MetroProcessor

Release 1.3.0

Feb 01, 2021

Contents:

The MetroProcessor device implements a runtime-programmable pipeline to transform and analyze data flowing through Karabo, e.g. to perform online monitoring or preprocessing.

CHAPTER 1

Introduction

The MetroProcessor expresses transformations via Python code injectable at runtime as a collection of functions, so called *views*. A view takes input from one or more data sources in Karabo, e.g. properties or fast pipeline data, and may return a result, which in turn serves as input to other views. The execution of views occurs per train and all data is matched on the train boundary for this purpose.

Typical applications are online analysis from slicing data to a region of interest over performing simple statistical analyses to extensive pipelines to process raw data in realtime. As the underlying analysis code can be changed quickly at runtime, it allows for fast iterations and explicit implementations to solve a given problem in the very same moment. In general, any operation that can be expressed with Python may be performed. This also includes calling native code or bindings to larger frameworks, e.g. OpenCL or CUDA. The pipeline can scale from running in the same process to balancing trains across cores or even nodes.

The device takes care of wiring the data flow to and from the pipeline code. Its output may be sent back to Karabo as pipeline data (WIP!) and is accessible via a bridge-like ZMQ protocol for consumption by external tools, e.g. for visualization or further analysis. Currently, the code is typically supplied via a file on the respective node the device is running on.

As an example, consider two spectrometers measuring the spectral distribution of a laser pulse before and after some optics. The laser operator would like to monitor both the respective spectra as well as their normalized difference. As each spectrometer has a unique wavelength calibration of its sensor, the difference requires interpolation. The corresponding analysis code could look like this:

```
import numpy as np
from scipy.interpolate import interp1d

@View.Vector
def src(x: 'SQS_ILH_LAS/SPEC/SRC.wavelengths',
        y: 'SQS_ILH_LAS/SPEC/SRC:output.data.spectrum'):
    """Normalized spectral distribution at the source.
    """

    return x, y/y.max())

@View.Vector
```

(continues on next page)

(continued from previous page)

```
def exp(x: 'SQS_ILH_LAS/SPEC/EXP.wavelengths',
        y: 'SQS_ILH_LAS/SPEC/EXP:output.data.spectrum'):
    """Normalized spectral distibution at the experiment.
    """

    return x, y/y.max()

@View.Vector
def diff(src_data: 'src', exp_data: 'exp'):
    """Difference spectrum on interpolated X vector.
    """

    # Unpack the view arguments.
    x_src, y_src = src_data
    x_exp, y_exp = exp_data

    # Define a common X axis for both spectrometers.
    x_both = np.linspace(max(x_src.min(), x_exp.min()),
                         min(x_src.max(), x_exp.max()),
                         min(len(x_src), len(x_exp)))

    # Obtain interpolation functions from scipy.
    f_src = interp1d(x_src, y_src)
    f_exp = interp1d(x_exp, y_exp)

    return x_both, f_src(x_both) - f_exp(x_both)
```

Here, three *views* are defined called `src`, `exp` and `diff`.

The first two views `src` and `exp` are used to combine the wavelength calibration and the spectral measurement, which is also normalized to unity, for each of the two spectrometers. Each view receives two arguments, the `wavelength` property and the key `data.spectrum` within the `output` pipeline hash. Argument annotations are used to declare the source for a view's argument, which are all coming from Karabo in this case. Both views will be executed individually for each train when their respective devices send out fast data with a spectral measurement, while the wavelength property as slow data is always available.

The third view `diff` is used to obtain the difference between both spectra. As its input, this view takes the result of the aforementioned views - hence, it will execute whenever both views are executed for a given train **and** return an actual result. As with Karabo sources before, another view may be declared in an argument annotation. Since the wavelength vectors of the two spectrometers may be different, it uses interpolation to obtain the spectra on a common wavelength axis and then performs the substraction.

As all views are annotated with a `Vector` decorator, a client will know to display it as a line plot. It is customary to interpret a tuple of vectors as the respective X and Y data.

CHAPTER 2

Data pipeline

The MetroProcessor device provides a programmable pipeline to perform operations on data flowing through Karabo. This pipeline consists of several stages. The frontend lives in the Karabo device and connects to all other Karabo devices required for data input. The data is matched by train and then send to the pool stage. This consists of one or more workers, each processing a single train. All their results are then send further to the reduce stage, which may perform operations across trains, e.g. an average. Finally, this stage also outputs the results over ZMQ to any connected clients:

All these components may run on a single machine or be spread across multiple nodes, both the stages as a whole as well as the workers constituting the pool stage. This allows to run complex calculations exceeding 100 ms per train in the pool stage, while still keeping the final pipeline output running at 10 Hz. Alternatively, it promotes the use of expressive code over highly optimized code, which is often harder to maintain in a scientific setting. In turn, it is important to keep the operations in the reduce stage as lean as possible, as each train is executed synchronously here.

The context configures the programmable portion of the pipeline. It is written in Python code, which expresses the transformation steps to be performed, declare parameters or even run a scriped measurement. Currently, this code must be contained in a file readable by the MetroProcessor device. It is structured into a series of functions, which perform any operations on the data and (may) each return a result. These functions are called *views* in the sense that they allow a particular "view" into the data.

Please refer to the documentation of the underlying metropc framework for details on how to write context files.

CHAPTER 3

Device operation

Before instantiation, only the number of workers in the pool stage (see *below*) and the ZeroMQ connections (see *below*) may need to be configured. The former will soon be replaced by an automatic scaling at runtime.

During runtime, most actions are performed via a set of slots:

- **Start, Stop** Start or stop the data pipeline. Before the pipeline may operate, a context must be initialized successfully via reconfigure and the device be in the ACTIVE state. Stopping the pipeline while an operator runs will cancel this operator as well.
- **Reconfigure** Initialize a new context from the provided source file. This slot is central to any changes to the pipeline configuration during runtime, e.g. for code changes to take effect or recover from an error condition. It is possible to perform this action in the PASSIVE, ACTIVE, PROCESSING (called hit reconfigure) and ERROR state. Any internal buffers are cleared as well (see *Clear buffer* below), while constant data and parameters are preserved if possible from the previous context.
- **Clear buffer** Clear transient data in the data pipeline, which is typically used for reduction operations such as averaging. When the context is reconfigured, this happens implicitly as all internal objects are recreated. This will not clear any feedback data in the pipeline (see *Clear const* below).
- **Clear const** Clear constant data in the data pipeline, which is typically used for reference or background signals. This kind of data will persist through a reconfiguration process and can only be cleared with this slot.
- **Suspend** Set an empty context, which will cause all device proxies and pipeline connections to be closed. In this PASSIVE state, the device will consume almost no system resources. It can be recovered by reconfiguration with a non-empty context.

The context is contained in a source file, which must be readable by the *xctrl* user on the node the device runs on. Currently, this excludes storing the file on GPFS in a proposal directory.

3.1 States

- **INIT** Occurs during the initial device initialization as well as whenever the pipeline context is reconfigured during normal operation.

- **PASSIVE** When the device is suspended, an empty context is loaded. Thus, there are no proxies or pipeline connections in this state and the device will consume almost no resources at all. This state is recommended when the device will not be used in the foreseeable feature in order to reduce the load on fast data producers. It is reached by pressing "Suspend" or reconfiguring with an empty context path.
- **ACTIVE** In this state, a valid context is initialized, all device proxies and pipeline connections are established and the pipeline may start processing at any time. It is typically entered when a context is reconfigured successfully from the INIT, PASSIVE or ERROR state.
- **PROCESSING** The pipeline is in full operation in this state and matched data is being sent for processing. The context may still be reconfigured, which will immediately return to this state if possible, i.e. no error condition occurs.
- **ENGAGING** The device enters this state while the pipeline is running and an operator is executed. Hence this can only be reached from the PROCESSING state and will return to it once the operator finishes or a recoverable error occurs. The context may not be reconfigured while an operator is running.
- **ERROR** This condition is reached whenever an unrecoverable error occurs. Typically, this is caused by a problem in context code. The device can only recover via a context reconfiguration, which will always transition into ACTIVE if successful, even if the error occured in PROCESSING or ENGAGING.
- **STOPPING** Occurs only during device shutdown.

3.2 Built-in scenes

There are two scenes provided with the device:

- The default `overview` scene provides all properties and slot for diagnostics purposes at runtime.
- The optional `compact` scene includes only the top-most portion of the `overview` scene with the most critical information for normal operations.

For operators, it is recommended to start from the `compact` scene and extend it by any parameters, action slots, operators or other GUI elements relevant for a given application.

3.3 Context-dependant properties

The pipeline context may inject additional properties (via parameters) or slots (via action views and operators) into the device schema, which are then available under their respective nodes `parameters`, `actionViews` and `operators`.

The device will attempt to preserve a parameter value across context reconfiguration, if it continues to be defined by it and did not change its type. In order to reset a value back to its default setting set in the code, the device may be suspended first.

3.4 Matching strategies

The data from all input sources must be matched for a train before it can be sent to the data pipeline, as in particular fast data arrives asynchronously at the device. The MetroProcessor provides different strategies on how to deal with this situation, which may differ in train latency and order through the pipeline. All these implementations use the `max_train_latency` property of the device for fine tuning. In all cases, input data older than this value is discarded immediately.

- **PATIENT** This is the most basic matching strategy, which buffers all input events and processes these entries in the pipeline when the corresponding train reaches maximum train latency. Already buffered data will be overwritten with newer data for the same train and input source, if it arrives before the latency cap is reached.
  - Latency always equal to maximum train latency.
  - Within train latency window, pipeline data is overwritten by newer data.
- **GREEDY** In contrast to the patient matching strategy above, this implementation distinguishes between *complete* and *incomplete* trains. A train is considered complete once all the input sources required by the context's views is present. In this case, the train is processed immediately and may therefore happen before an earlier, but incomplete train, enters the pipeline. A train is processed at the latest when its age reaches the maximum train latency.

  While the overall rate is the same as for the patient matcher, this implementation can offer significantly faster feedback if all input sources have similar latency. At the same time, it avoids the excessive stuttering of the cunning train matcher below when one or more inputs drop trains frequently (at the cost of higher latency for incomplete trains). It is therefore recommended for online visualization of results, in particular if fast feedback as a reaction to control action is desired.

  Duplicate entries are dropped for the same input source and train, as a train may have been executed the moment the original input arrived.
  - Minimum latency for complete trains, same as patient for incomplete trains.
  - Good balance between smooth output and fast feedback.
  - Duplicate data for same input and train is discarded.
- **CUNNING** The cunning train matching strategy extends the greedy algorithm by the assumption that input sources always arrive in-order. Once data is received for a particular train, the same input will never send data for an earlier train. This way, it is possible to execute incomplete trains much sooner when there is no hope of completing them anymore. Hence few train will reach the maximum train latency, unless it its value is set too low or too many trains are dropped by a device.

  This strategy will yield the lowest latency, but may experience stuttering when the input sources drop trains at different rates and/or drop different train IDs. In this case, trains will be released in bursts with possibly long lags in between. However, unlike the greedy matcher, trains are always executed in order. It is therefore best suited for transformation of data without any real-time visualization.

  As with the greedy train matching above, data will not be replaced if multiple entries arrive for the same input and train.
  - As many complete trains as early as possible.
  - Overall minimum latency, but may cause stuttering.
  - Duplicate data for same input and train is discarded.

In general, it is recommended to use `GREEDY` for real-time visualization and `CUNNING` for transformation application. The `PATIENT` strategy offers the smoothest output rates at the cost of higher, but constant latency.

3.5 Train offsets

A train offset may be applied to any fast data, if the source device appears to have an offset relative to the pipeline device. This allows to match data actually belonging to the same, but differing in their reported train IDs.

The input path must be an unambiguous qualifier for a `karabo#` data path. Only the source and pipeline name are used, any hash key is ignored.

3.6 Worker pool

The pool size should be large enough to ensure this stage is able to proces trains at 10 Hz or more, i.e. `>= ceil(t // 10)` for processing time `t` in the pool stage. This can be verified by ensuring the load reported in the worker statistics node is less than 1.0 for all pool workers.

3.7 ZeroMQ configuration

There are several ZeroMQ connections between the device, the pipeline stages and any clients:

- **Output** The reduce stage binds a `PUB` socket to this address and sends the pipeline results. It is recommended to set this to an Infiniband address for maximum bandwidth to clients.
- **Control** The device binds a `ROUTER` socket to this address to control the pipeline stages. By default, this is set to an IPC connection on the same node using the device ID as path.
- **Reduce** The reduce stage binds a `PULL` socket to this address to receive the pool stage results directly via `PUSH`. By default, this is set to an IPC connection on the same node using the device ID as path.

In most cases, only the output address should require any adjustments, e.g. in order to run several device instances on the same node.

CHAPTER 4

Clients

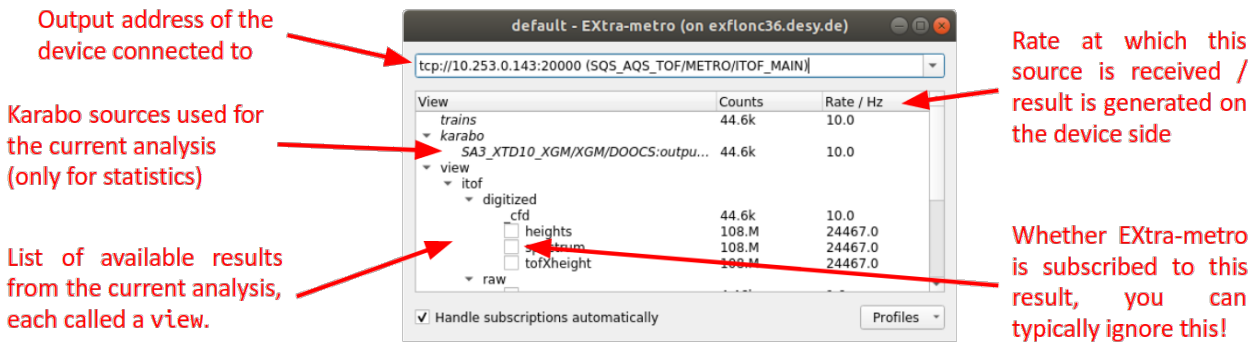
The output protocol is currently under development and not yet finalized. There will be a full-fledged client API to receive the pipeline results as well as integration into Karabo. Until then, there is only a single client solution called *EXtra-metro* provided on the European XFEL computing infrastructure.

4.1 EXtra-metro

The software is available through the module system. You may start this software on any node on the online cluster. Please find the nodes belonging to each SASE branch and/or instrument and information on their intended usage here.

```
% module load exfel EXtra-metro
% extra-metro
```

Once started, the main window shown below will appear. It allows to connect to the output address of any running MetroProcessor device, displays statistics to its pipeline data flow and visualize the results. The address for a number of deployed instances are provided by default, but any custom location may be entered. Please contact the data analysis team (da-support@xfel.eu) if you would like to add a permanent device instance to this list.



Once connected successfully, the next index received from the pipeline should fill the tree view below, with at least *trains* always appearing. Any further entries depend on the loaded context. When the pipeline begins processing, the counts and rate should update correspondingly.

Only the result of views is actually forwarded through the pipeline, the entries for other path types such as `karabo#` are only for statistics. If a raw signal from Karabo is desired, a corresponding view must be added to the pipeline context which simply returns said value. Views are organized into the tree structure further by slashes `/` in their name.

4.1.1 Visualizing results

If an entry provides a result, a checkbox is present next to its name and it may be visualized in several ways:

- Double-clicking a view will try to choose a suitable kind of display and configuration automatically. In most cases, this is the recommended method.
- Right-clicking a view and selecting *Display* will choose the same kind of display as above, but allows to change its initial configuration. For most arguments, a tooltip is provided.
- Right-clicking a view and selecting *Display by* allows to change both the kind of display as well as customize its configuration. Please note that their use may not be fully documented yet. The available modules, which may be of general use, are:
  - *fast_plot*: Plots 1D vectors or 2D arrays as list of vectors of up to 1M samples each.
  - *hist1d*: Bins a stream of scalars to a 1D histogram.
  - *hist2d*: Bins a stream of points (X,Y coordinates) into an image with extensive support for scaling/translating the projection via mouse.
  - *image*: Plots 2D arrays as images.
  - *polar_plot*: Plots a 1D vector in polar coordinates.
  - *sorted*: Displays a bar plot of a 2D array sorted by its second column and labeled by its first.
  - *value*: Prints any value as text.
  - *waveform*: Plots a scalar value over time.

4.1.2 Profiles

The current geometry of all open windows and their configuration state may be saved to a profile in order to restore it at a later time. A new profile is created by selecting *Save current configuration* in the *Profiles* menu. If a profile has already been loaded or saved recently, you may also simply choose *Overwrite last profile*. The top part of the entries in this menu will allow to load any of the available profiles, which may be refreshed manually by selecting *Rescan profiles*

The profile is saved as a JSON file either in your own home directory, which is shared across all nodes belonging to a particular SASE branch, or the *usr/* directory of a proposal. You may switch between these locations by selecting *Local profiles* or *Profiles in proposal*, respectively. The list of available profiles will change accordingly and this setting is preserved across closing an instance. Finally, you may load a file from any location using *Choose external file*.

A profile may be loaded immediately when opening a new instance of *EXtra-metro* by specifying it on command-line:

```
% extra-metro --profile <name>
```

If the profile location differs from your currently selected one or you want to select a location just for this execution, you may also specify a proposal:

```
% extra-metro --proposal <number>
```

Please note that only proposals available on the chosen online cluster node are available and you must have the corresponding access rights.

4.1.3 Result topics

The underlying protocol to the data pipeline uses ZeroMQs `PUB`/`SUB` pattern to only send those results to a client in which he is interested in. Each view result is represented as a topic to which a client may subscribe in order to receive the data.

If the checkbox *Handle subscriptions automatically* is selected, this will be done transparently by *EXtra-metro* whenever data is required for visualization or ceases to be required. You may change this subscription for each view result by the corresponding checkbox in its tree entry. However, this will stop any open visualization for this particular view from updating! You may override this automatic handling completely or just for particular views at any time, e.g. to preserve a particular result.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search