Handling Fast Data through Pipelines

Karabo Developer Workshop 2024

Raul Costa
Controls Group

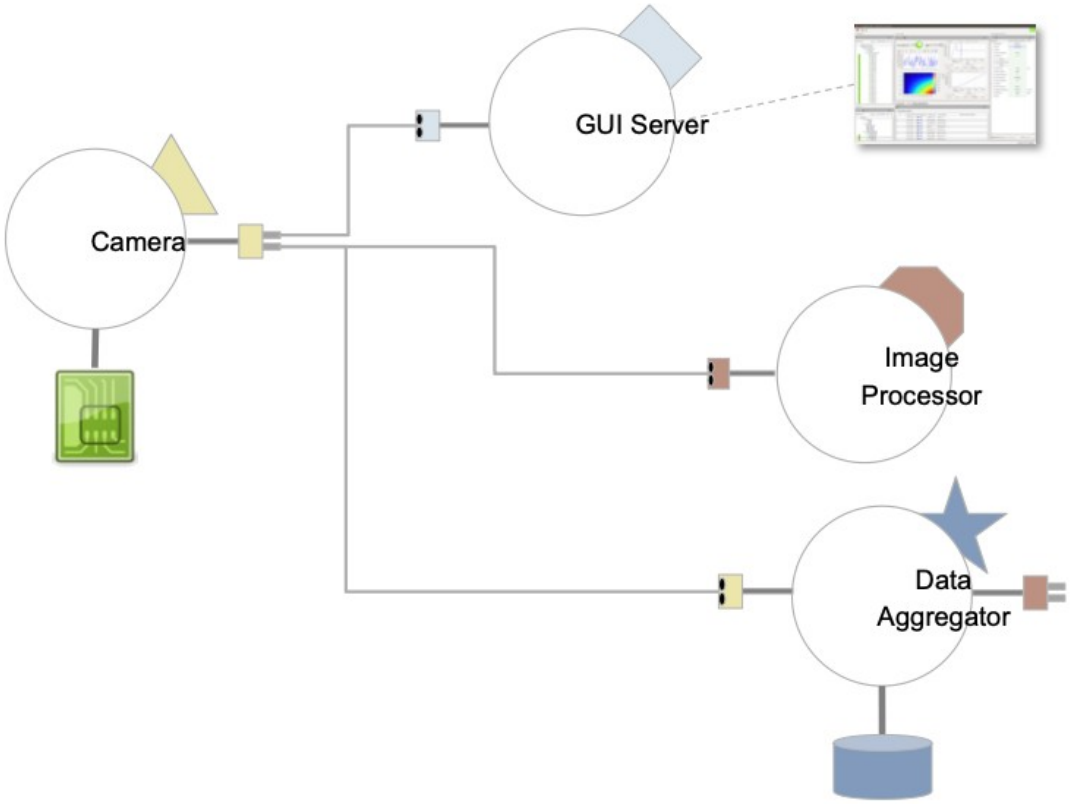
Schenefeld, March 1st, 2024.

Outline

- Introduction: Pipelines and their uses (~ 5 minutes)
- Hands-On Exercises (remaining time)

Introduction – Pipelines and their Uses

- Karabo data can be split in 2 categories: **Slow Data** and **Fast Data.**
- **Slow Data** refers to message exchanged with broker (AMQP) intermediation. Examples: devices configurations and events (instantiations, shutdowns, state changes). By default persisted by the Data Loggers (Influx)
- **Fast Data** refers to data that is sent directly from one device to another without the “overhead” of broker intermediation. Used when high data throughputs* must be supported. Examples: camera images, data collected by detectors. Saved by the DAQ when configured on a per proposal basis.

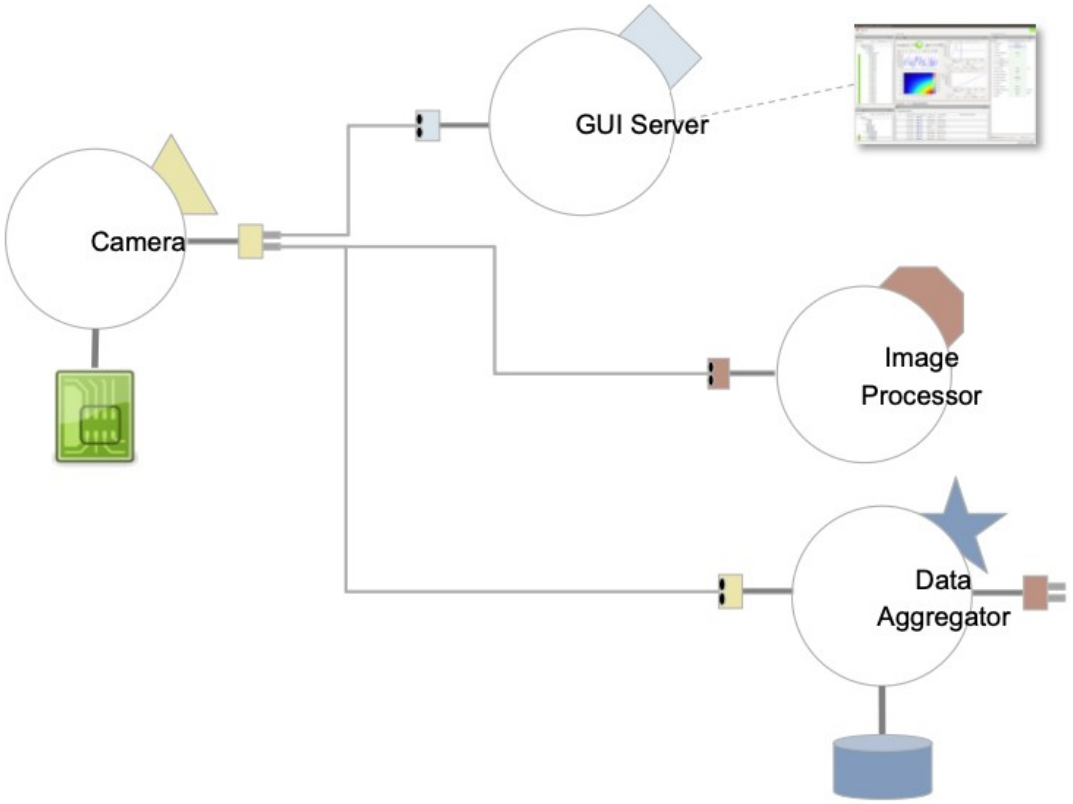


A typical pipeline usage scenario

* small data items at high frequencies can also be considered Fast Data.

Introduction – Pipelines and their Uses (continued)

- **Pipelines** are direct TCP connections between two Karabo devices through which Fast Data flows.
- A Pipeline has two ends: the one belonging to the Data-producing device (the Camera in the picture) is called **Output Channel**; the one belonging to the Data-consuming device is called **Input Channel**.
- A Karabo device can have zero, one or more Output Channel(s) and/or Input Channel(s).
- The Input Channel is the “most active” end of the pipeline – it is the end that initially establishes the connection with a given Output Channel and also controls the data flow by informing the Output Channel when it is ready to consume data (an Output Channel only sends to Input Channels it knows are ready to consume the data).



A typical pipeline usage scenario

For more information on Karabo Pipelines:

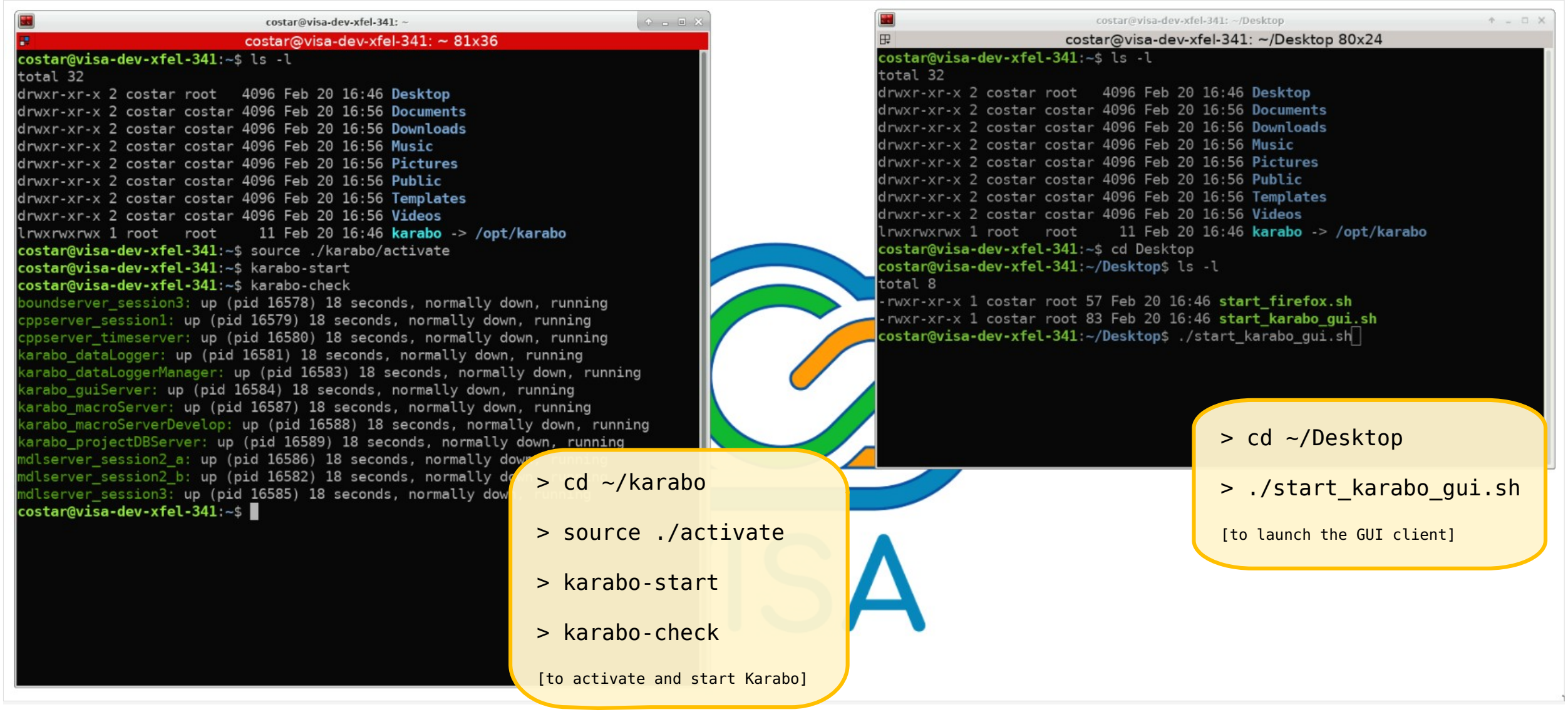
- Flucke, Gero – **C++ Pipelines for Karabo Device Developers**, 21.10.2022 - https://syncandshare.xfel.eu/index.php/s/NgMzA5yirNsFXnW?dir=undefined&path=%2FTrainings&openfile=9585678
- Flucke, Gero – **Karabo Pipeline Configurations**, 14.04.2023 - https://syncandshare.xfel.eu/index.php/s/NgMzA5yirNsFXnW?dir=undefined&path=%2FTrainings&openfile=16279716
- **Karabo Online Documentation – Karabo Concepts > Communications** - https://rtd.xfel.eu/docs/karabo/en/latest/concepts/communication.html

Now to the **Hands-On** part!

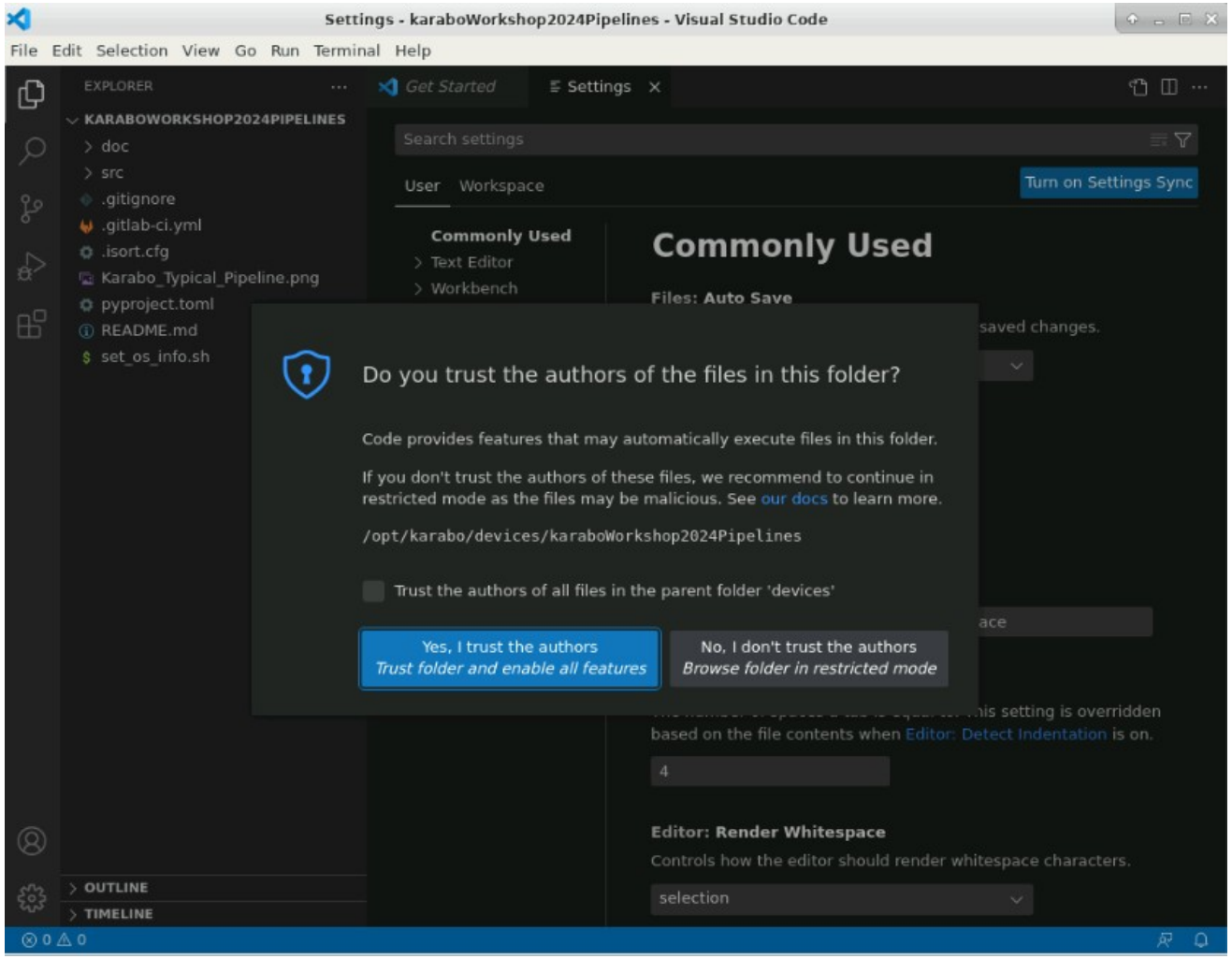
Hands-On Exercises - Outline

- Connect a skeleton MDL device to a simulated camera
- Code Part I: Process data sent by the camera
  - Step 1: Show the number of frames sent by the camera
  - Step 2: Show the min, max and average of the pixels of each frame
  - Step 3: Add PROCESSING and ERROR states, handle end-of-stream (optional)
- Code Part II: Forward processed data via an output channel
  - Step 4: Forward min, max and average of the pixels of each frame
  - Step 5: Capture and forward frame timestamp (optional)
  - Step 6: Forward camera acquisition cycles (end-of-stream events) (optional)

Start Karabo's Working Environment on the VISA VM



Open the Karabo Device on Visual Studio Code



From the directory **`~/karabo/devices/karaboWorkshop2024pipelines`**, *run* [> **`code .`**].

Launch and Explore the Simulated Camera

- In the VISA VM, activate the Karabo GUI Client and open the project **`Session_3`**.
- Find the simulated camera device – its name is **`SIM_BL_SYS/CAM/CAM`** - and instantiate it.
- Double-click on the simulate camera device node in the project tree – the scene for the camera will be displayed. This scene will be used multiple times throughout the exercises.
- In the Configuration Editor, expand the **`Output`** property of the simulated camera. This is the camera's output channel. Click on the **`Table Element`** button in front of the **`Output > Connections`** properties – those are the input channels currently connected to the camera's output channel. One connection should be present – the connection used by the device scene to show the camera image.
- Still in the Configuration Editor, expand the **`Output > schema`** property. This shows how the data the camera sends through its output channel is structured. Take a look at the **`Data > Image > Pixel Data`** path of the schema.

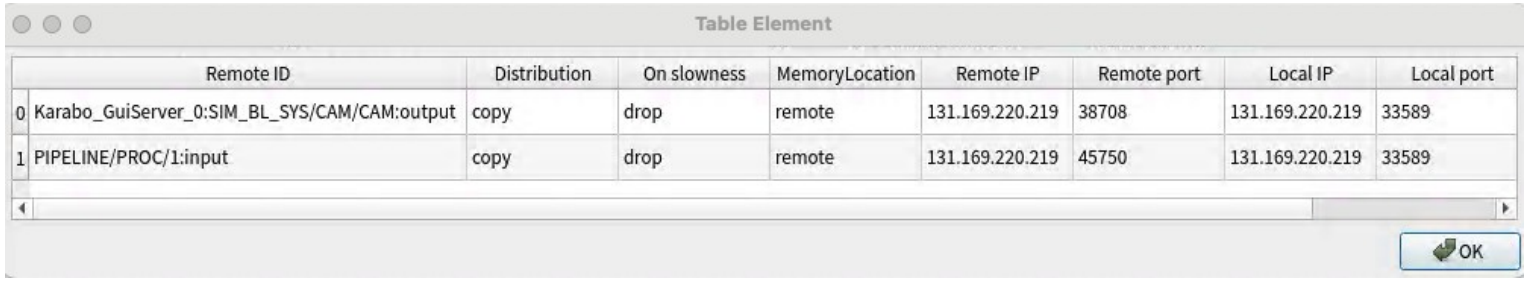
Instantiate the Skeleton MDL Device and connect it to the camera

- Still in the project **`Session_3`** opened in the previous step, find the device we will be working on, **`PIPELINE/PROC/1,`** and select it (no instantiation yet).

- In the Configuration Editor, check that the **`Input > Configured Connections`** property has the value **`SIM_BL_SYS/CAM/CAM:output`**. This the ID of the simulated camera output channel, formed by the concatenation of the DeviceID of the channel hosting device, a **`':'`**, and the ID of the output channel.

- Check the **`Output > Connections`** property of the simulated camera: the connection to **`PIPELINE/PROC/1:input`** should be there. If the simulated camera scene is open, it should be also listed there (image below).

- Pressing the **`Acquire`** and **`Stop`** buttons in the camera scene doesn't do anything on our device ... not for long!



Coding Part I

Process Data Sent by the Camera

Step 1 – Show the number of frames sent by the camera

- Issue a [> **`git checkout hands_on_1_initial]`** command from a terminal session with **`/opt/karabo/devices/karaboWorkshop2024Pipelines`** as the current directory.

- The initial version of our camera image processing device already comes with a input channel defined: the **`@InputChannel`** decorator for the **`async def input`** coroutine defines an input channel property for the device and establishes the coroutine as the handler for data received from an output channel.

- When the input channel is connected to a camera, each frame sent by the camera will activate the **`input`** coroutine once, passing the frame data via the **`data`** parameter. The structure of the parameter matches the schema of the camera's output channel. The second parameter, **`meta`**, is unused for now; I'll be used in Step 4.

- **The task of this step is** to add a **`framesAcquired`** property of type **`UInt32`** to our device - the number of frames received from the camera since the device instantiation. To test your progress, shutdown the device server of our device on the GUI client. As soon as the device server is back, instantiate the processing device - this syncs the running device with its latest version saved in Visual Studio Code.

- **`[> git diff hands_on_1_initial hands_on_1_done]`** will display the solution for this step.

Step 2 – Show the min, max, and average of the pixels values of each frame

- Issue a [> **`git checkout hands_on_1_done]`** command from a terminal session with **`/opt/karabo/devices/karaboWorkshop2024Pipelines`** as the current directory.
- The method **`async def process_image(self, pixels)`** currently does nothing. It's called from the **`async def input`** coroutine, which sends it the pixels of the current frame sent by the camera as the value for the **`pixels`** parameter.
- **The task of this step is** to add the properties **`pixelMean`** (of type **`Double`**), **`pixelMin`** (of type **`UInt16`**), and **`pixelMax`** (of type **`UInt16`**) to our device. Those properties values should be the average, minimum, and maximum values of the pixels of the most recent frame sent by the camera. **Hint:** the **`pixels`** argument passed to **`process_image`** is an object of type **`ndarray`** and has the methods **`min()`**, **`max()`**, and **`mean()`**.
- **`[> git diff hands_on_1_done hands_on_2_done]`** will display the solution for this step.

Step 3 (optional) – Add PROCESSING and ERROR states, handle end-of-stream

- Issue a [> **`git checkout hands_on_2_done]`** command from a terminal session with **`/opt/karabo/devices/karaboWorkshop2024Pipelines`** as the current directory.

- An end-of-stream event is sent by a camera when it stops acquiring images. To handle end-of-stream events, an input channel has to declare an **`async input(self, output_channel_id)`** method decorated with **@input.endOfStream**.

- **The task of this step are:**
  - Add a **`PROCESSING`** state to the device to indicate that data is being received from the camera.
  - Add an **`ERROR`** state to the device to indicate any error while processing data sent by the camera. Error details should be shown in the device's **`status`** property. Successful processing data while in **`ERROR`** state should take the device back to **`PROCESSING`** state.
  - Handle end-of-stream events from the camera by putting the device back in **`ON`** state and indicating that no processing is taking place by showing **`IDLE`** in the device's **`status`** property. Reset the **`framesAcquired`** value when the camera starts a new acquisition cycle.

- **`[> git diff hands_on_2_done hands_on_3_done]`** will display the solution for this step.

Coding Part II

Forward Processed Data Via an Output Channel

Step 4 – Forward min, max and average of the pixels of each frame

- Issue a [> **`git checkout hands_on_4_initial]`** command from a terminal session with **`/opt/karabo/devices/karaboWorkshop2024Pipelines`** as the current directory.

- We start with the device already with an output channel: its data structure is defined by **`class DataNode`** (line 22), which becomes the field **`data`** of **`class ChannelNode`** (line 39). **`Channel Node`** is then specified as the schema of the **`output`** OutputChannel (line 91).

- **The task of this step is** to forward the values computed for the **`pixelMean`**, **`pixelMin`**, **`pixelMax`** properties of the device through its output channel. **`Hint:`** reinstantiate the device after shutting down its device server. Take a look at the **`Output > schema > data`** property of the device in the Configuration Editor to see how the data must be structured. Await for the **`self.output.writeData()`** coroutine to send the data.

- **`[> git diff hands_on_4_initial hands_on_4_done]`** will display the solution for this step. The forwarded content can be seen in the scene **`PIPELINE_PROC_1_OUTPUT`** in the same project that has our device and the simulated camera

Step 5 (optional) – Capture and forward frame timestamp

- Issue a [> **`git checkout hands_on_4_done]`** command from a terminal session with **`/opt/karabo/devices/karaboWorkshop2024Pipelines`** as the current directory.
- The metadata (data about data) for the data received by an input channel is available as the second parameter of the **`async def input handler`** coroutine – **`meta`** parameter in line 64.
- The timestamp of the data received can be accessed within the input handler method as **`meta.timestamp.timestamp`**.
- **The task of this step is** to forward the timestamp of the data received by the input channel of our device to its output channel. **Hint:** the **`self.output.writeData`** call, currently using no argument, supports a keyword parameter called **`timestamp`** which allows specifying a timestamp for the data being written to the output channel.
- **`[> git diff hands_on_4_done hands_on_5_done]`** will display the solution for this step.

Step 6 (optional) – Forward camera acquisition cycles (end-of-stream events)

- Issue a [> **`git checkout hands_on_5_done]`** command from a terminal session with **`/opt/karabo/devices/karaboWorkshop2024Pipelines`** as the current directory.
- **The task of this step is** to forward any end-of-stream event received from the camera to the output channel of the device. Hint: the output channel has a coroutine that sends an end-of-stream through the channel. For our device it can be invoked with **`self.output.writeEndOfStream()`**.
- **`[> git diff hands_on_5_done hands_on_6_done]`** will display the solution for this step.