ERDOS Documentation

Release 0.3.1

The ERDOS Team

Sep 25, 2020

OVERVIEW

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

The system is built using techniques from streaming dataflow systems which is reflected by the API. Applications are modeled as directed graphs, in which data flows through *streams* and is processed by *operators*. Because applications often resemble a sequence of connected operators, an ERDOS application may also be referred to as a *pipeline*.

CHAPTER

ONE

EXAMPLE

The following example demonstrates a toy robotics application which uses semantic segmentation and the bounding boxes of detected objects to control a robot. The example consists of the driver part of the program, which is responsible for connecting operators via streams.

```
# Create a camera operator which generates a stream of RGB images.
camera_frames = erdos.connect(CameraOp)

# Connect an object detection operator which uses the provided model to
# detect objects and compute bounding boxes.
bounding_boxes = erdos.connect(ObjectDetectorOp, erdos.OperatorConfig(),
                               [camera_frames],
                               model="models/ssd_mobilenet_v1_coco")
# Connect semantic segmentation operator to the camera which computes the
# semantic segmentation for each image.
segmentation = erdos.connect(SegmentationOp, [camera_frames],
                             erdos.OperatorConfig(),
                             model="models/drn_d_22_cityscapes")

# Connect an action operator to propose actions from provided features.
actions = erdos.connect(ActionOp, erdos.OperatorConfig(),
                        [bounding_boxes, segmentation])
# Create a robot operator which interfaces with the robot to apply actions.
erdos.connect(RobotOp, erdos.OperatorConfig(), [actions])

# Execute the application.
erdos.run()
```

Further examples are available on GitHub

For information on building operators, see *§ Operators*.

CHAPTER

TWO

DRIVER

The driver section of the program connects operators together using streams to build an ERDOS application which may then be executed. The driver is typically the main section of the program.

The driver may also interact with a running ERDOS application. Using the `IngestStream`, the driver can send data to operators on a stream. The `ExtractStream` allows the driver to read data sent from an operator.

CHAPTER

THREE

DETERMINISM

ERDOS provides mechanisms to enable the building of deterministic applications. For instance, processing sets of messages separated by watermarks using watermark callbacks and the Rust time-versioned state data structure turns ERDOS pipelines into Kahn process networks.

For more information, see `WatermarkMessage` and `erdos.add_watermark_callback()`.

CHAPTER

FOUR

PERFORMANCE

ERDOS is designed for low latency. Self-driving car pipelines require end-to-end deadlines on the order of hundreds of milliseconds for safe driving. Similarly, self-driving cars typically process gigabytes per second of data on small clusters. Therefore, ERDOS is optimized to send small amounts of data (gigabytes as opposed to terabytes) as quickly as possible.

For performance-sensitive applications, it is recommended to use the Rust API as Python introduces significant overheads (e.g. serialization and reduced parallelism from the GIL).

View the codebase on GitHub.

You can export the dataflow graph as a DOT file by setting the `graph_filename` argument in `erdos.run()`.

4.1 What is ERDOS?

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

The system is built using techniques from streaming dataflow systems which is reflected by the API. Applications are modeled as directed graphs, in which data flows through *streams* and is processed by *operators*. Because applications often resemble a sequence of connected operators, an ERDOS application may also be referred to as a *pipeline*.

4.1.1 Example

The following example demonstrates a toy robotics application which uses semantic segmentation and the bounding boxes of detected objects to control a robot. The example consists of the driver part of the program, which is responsible for connecting operators via streams.

```
# Create a camera operator which generates a stream of RGB images.
camera_frames = erdos.connect(CameraOp)

# Connect an object detection operator which uses the provided model to
# detect objects and compute bounding boxes.
bounding_boxes = erdos.connect(ObjectDetectorOp, erdos.OperatorConfig(),
                               [camera_frames],
                               model="models/ssd_mobilenet_v1_coco")
# Connect semantic segmentation operator to the camera which computes the
# semantic segmentation for each image.
segmentation = erdos.connect(SegmentationOp, [camera_frames],
                             erdos.OperatorConfig(),
                             model="models/drn_d_22_cityscapes")

# Connect an action operator to propose actions from provided features.
```

(continues on next page)

(continued from previous page)

```
actions = erdos.connect(ActionOp, erdos.OperatorConfig(),
                        [bounding_boxes, segmentation])
# Create a robot operator which interfaces with the robot to apply actions.
erdos.connect(RobotOp, erdos.OperatorConfig(), [actions])

# Execute the application.
erdos.run()
```

Further examples are available on GitHub

For information on building operators, see *§ Operators*.

4.1.2 Driver

The driver section of the program connects operators together using streams to build an ERDOS application which may then be executed. The driver is typically the main section of the program.

The driver may also interact with a running ERDOS application. Using the `IngestStream`, the driver can send data to operators on a stream. The `ExtractStream` allows the driver to read data sent from an operator.

4.1.3 Determinism

ERDOS provides mechanisms to enable the building of deterministic applications. For instance, processing sets of messages separated by watermarks using watermark callbacks and the Rust time-versioned state data structure turns ERDOS pipelines into Kahn process networks.

For more information, see `WatermarkMessage` and `erdos.add_watermark_callback()`.

4.1.4 Performance

ERDOS is designed for low latency. Self-driving car pipelines require end-to-end deadlines on the order of hundreds of milliseconds for safe driving. Similarly, self-driving cars typically process gigabytes per second of data on small clusters. Therefore, ERDOS is optimized to send small amounts of data (gigabytes as opposed to terabytes) as quickly as possible.

For performance-sensitive applications, it is recommended to use the Rust API as Python introduces significant overheads (e.g. serialization and reduced parallelism from the GIL).

View the codebase on GitHub.

You can export the dataflow graph as a DOT file by setting the `graph_filename` argument in `erdos.run()`.

4.2 Streams

Streams are used to send data in ERDOS applications.

ERDOS streams are similar to ROS topics, but have a 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.

Streams expose 2 classes of interfaces that access the underlying stream:

1. Read-interfaces expose methods to receive and process data. They allow pulling data by calling `read()` and `try_read()`. Often, they also support a push data model accessed by registering callbacks (e.g. `add_callback` and `add_watermark_callback`). Structures that implement read interfaces include:

- `ReadStream`: used by operators to read data and register callbacks.
- `ExtractStream`: used by the driver to read data.

1. Write-interfaces expose the send method to send data on a stream. Structures that implement write interfaces include:

- `WriteStream`: used by operators to send data.
- `IngestStream`: used by the driver to send data.

Some applications may want to introduce loops in their dataflow graphs which is possible using the `LoopStream`.

4.2.1 Sending Messages

Operators use Write Streams to send data.

4.2.2 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.

4.2.3 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.

4.2.4 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.

4.3 Operators

An ERDOS operator receives data on `ReadStreams`, and sends processed data on `WriteStreams`. We provide a standard library of operators for common dataflow patterns under `erdos.operators`. While the standard operators are general and versatile, some applications may implement custom operators to better optimize performance and take fine-grained control over exection.

All operators must inherit from the `Operator` base class and implement `__init__()` and `connect()` methods.

- `__init__()` takes all `ReadStreams` from which the operator receives data, all `WriteStreams` on which the operator sends data, and any other arguments passed when calling `connect()`. Within `__init__()`, the state should be initialized, and callbacks may be registered across `ReadStreams`.

- The `connect()` method takes `ReadStreams` and returns `WriteStreams` which are all later passed to `__init__()` by ERDOS. The `ReadStreams` and `WriteStreams` must appear in the same order as in `__init__()`.

While ERDOS manages the execution of callbacks, some operators require more finegrained control. Operators can take manual control over the thread of execution by implementing `Operator.run()`, and pulling data from `ReadStreams`. *Callbacks are not invoked while run executes.*

4.3.1 Operator API

4.3.2 Examples

Full example at python/examples/simple_pipeline.py.

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(erdos.Operator):
    def __init__(self, read_stream):
        self.read_stream = read_stream

    @staticmethod
    def connect(read_streams):
        return []

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

4.4 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.

4.4.1 Timestamps

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.

4.4.2 Watermarks

Watermarks in ERDOS signal completion of computation. More concretely, sending a watermark with timestamp `t` on a stream asserts that all future messages sent on that stream will have timestamps `t' > t`. ERDOS also introduces a *top watermark*, which is a watermark with the maximum possible timestamp. Sending a top watermark closes the stream as there is no `t' > t_top`, so no more messages can be sent.

4.5 ERDOS Package Reference