

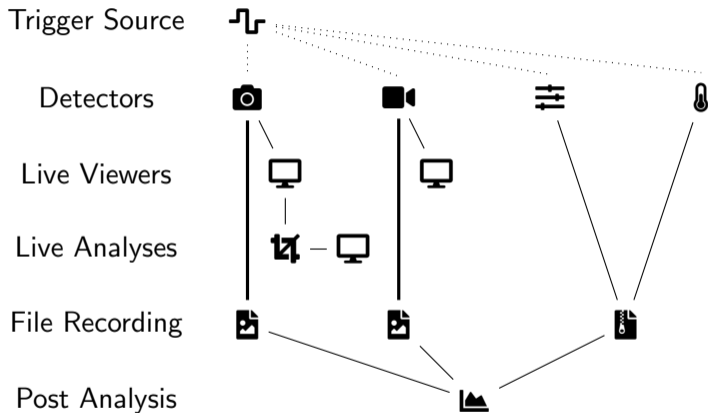
dranspose: distributed event formation with dynamic map-reduce

Felix Engelmann

`felix.engelmann@maxiv.lu.se`

30th November 2023

Existing Infrastructure



Limitations of Live Analysis

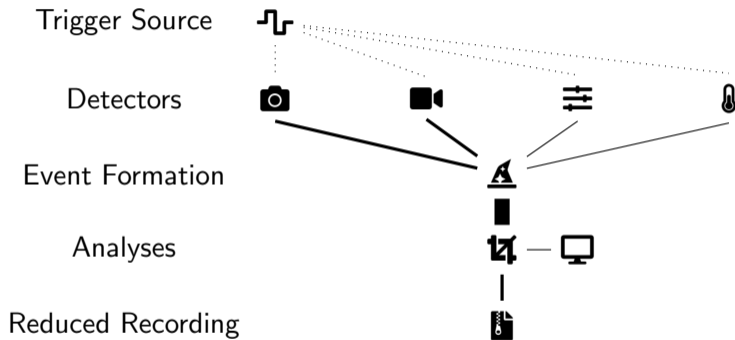
Single Data Stream

- ▶ only one detector data available
- ▶ simple tools, e.g. azint, crop, time integration
- ▶ beyond limits: normalisation to I_0 , sorting by motor position

Custom Modules

- ▶ module development by SciDa
- ▶ custom deployment/integration
- ▶ custom live viewer interaction (mostly REST)

Event Formation


















Frame/Worker Matrix Transformation

Frame Stream

	1	2	3	4	5
	1	2	3	4	5
	1			2	
	1		2		3

Event Stream

Event 1	 1	 1	 1	 1
Event 2	 2	 2		
Event 3	 3	 3		 2
Event 4	 4	 4	 2	
Event 5	 5	 5		 3

Stream

- ▶ matrix is sequentially filled column by column
- ▶ possibly unknown size (reactive scanning)

Bandwidth and Latency

Limitations

- ▶ TCP connection max 60 Gbit/s
- ▶ ZMQ connection measured ca. 30 Gbit/s

Bottleneck

Event Formation



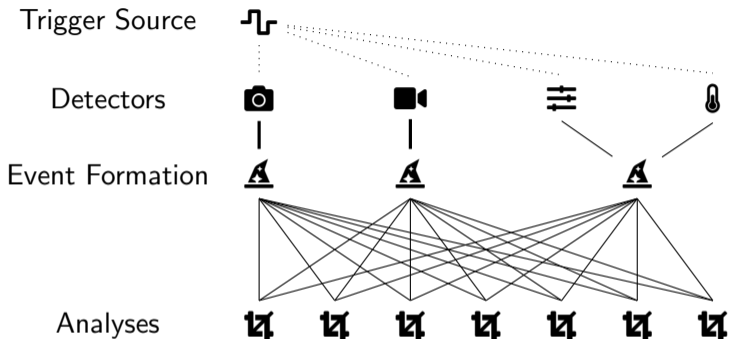
Analyses



Processing Delay

- ▶ acquisition at $\approx 100 - 1000$ Hz
- ▶ processing at $\approx 5-10$ Hz

Parallelisation



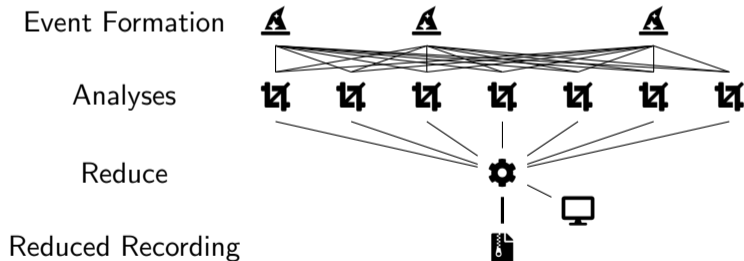
Inter Event Analysis

- ▶ time integration
- ▶ temporal correlations

Sequential Reduce

Operations at Acquisition Speed

- ▶ append to list
- ▶ sum


















Intense Inter Event Computation

Examples

- ▶ aligning images (correlation)
- ▶ temporal fourier transform

Stateful Workers

- ▶ load balance with constraints
- ▶ e.g. worker selected for event n will also get $n + 1$

Worker 1	 1	 1	 1	 1
Worker 1	 2	 2		
Worker 2	 3	 3		 2
Worker 2	 4	 4	 2	
Worker 1	 5	 5		 3





Trigger Map

Event Definitions

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

Virtual Workers

- ▶ virtual workers are dynamically assigned to real workers
- ▶ special *all* workers (stream headers), or discard frame with \emptyset
- ▶ *none* if stream has no frame for event

	all	1	3	5	7	8
	all	2	4	6	7	9
	all	all	none	none	\emptyset	none
	all	{1,2}	none	{5,6}	none	{8,9}

Scanning

trigger map specified by scanning software, append-only extendable

Development: Events

ease of use/development by SciDa and beamline staff

Event Structure

```
StreamName = NewType("StreamName", str)
EventNumber = NewType("EventNumber", int)
```

```
class StreamData(BaseModel):
    typ: str
    frames: list[zmq.Frame]

class EventData(BaseModel):
    event_number: EventNumber
    streams: dict[StreamName, StreamData]
```

Development: Worker

```
class FluorescenceWorker:
    def __init__(self):
        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": ...}
```

reinstantiated for every scan (new Trigger Map)

Development: Reducer

```
class FluorescenceReducer:
    def __init__(self):
        self.publish = {"map": {}}

    def process_result(self,
                      result: ResultData,
                      parameters=None):
        print(result.event_number)
        print(result.worker)
        print(result.parameters_uuid)
        data = result.payload
        self.publish["map"][data["position"]] = \
            data["concentrations"]
```

reinstantiated for every scan (new Trigger Map)

Development: Viewer

Jupyter Notebook

```
import requests, pickle
import matplotlib.pyplot as plt

params = {}
requests.post("http://<ns>-ctrl.../params", json=params)
# start scan
r = requests.get("http://<ns>-reducer.../result/pickle")
data = pickle.loads(r.content)
# data = FluorescenceReducer.publish

plt.imshow(data["map"])
```

Development: Silx based Viewer

Parameters

- ▶ set parameters
- ▶ influence slow processing

Partial Views

- ▶ keep large accumulated data set on reducer
- ▶ query specific slice of `reducer.publish`
- ▶ e.g. `http://<ns>-reducer.../result/map/100:110, :`

Development: Testing

Recording

ingesters optionally write all zmq frames to disk (sequential pickle dumps)

Replay

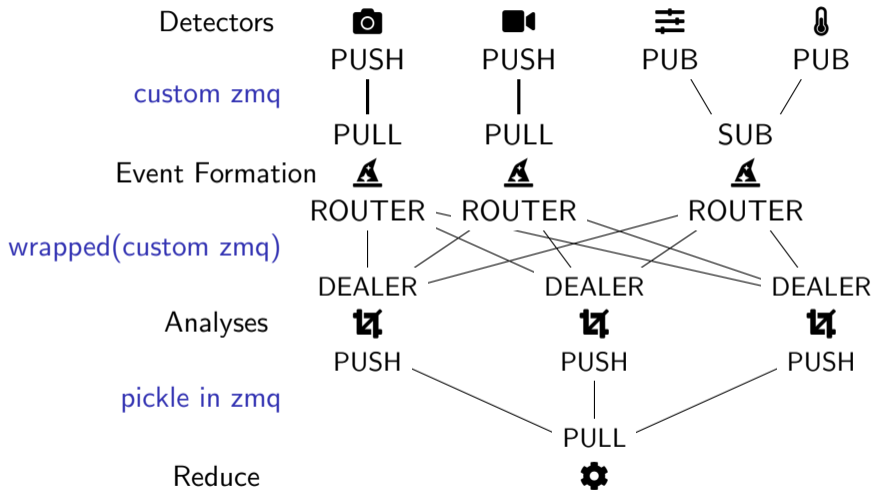
- ▶ from recorded zmq frames
- ▶ from hdf5 files

Local

run file-based ingesters, workers and reducer locally

Internals

Architecture, ZMQ



Architecture, redis

config

- ▶ components publish config (connected peers, trigger map version, zmq url)
- ▶ timeout for liveness probe

▶▶ updates

- ▶ controller notifies of new mapping/parameters

▶▶ ready

- ▶ workers notify readiness after event processed

▶▶ assign

event_number: EventNumber

assignments: dict[StreamName, list[WorkerName]]

Event Coordination


Controller

- ▶ wait for new entry in  ▶▶ ready
- ▶ assign worker to first unassigned virtual worker (and *all*)
- ▶ distribute WorkAssignment in  ▶▶ assign

Ingester

- ▶ filter assignment for own streams
- ▶ combine all local streams
- ▶ copy event to specified workers (ROUTER)

Worker

- ▶ filter assignment for own work
- ▶ listen to ingesters with participating streams
- ▶ assemble EventData
- ▶ call custom code
- ▶ send pickled result to reducer
- ▶ send ready message to  ▶▶ ready

Common Modules

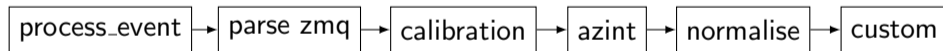
ZMQ Format / STINS

- ▶ unpacking of (mult-part) zmq frames to numpy arrays

Calibration

- ▶ installed as python modules

Middlewares



- ▶ maybe register required parameters?
- ▶ registered in Worker `__init__`

Tests

End-to-End

- ▶ stream fake zmq frames
- ▶ full scan test

Protocols

- ▶ Pydantic BaseModel (similar to dataclass)
- ▶ all messages defined and validated
- ▶ no dicts with random fields

Typing

- ▶ type hint annotations
- ▶ mypy strict

Deployment

Docker

- ▶ install custom dependencies
- ▶ end-to-end build latency multiple minutes

K8s

- ▶ HELM chart for beamline
- ▶ restart pulls new version
- ▶ different containers for different experiments

Versioning

- ▶ add git commit hash to reduced data
- ▶ add parameters to h5 file

Performance

Bandwidth

- ▶ 10 Gbit/s from b-daq-cn2 and b-daq-cn3
- ▶ 8 workers

horizontally scalable if each stream ≤ 30 Gbit/s

Latency

- ▶ ≈ 2 kHz with enough workers

practically limited to $\approx n$ workers \Rightarrow max worker runtime $\frac{n}{\text{acquisition rate [Hz]}}$ s

Virtual Worker Distribution

$\forall st_0 \in \mathbb{N}, h \in [|\text{workers}|, \infty)$:

$$|\{M_{\text{ev},st} : st_0 \leq \text{ev} < (st_0 + h), st \in \text{streams}\}| > h - \epsilon$$

Outlook

File Writing

- ▶ custom by developers?

Autoscaling

- ▶ observability of workers (queues)
- ▶ duty cycle of workers
- ▶ non-deterministic worker functions
- ▶ integration with k8s

Scan Integration

- ▶ publish trigger map
- ▶ append to trigger map