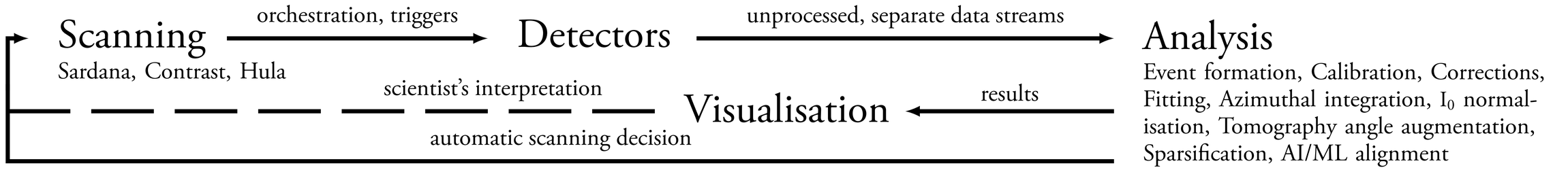
DRANSPO.SE – A Flexible Live Processing Pipeline

Felix Engelmann

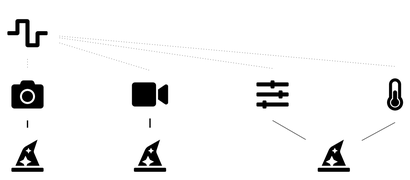
Scientific Data

Accelerating Scientific Insight Through Rapid Feedback



Source Trigger Map

Ingesters connect to different data sources, such as detectors, electrometers and encoders and form events according to the Trigger Map.



- STINS (2d detectors, multipart zmq PULL)
- PandABox PCAP (encoders, TCP/ascii)
- Contrast/Sardana (snapshots)
- Xspress3 (energy spectra, zmq SUB)

Which *frames* from which *detectors* belong to the same *event* and have to be processed by the same *worker* having which *tags*?

📷	all	1	{3,debug}	5	7	8	10	10	11
📹	all	2	{4,debug}	6	7	9	10	10	11
🎚	all	all	debug	none	∅	none	none	none	none
🌡	all	{1,2}	none	{5,6}	none	{8,9}	10	none	11

This supports detectors not producing a frame for a trigger (*none*) and discarding frames (∅). Meta information is easily distributed to all workers by *all*. Tags allow different sets of workers, the **debug worker** just exposes the last n events.
The trigger map is provided by the scanning software:

contrast: patch after `_generate_positions`

sardana: global hook to extract number of points

Map Reduce

Events are dispatched to the **next free** available worker satisfying the constraints and tags. The fine grained **load balancing** allows to keep state in the worker. This enables **temporal analysis**, e.g. calculate the difference of two consecutive exposures. All required parameters need to be described.

```python
class FluorescenceWorker:
    @staticmethod
    def describe_parameters():
        return [IntParameter(name="roi1-start")]

    def __init__(self, parameters=None):
        self.number = 0

    def process_event(self, event: EventData, parameters=None):
        print(event)
        # parse zmq frames, fit spectra to get concentrations, extract motor position
        return {"position": mot, "concentrations": ...}
```

Workers emit arbitrary objects as reduced results. It is important that the output of all workers combined does not exceed a bottleneck of around 10 Gbit/s.
The results of workers are forwarded to a **single reducer** which has access to the full history of the scan. It has limited capacity and needs to operate at line speed which is fine for simple operations such as appending worker results to a list.

```python
class FluorescenceReducer:
    @staticmethod
    def describe_parameters():
        return [FileParameter(name="dest_file")]

    def __init__(self, parameters=None):
        self.publish = {"map": {}}

    def process_result(self, result: ResultData, parameters=None):
        if result.payload:
            self.publish["map"][result.payload["position"]] = result.payload["concentations"]
```

The reducer has a special attribute `publish` which is exposed through a REST api with JsonPATH selections and numpy slicing.

Visualisation

The most flexible way to view results is to access the reducer data through a Jupyter notebook.

```python
req = requests.get("http://pipeline-reducer/api/v1/result/$")
result = pickle.loads(req.content)
pos = list(result[0]["map"].keys())
plt.scatter([x[0] for x in pos],[x[1] for x in pos])
```

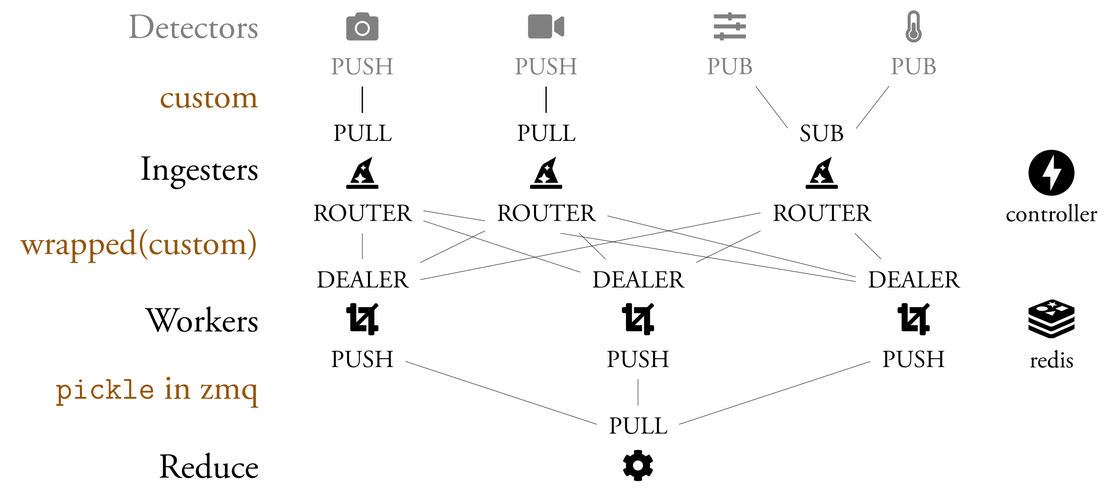
Alternatively a custom GUI may be developed which integrates setting **parameters** or selectively zooming into regions, if the whole data set is too large for the local memory.

Control System Integration

A Tango device server presents the status of a pipeline and allows to set parameters. It shows which ingester streams are available and the load of workers, allowing to adjust the worker **scaling**. Pipeline parameters support `str`, `int`, `float` and `file`.

State	CLOSE
Status	contrast-ingester, streams: ['contrast'] xspress3-ingester, streams: ['xspress3'] x3mini-ingester, streams: ['x3mini'] Worker-VKbSqFyDIU, tags: ['generic'], last:133, evs:19, ld10: 0.06, ld: 0.09 Worker-mPfJRkLXjj, tags: ['generic'], last:133, evs:18, ld10: 0.06, ld: 0.09
completedEvents	134
totalEvents	134
mca_config	/data/xrf/fit_config_removed.cfg

Architecture & Performance



The throughput of the system is only limited by the underlying kernel and zmq library. With jumbo frames, we successfully processed a single stream of **23 GBit/s** from a 16 bit `cmos` camera at 120 Hz. Affinity of ingesters heavily impacts performance.
The design allows arbitrary **horizontal scaling** for many ingesters. If a detector supports one stream per module, multiple ingesters are used to reassemble full frames for a worker. We tested ingesting events at up to **2 kHz**.

Deployment

If a kubernetes cluster is available, deployment is handled by a `helm` chart with values

```yaml
ingesters:
  orca:
    connect_url: "tcp://danmax-orca-daq-zmq-egress.danmax-orca:5556"
    ingester_class: "ZmqPullSingleIngester"
    affinity:
      namespace: danmax-orca
      component: daq
  pcap:
    connect_url: "tcp://172.16.214.46:8889" # panda main
    ingester_class: "TcpPcapIngester"
    stream: "pcap_rot"
workers: 2
science_image: "harbor.maxiv.lu.se/daq/dranspose/danmax-fluorescence:main"
worker: {class: "src.worker:FluorescenceWorker"}
reducer: {class: "src.reducer:FluorescenceReducer"}
```

Without kubernetes, the only dependency is a `redis` to which all **containerised** components connect.

Development `pip install dranspose`

The package provides a `dranspose` cli to run components or develop scientific code.

- Ingesters dump stream messages to storage.
- Replay streams to custom workers and reducers.

To develop a new ingester, capture the raw packets and perform test-driven development. To get insight into live packets, a **debug worker** exposes sampled full events over a REST interface.
Documentation at `https://dranspo.se/`