O2 DATA PROCESSING LAYER

Giulio Eulisse (CERN)

DISCLAIMER

Design and implementation subject to change:

What I am about to describe is not yet approved by the O2 TB and it's therefore still internal and up to discussion within WP4 (Framework).

Opt-in

Baseline will always be "if you write a device which respects O2 Data model, you will be able to read from / write to the Data Processing Layer".

Still we think there is some value in it

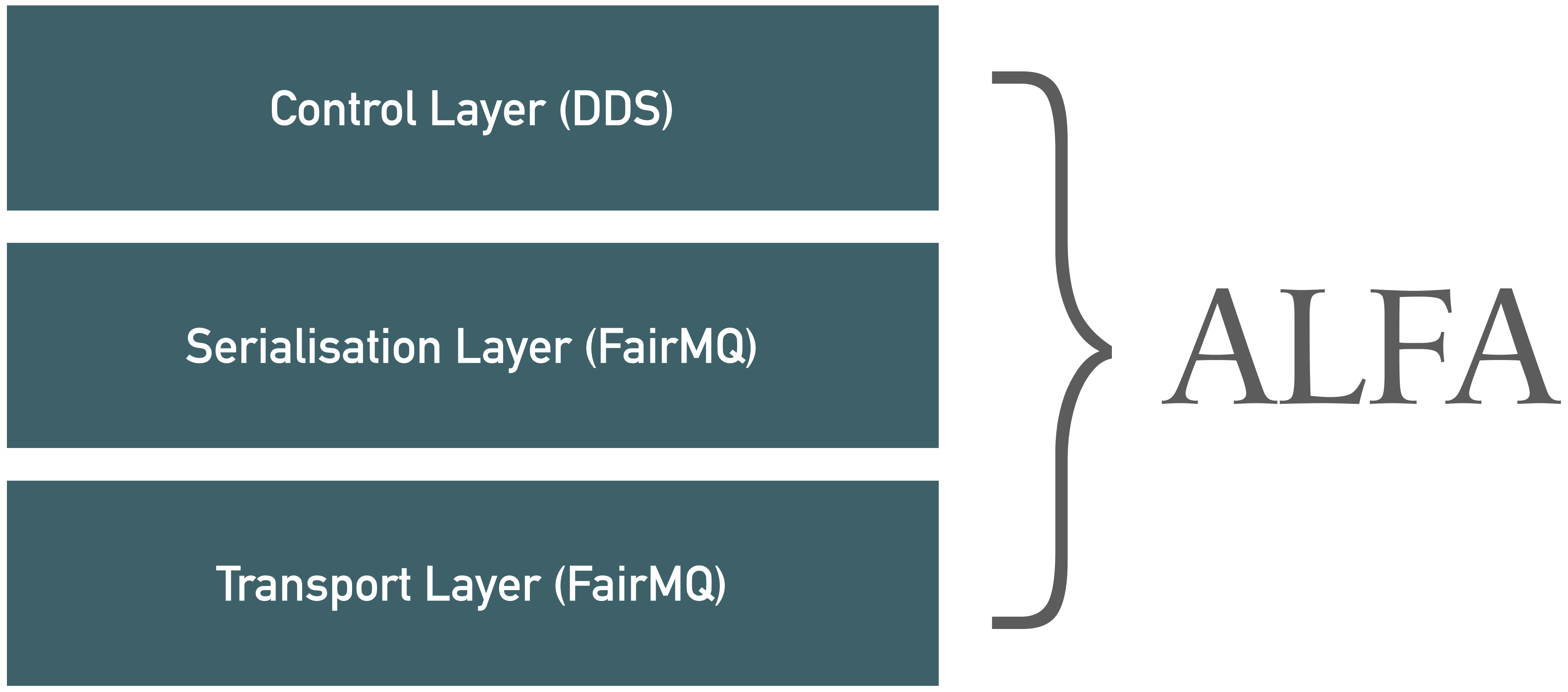
It nevertheless tries to give an idea on what are our (WP4) ideas on how Data Processing should work.

ALFA

Powerful Actor Model framework

Computation wrapped in entities ("Devices") which communicate via message queue

Building block for a distributed, fault tolerant, asynchronous system



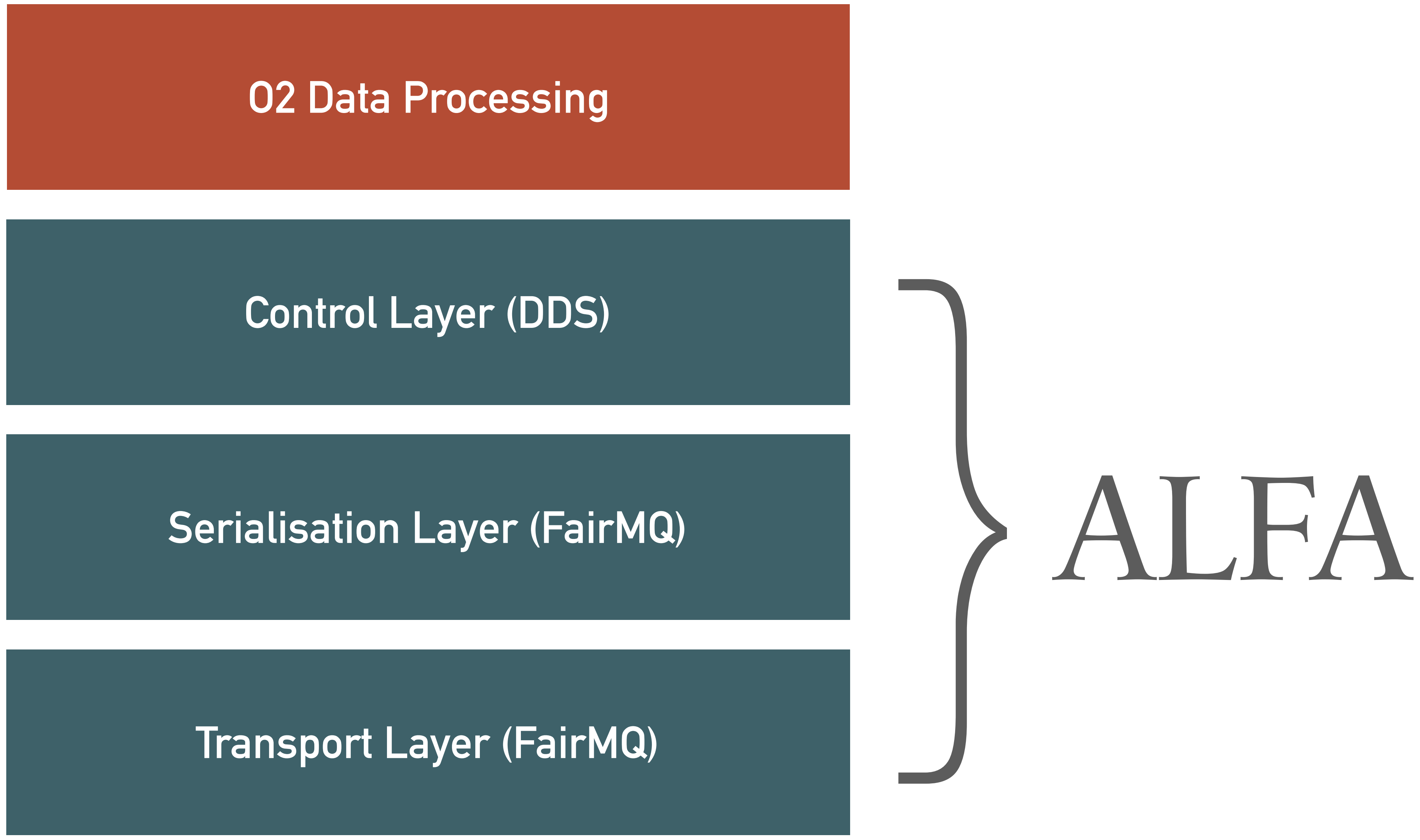
DISTRIBUTED SYSTEMS ARE HARD

There are only **two hard problems** in distributed systems:

2. Exactly-once delivery
1. Guaranteed order of messages
2. Exactly-once delivery

O2 DATA PROCESSING LAYER

Let's try to have something which simplifies the life of 90% of the users when writing code for AliceO2.



IN ONE SLIDE

FairMQ + O2 Data Model

Marry the FairMQ with the O2 Data Model in a DataFlow oriented framework, which takes advantage of the latter to hide hiccups of distributed systems to users.

- ***Define Inputs** for all the Data Processors (Tasks)*
- ***Define Outputs** for all the Data Processors*
- ***Define actual Algorithm** performed by the Data Processor on the Inputs to produce the Outputs*
- ***Run:** the framework automatically deploys the explicit topology from the declarative workflow specified above, while the user does not need to take care of setting up and connecting devices.*

DATAFLOW COMPUTING

DataFlow programming: *is a programming paradigm that models a program as a directed graph of the data flowing between operations, thus implementing dataflow principles and architecture (Wikipedia).*

Concepts date back to 1960 *(Jack Dennis & students at MIT) and have recently become "trendy" thanks to AirFlow (AirBnB), Apache Flink (Hadoop ecosystem), MillWheel & Apache Beam (Google, others) and TensorFlow (Google).*

Any resemblance to existing HEP frameworks is purely fictional.

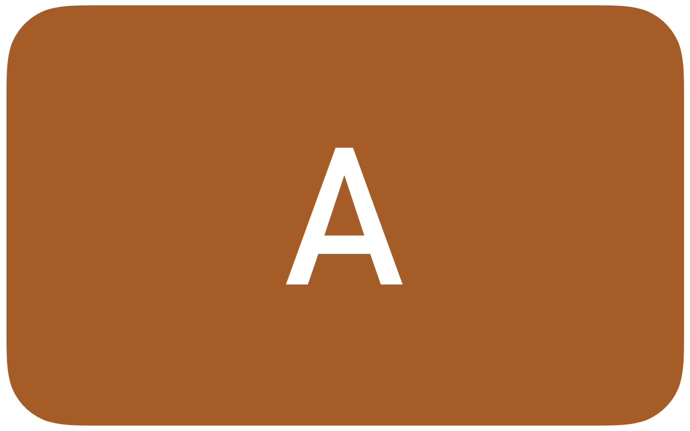
DESIGN CHOICES

- **Push, don't pull.** *Pulling means you do a get and wait for I/O. Pushing means that you get invoked only when the required I/O has been performed and all the inputs are available. The latter is typical of Reactive architectures and has the advantage that the client code is abstracted from the backend retrieving data to be processed.*
- **State should be part of the stream, not orthogonal.** *While it might not be always natural or convenient to do so (and will not be required), there is huge value in terms of robustness and parallelism to be achieved if that is the case.*
- **Purely message passing.** *The architecture should be agnostic to the fact that all the devices run on a single machine in shared memory, or 1000 using TCP interconnect. This means that either we copy data, or only one device at the time has ownership of data in a message => explicit data parallelism or timeframe pipelining to mitigate.*
- **Avoid central, edge triggered, coordination points.**

STEP BY STEP EXAMPLE (1/7)

- Declare a first Data Processor

```
DataProcessorSpec{
  "A"
};
```



STEP BY STEP EXAMPLE (2/7)

- Declare a second Data Processor

```
DataProcessorSpec{
  "B"
};
```

STEP BY STEP EXAMPLE (3/7)

- Declare inputs for the second one

```
DataProcessorSpec{
 "B",
 InputSpec{"x", "TPC", "TRACKS"}
};
```

STEP BY STEP EXAMPLE (4/7)

- Declare outputs for the first one

```
DataProcessorSpec{
 "A",
 Inputs{},
 {OutputSpec{"TPC", "TRACKS"}}
};
```

STEP BY STEP EXAMPLE (5/7)

➤ Specify code to be run on the first one

```
DataProcessorSpec{
 "A", ...,
 AlgorithmSpec{[](ProcessingContext &c) {
   c.allocator().newCollection<Track>(OutputSpec{"TPC", "TRACKS"}, 10);
 }},
};
```

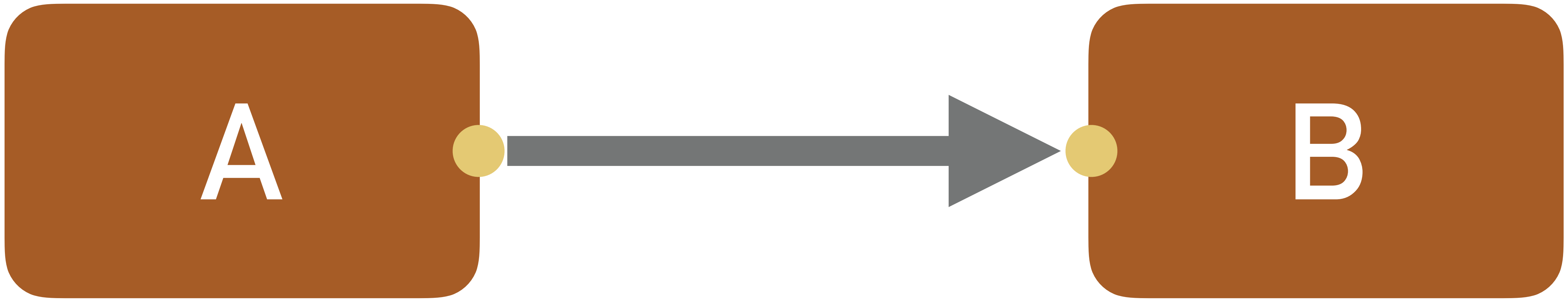
STEP BY STEP EXAMPLE (6/7)

- Specify code to be run on the second one

```
DataProcessorSpec{
 "B", ...,
 AlgorithmSpec{[](ProcessingContext &c) {
  c.inputs().get("tracks");
 }},
};
```

STEP BY STEP EXAMPLE (7/7)

- Once the workflow is completely described, the system automatically:
  - Matches inputs to outputs
  - Creates the device topology for you
  - Instantiates the topology for you (or gives you the corresponding DDS configuration).



```
#include "Framework/runDataProcessing.h"

using namespace o2::framework;

AlgorithmSpec simplePipe(o2::Header::DataDescription what) {
  return AlgorithmSpec{
    [what](ProcessingContext &ctx)
      {
        auto bData = allocator.newCollectionChunk<int>(OutputSpec{"TST", what, 0}, 1);
      }
    };
}

void defineDataProcessing(WorkflowSpec &specs) {
  WorkflowSpec workflow = {
  {
    "A",
    Inputs{},
    Outputs{
      {"TST", "A1", OutputSpec::Timeframe},
      {"TST", "A2", OutputSpec::Timeframe}
    },
    AlgorithmSpec{
      [](ProcessingContext &ctx) {
       sleep(1);
       auto aData = ctx.allocator().newCollectionChunk<int>(OutputSpec{"TST", "A1", 0}, 1);
       auto bData = ctx.allocator().newCollectionChunk<int>(OutputSpec{"TST", "A2", 0}, 1);
      }
    }
  },
  {
    "B",
    Inputs{{"a", "TST", "A1", InputSpec::Timeframe}},
    Outputs{{"TST", "B1", OutputSpec::Timeframe}},
    simplePipe(o2::Header::DataDescription{"B1"})
  },
  {
    "C",
    Inputs{{"a", "TST", "A2", InputSpec::Timeframe}},
    Outputs{{"TST", "C1", OutputSpec::Timeframe}},
    simplePipe(o2::Header::DataDescription{"C1"})
  },
  {
    "D",
    Inputs{
      {"b", "TST", "B1", InputSpec::Timeframe},
      {"c", "TST", "C1", InputSpec::Timeframe},
    },
    Outputs{},
    AlgorithmSpec{
      [](ProcessingContext &ctx) {},
    }
  }
  };
  specs.swap(workflow);
}
```

The one slide challenge:

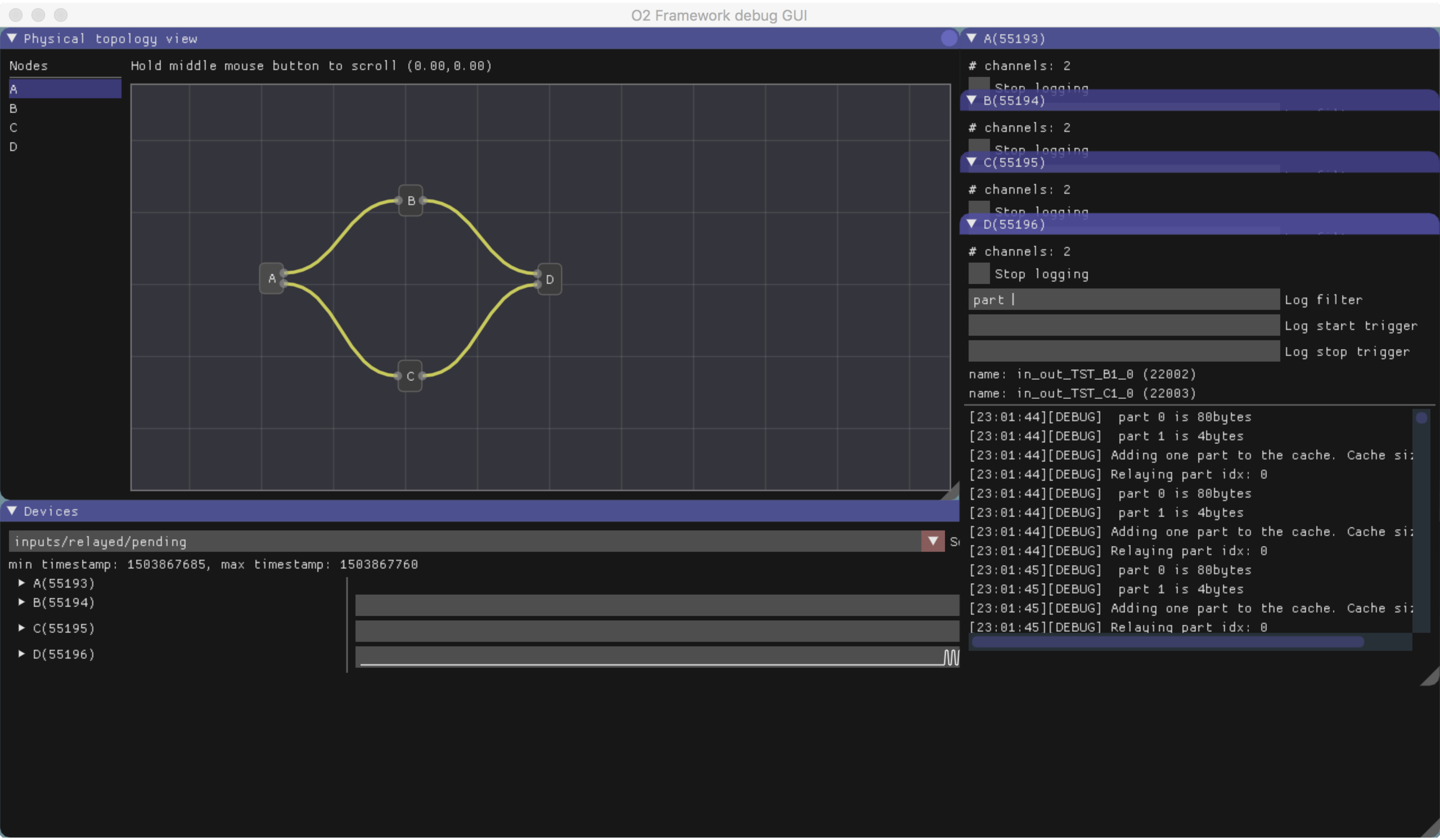
4 devices in 53 SLOC

Single executable

Debug GUI

Slide actually compiles and runs

ONE SLIDE CHALLENGE: OBLIGATORY SCREENSHOT



```
inputs/relayed/pending
min timestamp: 1503867685, max timestamp: 1503867760
```

```
D(55196)
# channels: 2
part |
name: in_out_TST_B1_0 (22002)
name: in_out_TST_C1_0 (22003)
```

```
[23:01:44][DEBUG]  part 0 is 80bytes
[23:01:44][DEBUG]  part 1 is 4bytes
[23:01:44][DEBUG] Adding one part to the cache. Cache si
[23:01:44][DEBUG] Relaying part idx: 0
[23:01:44][DEBUG]  part 0 is 80bytes
[23:01:44][DEBUG]  part 1 is 4bytes
[23:01:44][DEBUG] Adding one part to the cache. Cache si
[23:01:44][DEBUG] Relaying part idx: 0
[23:01:45][DEBUG]  part 0 is 80bytes
[23:01:45][DEBUG]  part 1 is 4bytes
[23:01:45][DEBUG] Adding one part to the cache. Cache si
[23:01:45][DEBUG] Relaying part idx: 0
```

FEATURES

Declarative workflow specification: *relies on O2 Data Model and allows implicit definition of topology by specifying inputs and outputs data types.*

Generic "DataProcessingDevice": *waits for all inputs to be available before starting processing, handles missing inputs, does caching.*

Metrics and file services: *standard APIs for "out-of-band" flow.*

Single executable driver: *for laptop usage. Different devices are still separate processes but the user sees a single entry point.*

Debug GUI: *visualise topology, view logs, show metrics, minimal control (e.g. pause logging).*

FEATURES

Simplified message creation: *includes enforcing of output constrains and manages lifetime to last for the whole invocation duration.*

- *PoD data arrays ("Collections")*
- *ROOT serialised messages (involves serialisation and copy)*
- *Raw byte buffers*

Configuration options management: *declarative wrapper to FairMQProgOptions*

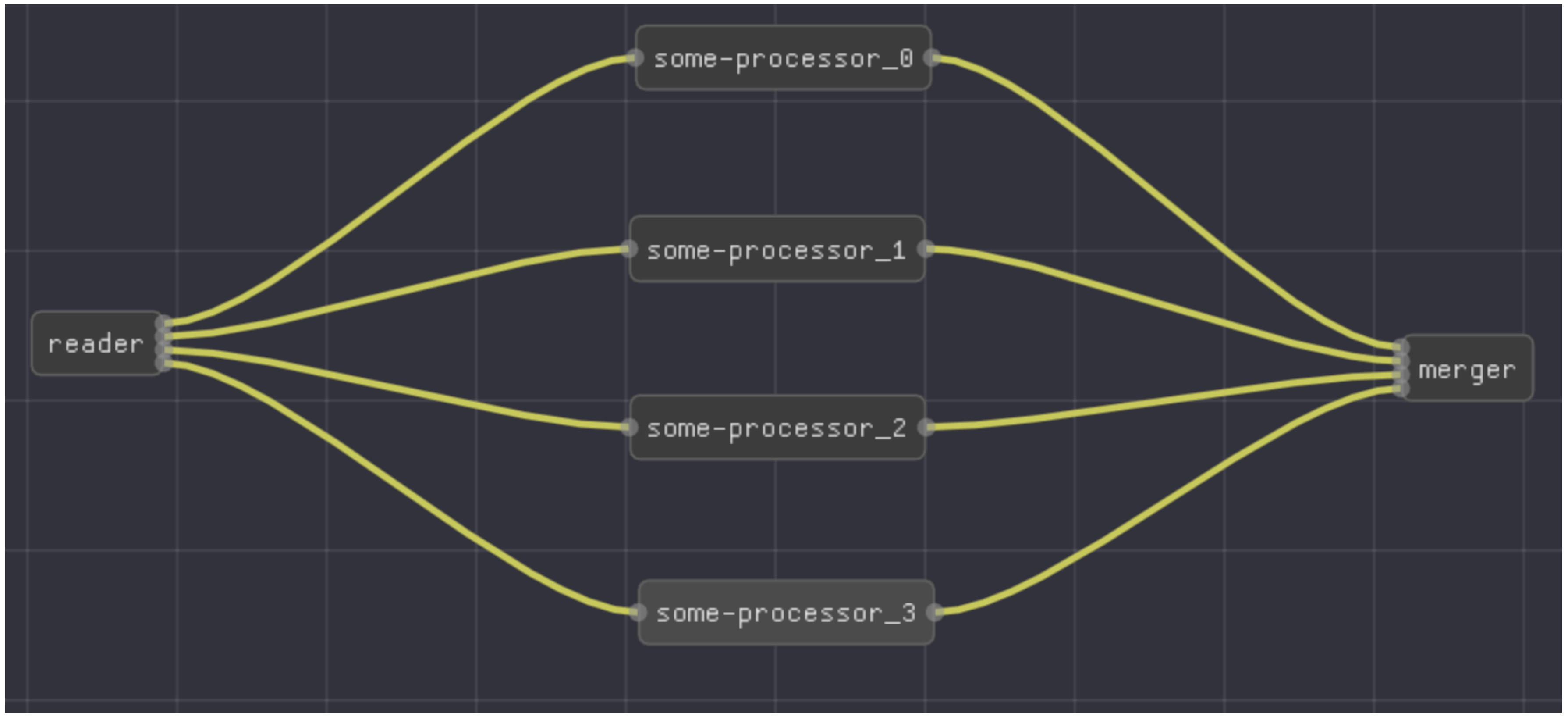
Generate **Graphviz** diagrams

Generate **DDS** configuration fragments

EXPRESSING PARALLELISM: DATA PARALLEL PROCESSING

Split data in parts and assign different subparts to a device.

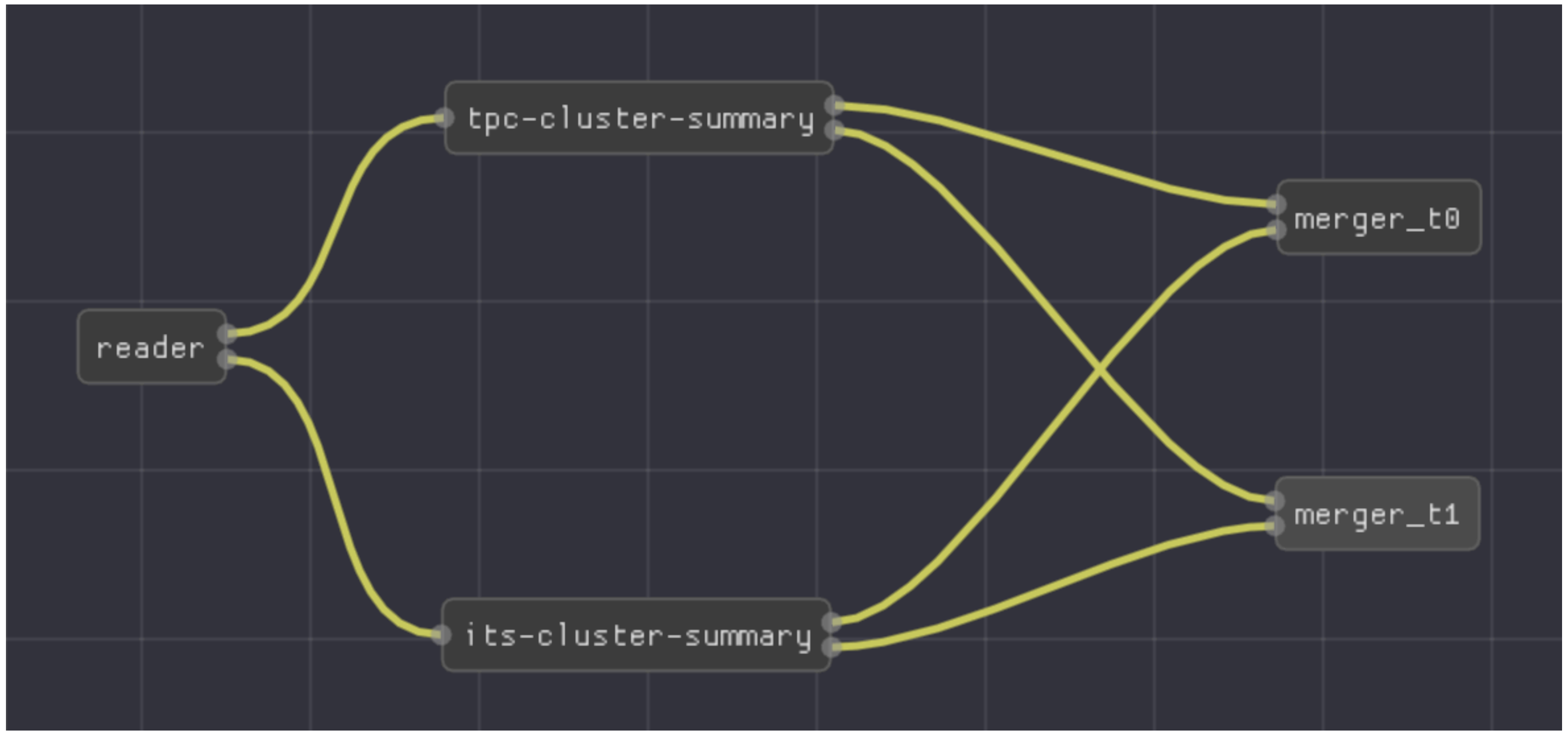
```
parallel(
  DataProcessorSpec{"some-processor",
    Inputs{{"TPC", "CLUSTERS"}},
    Outputs{{"TPC", "TRACKS"}},
    ...
  }, 4,
  [](DataProcessorSpec &spec, size_t idx) {
    spec.outputs[0].subSpec = idx;
    spec.inputs[0].subSpec = idx;
  });
```



EXPRESSING PARALLELISM: TIME PIPELINING

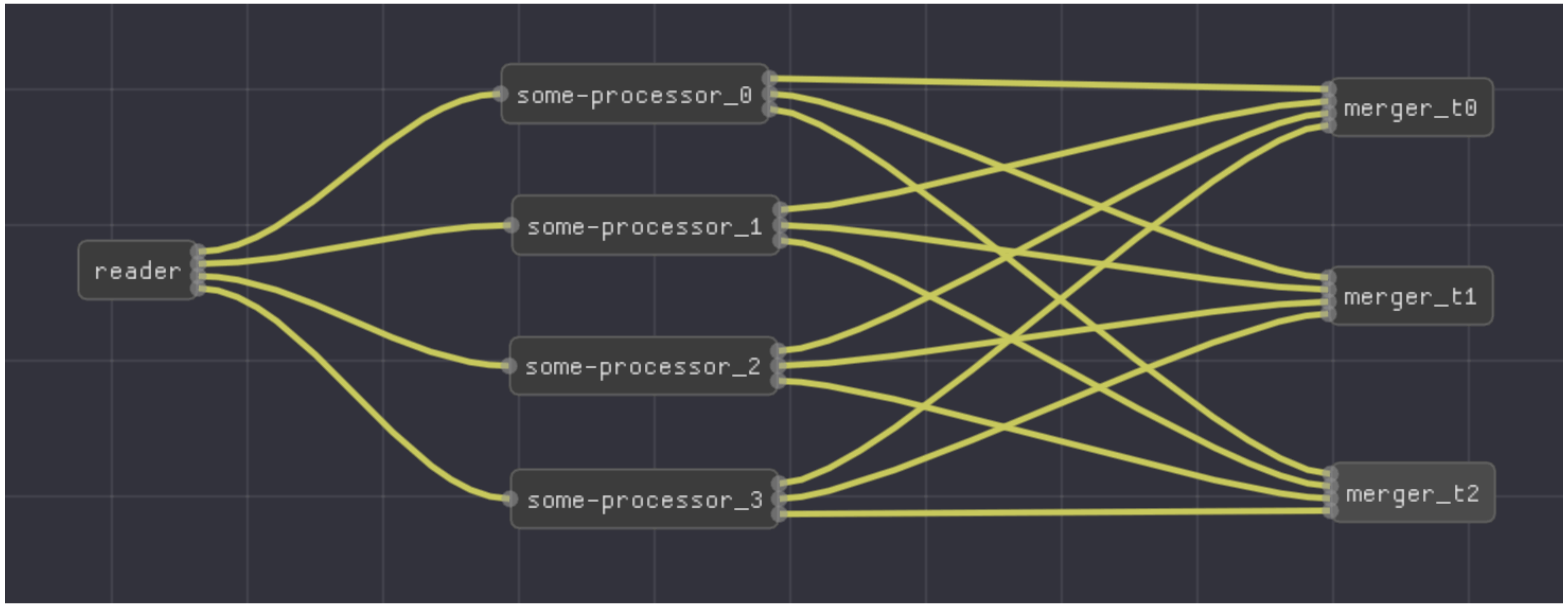
Automatically generate pipeline setups using "timePipeline" modifier in the workflow specification.

```
timePipeline(DataProcessorSpec{"merger",
    Inputs{
      {"x", "TPC", "SUMMARY"},
      {"y", "ITS", "SUMMARY"}
     },
    ...
  }, 2)
```



BRINGING IT ALL TOGETHER

With just these two primitives, complex workflows can actually be constructed and the associated topology generated automatically.



NEXT STEPS

Realistic examples & revamped O2 tutorial

I think we are at the point where we have enough features that we should get some real users and algorithms.

Interpreted configuration

Right now workflow configuration has to be compiled, but I've heard there is work being done in some obscure lab on a C++ interpreter... ;-)

Treat conditions as inputs

There is clear demand for allowing to access the CCDB directly via an explicit "get" operation. This will never be forbidden, however I personally see an advantage in letting the framework retrieve the valid conditions payloads and push them together with the data.

RANDOM IDEAS

Resource aware parallelism

E.g. if something declares itself as "data parallel", use different partitioning depending on the available resources.

Runtime optimisation of topology (will require tighter DDS integration)

- *Monitor performance of the deployed topology.*
- *Update deployed topology to optimise throughput (e.g. by increasing the pipelining of long running stages).*
- *Update deployed topology to optimise resource utilisation (if a new machine become available, redeploy accordingly).*

AliFlow, FairFlow (... HEPFlow! ;-)): *I think FairMQ provides excellent building blocks to construct an HEP oriented, C++ based DataFlow computing architecture.*

TAKE AWAY MESSAGES

- An ongoing **design document** for the Data Processing Layer is being drafted (see https://github.com/AliceO2Group/AliceO2/blob/dev/Framework/Core/README.md).
- A second pass **demonstrator** for the design document exists and it's merged in AliceO2, see code in:

  Framework/Core: *actual code*

  Framework/Core/test: *mostly unit testing and very simple topologies*

  Framework/TestWorkflows: *slightly more complicated examples*
- I am personally convinced we can use a **DataFlow computing architecture** to perform all our data processing needs and use it to efficiently use resources in a simple and elegant manner.
- **Work ongoing** to convince y'all of the above. ;-)