ERDOS Documentation

Release 0.2.0

The ERDOS Team

Jul 22, 2020

Getting Started

ERDOS is a platform for developing self-driving cars and robotics applications.

View the codebase on GitHub.

CHAPTER 1

Example Use

```
# Construct the dataflow graph in the driver.
# The dataflow graph consists of operators which process data,
# and streams which broadcast messages to other operators.
def driver():
    # Create a camera operator which generates a stream of RGB images.
    camera_stream = erdos.connect(CameraOp)
    # Create an object detection operator with the provided model.
    # The object detection operator reads RGB images and sends bounding boxes.
    bounding_box_stream = erdos.connect(ObjectDetectorOp, [camera_stream],
                                        model="models/ssd_mobilenet_v1_coco")
    # Create semantic segmentation operator with the provided model
    segmentation_stream = erdos.connect(SegmentationOp, [camera_stream],
                                        model="models/drn_d_22_cityscapes")
    # Create an action operator to propose actions from provided features
    action_op = erdos.connect(ActionOp, [bounding_box_stream, segmentation_stream])
    # Create a robot operator to interface with the robot
    erdos.connect(RobotOp)

if __name__ == "__main__":
    # Execute the application
    erdos.run(driver)
```

1.1 Operators

Operators process data in ERDOS applications.

Operators receive messages from streams passed to the *connect* static method. Operators also create streams on which they send messages. These streams must created and returned by the *connect* method. ERDOS expects developers to specify which streams operators read from and write to. For more details, see the data streams documentation.

All operators must implement *erdos.Operator* abstract class.

Operators set up state in the *__init__* method. Operators should also add callbacks to streams in *__init__*.

Implement execution logic by overriding the *run* method. This method may contain a control loop or call methods that run regularly. *Callbacks are not invoked while run executes.*

1.1.1 API

1.1.2 Examples

Periodically Publishing Data

```
class SendOp(erdos.Operator):
    def __init__(self, write_stream):
        self.write_stream = write_stream

    @staticmethod
    def connect():
        return [erdos.WriteStream()]

    def run(self):
        count = 0
        while True:
            msg = erdos.Message(erdos.Timestamp(coordinates=[count]), count)
            print("SendOp: sending {msg}".format(msg=msg))
            self.write_stream.send(msg)

            count += 1
            time.sleep(1)
```

Processing Data via Callbacks

```
class CallbackOp(erdos.Operator):
    def __init__(self, read_stream):
        print("initializing  op")
        read_stream.add_callback(CallbackOp.callback)

    @staticmethod
    def callback(msg):
        print("CallbackOp: received {msg}".format(msg=msg))

    @staticmethod
    def connect(read_streams):
        return []
```

Processing Data by Pulling Messages

```
class PullOp:
    def __init__(self, read_stream):
        self.read_stream = read_stream

    @staticmethod
    def connect(read_streams):
        return []
```

(continues on next page)

(continued from previous page)

```
    def run(self):
        while True:
            data = self.read_stream.read()
            print("PullOp: received {data}".format(data=data))
```

1.2 Streams

Streams are used to send messages in ERDOS applications.

Operators send and read messages from streams to communicate with other operators and ingest/extract data from the system.

ERDOS streams are similar to ROS topics, but have few additional desirable properties. Streams facilitate one-to-many communication, so only 1 operator sends messages on a stream. ERDOS broadcasts messages sent on a stream to all connected operators. In addition, streams are typed when using the Rust API.

Use the following interfaces to send and receive data from streams: Read Stream, Write Stream, Ingest Stream, and Extract Stream.

1.2.1 Sending Messages

Operators use Write Streams to send data.

1.2.2 Receiving Messages

Operators receive data by registering callbacks or manually reading messages from Read Streams.

Callbacks are functions which take an ERDOS message and any necessary write streams as arguments. Generally, callbacks process received messages and publish the results on write streams.

1.2.3 Ingesting and Extracting Data

Some applications have trouble placing all of the data processing logic inside operators. For these applications, ERDOS provides special stream interfaces to *ingest* and *extract* data.

A comprehensive example is available here.

1.2.4 Loops

Certain applications require feedback in the dataflow. To support this use case, ERDOS provides the LoopStream interface to support loops in the dataflow.

A comprehensive example is available here.

1.3 Messages

ERDOS applications send data on streams via messages. Messages wrap data and provide timestamp information used to resolve control loops and track data flow through the system.

Timestamps consist of an array of coordinates. Timestamp semantics are user-defined for now; however, we may eventually formalize their use in the future in order to provide more advanced features in order to scale up stateful operators. Generally, the 0th coordinate is used to track message's sequence number and subsequent coordinates track the message's progress in cyclic data flows.