Evolution of the ALICE Software Framework for LHC Run 3

Giulio Eulisse (CERN), for the ALICE Collaboration
ALICE IN RUN 3: POINT2

Detector

≈3TB/s

ReadOut

FLP

FLP

FLP

FLP

SP

Synchronous reconstruction (data reduction)

up to 500GB/s

On-site storage

EPN

EPN

EPN

EPN

Asynchronous reconstruction (improved conditions)

up to 100GB/s

Permanent storage

EPN / Grid

EPN / Grid

EPN / Grid

EPN / Grid

EPN input data quantum is the "timeframe": 23ms of continuous readout data. ~10GB

BEAM ON: data reduction

BEAM OFF: improved calibration
ALICE O2 SOFTWARE FRAMEWORK IN ONE SLIDE

Transport Layer: ALFA / FairMQ

- Standalone processes for deployment flexibility.
- Message passing as a parallelism paradigm.
- Shared memory backend for reduced memory usage and improved performance.

1See "ALFA: ALICE-FAIR new message queuing based framework" by Mohammad Al Turani
Message passing aware data model. Support for multiple backends:

- **Simplified, zero-copy** format optimised for performance and direct GPU usage. Useful e.g. for TPC reconstruction on the GPU.
- **ROOT based serialisation.** Useful for QA and final results.
- **Apache Arrow based.** Useful as backend of the analysis ntuples and for integration with other tools.

- **Standalone processes** for deployment flexibility.
- **Message passing** as a parallelism paradigm.
- **Shared memory** backend for reduced memory usage and improved performance.
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
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

Since too many people did not get the joke, we started thinking how to simplify this for the user, as a result we decided to build a data flow engine (pipelines!) on top of our distributed system backend.
**Data Processing Layer (DPL)**

Abstracts away the hiccups of a distributed system, presenting the user a familiar "Data Flow" system.

- **Reactive-like design** (push data, don't pull)
- **Declarative Domain Specific Language** for topology configuration (C++17 based).
- **Integration** with the rest of the production system, e.g. Monitoring, Logging, Control.
- **Laptop mode**, including graphical debugging tools.

**Data Layer: O2 Data Model**

Message passing aware data model. Support for multiple backends:

- **Simplified, zero-copy** format optimised for performance and direct GPU usage. Useful e.g. for TPC reconstruction on the GPU.
- **ROOT based serialisation**. Useful for QA and final results.
- **Apache Arrow based**. Useful as backend of the analysis ntuples and for integration with other tools.

**Transport Layer: ALFA / FairMQ**

- **Standalone processes** for deployment flexibility.
- **Message passing** as a parallelism paradigm.
- **Shared memory** backend for reduced memory usage and improved performance.
A `DataProcessorSpec` defines a pipeline stage as a building block.

- Specifies inputs and outputs in terms of the O2 Data Model descriptors.
- Provide an implementation of how to act on the inputs to produce the output.
- Advanced user can express possible data or time parallelism opportunities.
DATA PROCESSING LAYER: IMPLICIT TOPOLOGY

Data Processing Layer

Topology is defined implicitly.
Topological sort ensures a viable dataflow is constructed (no cycles!).
Laptop users get immediate feedback through the debug GUI.
Service API allows integration with non-data flow components (e.g., Control)
4 FairMQ devices exchanging messages in a diamond topology
GUI shows state of the various message queues in realtime. Different colors mean different state of data processing.
Clicking on a node provides the log.
An embedded metrics viewer provides GUI feedback on DPL & user-defined metrics. Multiple backends supported, including of course InfluxDB (i.e., for ALICE data taking) and Monalisa (Grid deployments). See “Towards the integrated ALICE Online-Offline (O2) monitoring subsystem”, by Adam Wegrzynek.
Data is described as pushed through the pipeline.

**Timeframe Input A**

---

**Timeframe Input B**

---

**Conditions**

---

**Algorithm**

---

**Result**
REACTIVE DESIGN

Algorithm is dispatched when all the inputs are ready.
Different inputs can have different lifetimes. E.g. conditions.
When data is "late" DPL can (optionally) drop it.
Compiles into a single executable for the laptop user.

Generates DDS configuration for deployment on a farm.

Integration with O2 Control system ongoing.

1 see "DDS – The Dynamic Deployment System" poster by Andrey Lebed
2 see "Towards the ALICE Online-Offline (O2) control system" by Teo Mrnjavac
#include "Framework/runDataProcessing.h"

using namespace o2::framework;

AlgorithmSpec simplePipe(std::string const &what) {
    return AlgorithmSpec{ [what](ProcessingContext& ctx) {
        auto bData = ctx.outputs().make<int>(OutputRef{what}, 1);
    } };
}

WorkflowSpec defineDataProcessing(ConfigContext const& specs) {
    return WorkflowSpec{
        { "A", Inputs{}, {OutputSpec{{"a1"}, "TST", "A1"}}, OutputSpec{{"a2"}, "TST", "A2"}},
        AlgorithmSpec{
            [](ProcessingContext &ctx) {
                auto aData = ctx.outputs().make<int>(OutputRef{ "a1" }, 1);
                auto bData = ctx.outputs().make<int>(OutputRef{ "a2" }, 1);
            }
        },
        { "B", {InputSpec{"x", "TST", "A1"}}}, {OutputSpec{{"b1"}, "TST", "B1"}}, simplePipe("b1")},
        { "C", {InputSpec{"x", "TST", "A2"}}}, {OutputSpec{{"c1"}, "TST", "C1"}}, simplePipe("c1")},
        { "D", {InputSpec{"b", "TST", "B1"}, InputSpec{"c", "TST", "C1"}}, Outputs{}},
        AlgorithmSpec{[](ProcessingContext &ctx) {}}
    };
}
A FEW EXAMPLES
Declarative configuration allows for easy customisation: e.g. adding a (one or more) dispatchers for QA.
RECONSTRUCTION & GENERAL DATAFLOW

See "Data handling in the ALICE O2 event processing" by Matthias Richter
We are investigating about using the Data Processing Layer also for Analysis.

See "The ALICE Analysis Framework for LHC Run 3 " by Dario Berzano
PARALLEL DIGITIZATION

See "A scalable and asynchronous detector simulation system based on ALFA" by Sandro Wenzel
DPL USAGE: MUON IDENTIFIER FILTERING CHAIN

Nice demonstrator by Gabriele Fronzè for Muon Identifier (MID) filtering.
The challenges posed by Run 3 imposed to rethink ALICE Computing Architecture, blending the traditional Online and Offline roles.

The message passing ALFA Framework is the foundation of ALICE O2 Software Framework.

We built a message passing / shared memory friendly data model which minimises copy and (de-)serialisation.

Taking advantage of the O2 Data Model we build a data flow engine on top of ALFA to reduce user code and abstract away common hiccups of distributed systems.
BACKUP
TIMEFRAME

Data quantum will not be the event, but the "Timeframe".

➤ ~23ms worth of data taking in continuous readout. Equivalent to 1000 collisions. Atomic unit.

➤ ~10GB after timeframe building. Vast majority in TPC clusters.

➤ Compressed to ~2GB after asynchronous reconstruction, thanks to track-model-compression, storing clusters instead of ADC values, tailored fixed point integer format, logarithmic precision, entropy encoding.

➤ 50x the number of collisions of RUN2.

➤ All MinBias. We need to (lossly) compress information, not filter it.
A timeframe is a collection of (header, payload) pairs. Headers defines the type of data. Different header types can be stacked to store extra metadata (mimicking a Type hierarchy structure). Both header and payloads should be usable in a message passing environment.

Different payloads might have different serialisation strategies. E.g.:

- TPC clusters / tracks: flat POD data with relative indexes, well suitable for GPU processing.
- QA histograms: serialised ROOT histograms.
- AOD: some columnar data format. Multiple solutions being investigated.
Data Processing Layer: How

```
DataProcessorSpec{
  "A",
  Inputs{
    InputSpec{"a", "TPC", "CLUSTERS"}
  },
  Outputs{
    OutputSpec{"b"}, "TPC", "TRACKS"}
  },
  AlgorithmSpec{
    [](ProcessingContext &ctx) {
      auto track = ctx.outputs().make<Track>(OutputRef{ "b" }, 1);
    }
  }
}
```
DATA PROCESSING LAYER: HOW

DataProcessorSpec{
  "A",
  Inputs{
    InputSpec("a", "TPC", "CLUSTERS")
  },
  Outputs{
    OutputSpec{"b"}, "TPC", "TRACKS"}
  },
  AlgorithmSpec{
    [](ProcessingContext &ctx) {
      auto track = ctx.outputs().make<Track>(OutputRef{ "b" }, 1);
    }
  }
}
DATA PROCESSING LAYER: HOW

InputSpec

OutputSpec

AlgorithmSpec

DataProcessorSpec

"A",
Inputs{
  InputSpec{"a", "TPC", "CLUSTERS"}
},
Outputs{
  OutputSpec{"b", "TPC", "TRACKS"}
},
AlgorithmSpec{
  [](ProcessingContext &ctx) {
    auto track = ctx.outputs().make<Track>(OutputRef{ "b", 1});
  }
}
DATA PROCESSING LAYER: HOW

DataProcessorSpec{
  "A",
  Inputs{
    InputSpec("a", "TPC", "CLUSTERS")
  },
  Outputs{
    OutputSpec("b", "TPC", "TRACKS")
  },
  AlgorithmSpec{
    [](ProcessingContext &ctx) {
      auto track = ctx.outputs().make<Track>(OutputRef{ "b" }, 1);
    }
  }
}
HOW DO YOU LIMIT CONTEXT SWITCH COSTS?

We will have a number of running processes which is \(\leq\) the number of cores.

Our tasks take long on a CPU scale (seconds) thanks to the fact we treat one timeframe at the time (\(~1000\) collisions). User code runs lock free.

By describing our computation in terms of composable pipeline stages we keep door open for (eventually dynamic) NxM mapping between data processors and actual processes.

We are willing to pay an extra price for the sake of:

- Ease of deployment (microservices!)
- Crash resilience (data taking!)
- Ability to distribute over multiple nodes (HPC!)
- Flexibility (run GEANT3 + GEANT4 + FLUKA!)

Limiting factor is in any case the GPU for TPC tracking (at least for the synchronous phase).