add Python bindings for Agent Nodes.
- Rename `agent_tool` package to `amlip_agent`.

This includes the following **Bugfixes**:

- Update `AsyncComputingNode` and `AsyncInferenceNode` to only stop if the current state is running and only run if the current state is stopped.
- ASAN (Address Sanitizer) fixes.
- Fix allowlist namespacing in `AgentNode`.
- Call change status in `AsyncComputingNode`.

This release includes the following **CI improvements**:

- Migrate CI actions to use eProsima-CI.
- Include branch environment variables in CI configuration.

This release add new **Documentation features**:

- Add instructions to build the Docker image.
- Add `Agent Tool` section.
- Add `Enabling Technologies` and `Internal Protocols` sections.

This release includes the following *Dependencies Update*:

	Repository	Old Version	New Version
Foonathan Memory Vendor	eProsima/foonathan_memory_vendor	v1.3.1	v1.3.1
Fast CDR	eProsima/Fast-CDR	v1.1.0	v2.2.1
Fast DDS	eProsima/Fast-DDS	v2.11.0	v2.14.0
Dev Utils	eProsima/dev-utils	v0.4.0	v0.6.0
DDS Pipe	eProsima/DDS-Pipe	v0.2.0	v0.4.0
DDS Router	eProsima/DDS-Router	v2.0.0	v2.2.0

3.22 Previous Versions

3.22.1 Version v0.1.0

This is the first release of eProsima *AML-IP* (Algebraic Machine Learning - Integrating Platform).

This release includes the following **User Interface features**:

- C++ API
- Python API
- Add implementation of DDS entities.
- Add implementation of Multiservice protocol.
- Add implementation of Asynchronous Multiservice protocol.
- New internal package `amlip_demo_nodes` to include the demos packages.
- Dockerfile for creating a docker image with AML-IP.
- Add Custom RPC communication over DDS.

This release supports the following **Deployment Scenarios**:

- *Monitor State*
- *Workload Distribution*
- *Collaborative Learning*
- *Distributed Inference*

This release includes the following new **AML-IP Nodes**:

- *Status*: node that listens to other nodes status.
- *Main*: node that sends training data and collects the solution to that data.
- *Computing*: node that waits for training data and retrieves a solution.
- *Edge*: node that sends data and waits for the inferred solution.
- *Inference*: node that waits for data and retrieves an inference.
- ***Agent*: node in charge of the communication with the network.**
  - *Client*
  - *Server*
  - *Repeater*
- *Model Manager Receiver*: node that receives statistical data from models and sends requests to those models.
- *Model Manager Sender*: node that sends statistical data from models, receives requests and sends replies to those requests.

This release includes the following new **AML-IP Data Types**:

- *Status*: status messages sent by each node with its id, type and current state.
- *Job*: messages that represent training data.
- *Job Solution*: messages that represent the solution for a given set of training data.
- *Inference*: messages that represent a partial data-set.

- *Inference Solution*: messages that represent the inference of a data-set.
- *Model Statistics*: messages that represent statistical data from models.
- *Model*: messages that represent a problem model request.
- *Model Solution*: messages that represent a problem reply with the requested model.

This release includes the following **Demos**:

- *Collaborative Learning*
- *TensorFlow Inference*
- *TensorFlow Inference using ROSbot 2R*
- *Workload Distribution*

This release includes the following **Documentation features**:

- This same Documentation
- API Code Documentation

This release includes the following **Continuous Integration features**:

- Continuous Integration deployment in GitHub Actions.
- Compile with *-Wall* flag *Clang* job.
- Add *Address Sanitizer* check to all tests.
- Add Python Liner test to the Python API.
- Disable Data Sharing from test.

3.23 Glossary

API **Application Programming Interface**

CI **Continuous Integration**

OOP **Object Oriented Programming**

OS **Operating System**

3.23.1 AML-IP

Action Each of the steps in which any calculation or communication process is divided.

AML-IP **Algebraic Machine Learning - Integrating Platform**

Scenario Set of *Actions* that performs a whole, independent and self-contained behavior inside an *AML* network. More information in *User Manual section*.

Node Independent and stand-alone piece of software that performs different *Actions*. Each Node belongs to one and only one *Scenario*. More information in *User Manual section*.

3.23.2 AML

AML **Algebraic Machine Learning**

Atomization Specific state of an *AML* model.

ML **Machine Learning**

3.23.3 DDS

DDS **Data Distribution Service** protocol. Specification: https://www.omg.org/spec/DDS/. More information in *User Manual section*.

Domain Id Virtual partition for DDS networks.

Endpoint Individual Entity that can *Subscribe* or *Publish* in a specific *Topic*.

Publish To send a data or message to all the entities in the network subscribed to the same *Topic* in which the data is being published.

RTPS **Real-time Publish-Subscribe** protocol https://www.omg.org/spec/DDSI-RTPS/.

Subscribe To connect to a specific *Topic* and to receive messages published in such topic.

Topic An abstract channel of communication that connects *Publishers* that *Publish* and *Subscribers* that *Subscribe*.

OMG **Object Management Group** https://www.omg.org/.

QoS **Quality of Service**. Configurations of *Topic* and *Endpoint* that allow to specify the communication behavior. This allows to create *reliable* or *best-effort* communication channels, to determine the life of a data sent, to set internal configurations, etc.

3.23.4 Networking

IP

- **Internet Protocol**

LAN **Local Area Network**

NAT **Network Address Translation**: Typically an internet router multiplexes all the traffic through a public IP to several private IPs. Usually, the machines under the router network cannot be accessed from the outside unless a Port is forwarded in the router configuration, or if such host has previously started a TCP communication with the message source.

P2P **Peer to Peer**

TCP **Transmission Control Protocol**

UDP **User Datagram Protocol**

URL **Uniform Resource Locator**

WAN **Wide Area Network**

INDEX