Data Processing Layer in O2 Framework

Status quo and motivation for an O2 Data Processing Layer

FairMQ currently provides a well documented and flexible framework for an actor based computation where each of the actors listens for message-like entities on channels and executes some code as a reaction. The key component which controls this is called a `FairMQDevice` (or *device* from now on) which can use different kind of transports to receive and send messages. In the most generic case, users are allowed to have full control on the state machine governing the message passing and have complete control on how the message handling is done. This of course covers all ALICE usecases in a generic manner, at the cost of extra complexity left to the user to implement. In most cases however a simplified way of creating devices is provided, and the user will simply create its own `FairMQDevice`-derived class, which registers via the `OnData(FairMQParts &parts)` method a callback that is invoked whenever a new message arrives. This however still holds the user responsible for:

- Verifying that the required inputs for the computation are all available, both from the actual data flow (being it for readout, reconstruction or analysis) and from the asynchronous stream (e.g. alignment and calibrations).
- Create the appropriate message which holds the results and send it.
- Ensure the ease of testability, the code reuse and the proper documentation `OnData` callback. In particular there is no way to inspect which data is expected by a device and which data is produced.

This is by design, because the FairMQ transport layer should not know anything about the actual data being transferred, while all the points above require some sort of inner knowledge about the data model and the data being moved around.

The aim is to achieve the following:

- **Explicit data flow:** Input and outputs are declared upfront and can be used for documentation or for automatic topology creation (assuming the actual processing environment is known).
- **Transport agnostic data processing:** users will not have to know about the details of how the data materialises on their device, they will always be handed the full set of payloads they requested, even if they come at different time.
- **Composability of data processing:** different process functions can in principle be chained and scheduled together depending on the desired granularity for devices.

Separating data-processing from transport

For the reasons mentioned above, we propose that the one of the developments which should happen with the O2 Framework work package is the development of a “Data Processing layer” which actually knows about the O2 Data Model (and for this reason cannot be part of FairMQ itself) and exploits it to validate, optimise and correctly schedule a computation on a user specified set of inputs.

The Data Processing Layer in particular requires:

- That the inputs of each computation are provided upfront.
- That the outputs of each computation are provided upfront.
- That a time identifier can be associated to inputs

and given these premises it actually guarantees:

- That whenever a new input arrives a `CompletionPolicy` is executed to decide wether the associate record is complete. By default such a `CompletionPolicy` waits for all the specified parts to have arrived.
- That no message passing happens during the performing of the computation, but uniquely at the end.

Instanciating a workflow

The description of the computation in such the Data Processing Layer is done via instances of the `DataProcessorSpec` class, grouped in a so called `WorkflowSpec` instance. In order to provide a description a computation to be run, the user must implement a callback which return an filled `WorkflowSpec`. E.g.:

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

WorkflowSpec defineDataProcessing(ConfigContext &context) {
  return WorkflowSpec {
    DataProcessorSpec{
      ...
    },
    DataProcessorSpec{
      ...
    }
  };
};
```

See next section, for a more detailed description of the `DataProcessorSpec` class. The code above has to be linked into a single executable together with the Data Processing Layer code to form a so called driver executable which if run will:

- Map all `DataProcessorSpec` to a set of `FairMQDevices` (using 1-1 correspondence, in the

current implementation).
- Instanciate and start all the devices resulted from the previous step.
- (Optionally) start a GUI which allows to monitor the running of the system.

The `ConfigContext` object being passed to the function contains a set of user provided options to customise the creation of the workflow. For example you might want to change the number of workers for a given task or disable part of the topology if a given detector should not be enabled.

In order to specify which options are to be present in the `ConfigContext`, the user can define the extension point:

```
void customize(std::vector<o2::framework::ConfigParamSpec> &workflowOptions)
```

Describing a computation

The description of the computation in such a layer is done via a `DataProcessorSpec` class, which describes some sort of processing of a (set of) O2 Data Payloads (*payloads* from now on), as defined by the O2 Data Model, eventually producing new payloads as outputs. The inputs to the computation, the outputs and the actual code to run on the former to produce the latter, is specified in a `DataProcessorSpec` instance. Multiple `DataProcessorSpec` instances can be grouped together in a `WorkflowSpec`to the driver code which maps them to processing devices accordingly.

The `DataProcessorSpec` is defined as follows:

```
struct DataProcessorSpec {
   using InitCallback = std::function<ProcessCallback(InitContext &)>;
   using ProcessCallback = std::function<void(ProcessingContext &)>;
   using ErrorCallback = std::function<void(ErrorContext &)>;
   std::vector<InputSpec> inputs;
   std::vector<OutputSpec> outputs;
   std::vector<ConfigParamSpec> configParams;
   std::vector<std::string> requiredServices;
   AlgorithmSpec algorithm;
};
```

In the above both `InputSpec` and `OutputSpec` are like:

```
struct InputSpec {
  std::string binding;
  o2::Headers::DataDescription description;
  o2::Headers::DataOrigin origin;
  o2::Headers::SubSpecificationType subSpec;
  enum Lifetime lifetime;
  ...
};
```

where description, origin and subSpec match the O2 Data Model definition. For the moment we will consider this a one to one mapping with the `o2::Headers::DataHeader` ones. In principle one could think of a one-to-many relationship (e.g. give me all the clusters, regardless of their provenance) and the processing layer could automatically aggregate those in a unique view. This is also the semantic difference between `InputSpec` and `OutputSpec`: the former is to express data that matches a given query (which must be exact at the moment) the latter is to describe in all details and without any doubt the kind of the produced outputs.

The `lifetime` property:

```
enum struct Lifetime {
  Timeframe,
  Condition,
  QA,
  Transient
};
```

will be used to distinguish if the associated payload should be considered payload data, and therefore be processed only once, or alignment / conditions data, and therefore it would be considered valid until a new copy is made available to the device.

The `configParams` vector would be used to specify which configuration options the data processing being described requires:

```
struct ConfigParamSpec {
  std::string name;
  enum ParamType type;
  variant defaultValue;
  ...
};
```

command line / configuration options would be automatically generated by it. These are available only at init stage, and can be used to configure services. They are not available to the actual `process` callback as all the critical parameters for data processing should be part of the data stream itself, eventually coming from CCDB / ParameterManager.

Similarly the `requiredServices` vector would define which services are required for the data processing. For example this could be used to declare the need for some data cache, a GPU context, a thread pool.

The `algorithm` property, of `AlgorithmSpec` is instead used to specify the actual computation. Notice that the same `DataProcessorSpec` can use different `AlgorithmSpec`. The rationale for this is that while inputs and outputs might be the same, you might want to compare different versions of your algorithm. The `AlgorithmSpec` resembles the following:

```
struct AlgorithmSpec {
  using ProcessCallback = std::function<void(ProcessingContext &)>;
  using InitCallback = std::function<ProcessCallback(InitContext &)>;
```

```
  using ErrorCallback = std::function<void(ErrorContext &)>;

  InitCallback onInit = nullptr;
  ProcessCallback onProcess = nullptr;
  ErrorCallback onError = nullptr;
  ...
};
```

The `onProcess` function is to be used for stateless computations. It's a free function and it's up to the framework to make sure that all the required components are declared upfront. It takes as input the context for the current computation in the form of a `ProcessingContext &` instance. Such a context consist of:

- An `InputRecord` which allows retrieving the current inputs matching the provided specification.
- A `ServiceRegistry` referencing the set of services it declared as required the computation.
- A `DataAllocator` allocator which can allocate new payloads only for the types which have been declared as `outputs`.

`onProcess` is useful whenever your computation is fully contained in your input. In several cases, however, a computation requires some ancillary state, which needs to be initialised only on (re-)start of the job. For example you might want to initialise the geometry of your detector. To do so, you can use the `onInit` callback and allocate the state and pass it to the returned `ProcessCallback` as captured arguments.

E.g:

```
AlgorithmSpec{
  InitCallBack{[](InitContext &setup){
      auto statefulGeo = std::make_shared<TGeo>();
      return [geo = statefulGeo](ProcessingContext &) {
        // do something with geo
      };
    }
  }
}
```

Implementing a computation

This chapter describes how to actually implement an `AlgorithmSpec`.

Using inputs - the `InputRecord` API

Inputs to your computation will be provided to you via the `InputRecord`. An instance of such a class is hanging from the `ProcessingContext` your computation lambda is passed and contains one value for each of the `InputSpec` you specified. E.g.:

```
InputRecord &inputs = ctx.inputs();
```

From the `InputRecord` instance you can get the arguments either via their positional index:

```
DataRef ref = inputs.getByPos(0);
```

or using the mnemonics-label which was used as first argument in the associated `InputSpec`.

```
DataRef ref = inputs.get("points");
```

You can then use the `DataRef header` and `payload` raw pointers to access the data in the messages.

If the message is of a known messageable type, you can automatically get the content of the message by passing type T as template argument. The actual operation depends on the properties of the type. Not all types are supported, in order to get an object with pointer-like behavior, T has to be a pointer (T = U*).

```
auto p = args.get<T>("input");
```

The return type is

- `T const&` if `T` is a messageable type
- `T` if `T` is a `std::container` of a ROOT-serializable type
- `smart_pointer<T>` if `T` is a ROOT-serializable type and `T*` is passed

Examples:

- for messageable types there is no additional copy involved, the content is only for reading.

```
XYZ const& p = args.get<XYZ>("points");
```

- ROOT-serialized objects are automatically deserialized and returned as a smart pointer. Note that the requested type has to be pointer.

```
auto h = args.get<TH1*>("histo");
h->Print();
```

- container of ROOT-serialized objects are automatically deserialized and returned as container object.

```
auto points = args.get<std::vector<TPoint>>("points");
for (auto& point : points) {}
```

Check next section for known types. The framework will also take care of necessary deserialization.

Creating outputs - the DataAllocator API

In order to prevent algorithms to create data they are not supposed to create, a special `DataAllocator` object is passed to the process callback, so that only messages for declared outputs can be created. A `DataAllocator` can create Framework owned resources via the `make<T>` method. In case you ask the framework to create a collection of objects, the result will be a `gsl::span` wrapper around the collection. A `DataAllocator` can adopt externally created resources via the `adopt` method. A `DataAllocator` can create a copy of an externally owned resource via the `snapshot` method.

Currently supported data types for `make<T>` are:

- Vanilla `char *` buffers with associated size: this is the actual contents of the FairMQ message.
- Messageable types: trivially copyable, non-polymorphic types.
  These get directly mapped on the message exchanged by FairMQ and are therefore “zerocopy” for what the Data Processing Layer is concerned.
- Collections of messageable types, exposed to the user as `gsl::span`.
- TObject derived classes.
  These are actually serialised via a TMessage and therefore are only suitable for the cases in which the cost of such a serialization is not an issue.

Currently supported data types for `snapshot` functionality, state at time of calling `snapshot` is captured in a copy, and sent when processing is done:

- Messageable types.
- ROOT-serializable classes, serialised via a `TMessage`.
  Classes implementing ROOT’s `TClass` interface and std containers of those are automatically detected. ROOT-serialization can be forced using type converter `ROOTSerialized`, e.g. for types which can not be detected automatically
- `std::vector` of messageable type, at receiver side the collection is exposed as `gsl::span`.
- `std::vector` of pointers to messageable type, the objects are linearized in th message and exposed as gsl::span on the receiver side.

The `DataChunk` class resembles a `iovec`:

```
struct DataChunk {
  char *data;
  size_t size;
};
```

however, no API is provided to explicitly send it. All the created DataChunks are sent (potentially using scatter / gather) when the `process` function returns. This is to avoid the “modified after send” issues where a message which was sent is still owned and modifiable by the creator.

Error handling

When an error happens during processing of some data, the writer of the `process` function should simply throw an exception. By default the exception is caught by the `DataProcessingDevice` and a message is printed (if `std::exeception` derived `what()` method is used, otherwise a generic message is given). Users can provide themselves an error handler by specifying via the `onError` callback specified in `DataProcessorSpec`. This will allow in the future to reinject data into the flow in case of an error.

Services

Services are utility classes which `DataProcessors` can access to request out-of-bound, deployment dependent, functionalities. For example a service could be used to post metrics to the monitoring system or to get a GPU context. The former would be dependent on whether you are running on your laptop (where monitoring could simply mean print out metrics on the command line) or in a large cluster (where monitoring probably means to send metrics to an aggregator device which then pushes them to the backend.

Services are initialised by the driver code (i.e. the code included via `runDataProcessing.h`) and passed to the user code via a `ServiceRegistry`. You can retrieve the service by the type of its interface class. E.g. for Monitoring you can do:

```
#include <Monitoring/Monitoring.h>
// ...
auto service = ctx.services().get<Monitoring>(); // In the DataProcessor lambda...
service.send({ 1, "my/metric" }); ...
```

Currently available services are described below.

ControlService

The control service allow DataProcessors to modify their state or the one of their peers in the topology. For example if you want to quit the whole data processing topology, you can use:

```
#include "Framework/ControlService.h"
//...
auto ctx.services().get<ControlService>().readyToQuit(true) // In the DataProcessor
 lambda
```

RawDeviceService

This service allows you to get an hold of the `FairMQDevice` running the DataProcessor computation from with the computation itself. While in general this should not be used, it is handy in case you want to integrate with a pre-existing `FairMQDevice` which potentially does not even follow the O2 Data Model.

Monitoring service

Integration with the monitoring subsystem of O2 happens by getting the `o2::monitoring::Monitoring` interface. A simple example is:

```
#include <Monitoring/Monitoring.h>
// ...
auto service = ctx.services().get<Monitoring>(); // In the DataProcessor lambda...
service.send({ 1, "my/metric" }); ...
```

for the full API documentation please have a look at:

https://github.com/AliceO2Group/Monitoring

Some suffix for the metrics are reserved to represent vector and tabular metrics.

- `<some-metric-name>/n` contains the size of a vector metric at a given moment.
- `<some-metric-name>/m` contains the secondary size of a matrix metric at a given moment.
- `<some-metric-name>/<i>` where `<i>` is an integer contains the values of the i-th element in a vector metric or of the `<i>%n` column, `<i>/m` row of a matrix metric.

Callback service

A service that data processors can register callback functions invoked by the framework at defined steps in the process flow. This allows you to have customisation points for the following event:

- `CallbackService::Id::Start`: before entering the running state.
- `CallbackService::Id::Stop`: before exiting the running state.
- `CallbackService::Id::Reset`: before resetting the device.

Expressing parallelism

If we want to retain a message passing semantic and really treat shared memory as yet another transport, we need to be very careful in how to express parallelism on data, so that the “single ownership model” of message passing forces us to either duplicate streams that need to be accessed in parallel, or to serialise workers which need to access the same data. Solutions like reference counting shared memory would not be allowed in such a scenario and in any case would require extra caution and support to make sure that failures do not leave dangling reference around (e.g. when one of the parallel workers abruptly terminates). First of all let’s consider the fact that there are two level of parallelisms which can be achieved:

- Data flow parallelism: when data can be split in partitions according to some subdivision criteria (e.g. have one stream per TPC sector and have one worker for each).
- Time flow parallelism: when parallelism can be achieved by having different workers handle different time intervals for the incoming data. (e.g. worker 0 processes even timeframes, worker 1 processes odd timeframes).

In order to express those DPL provides the `o2::framework::parallel` and `o2::framework::timePipeline` helpers to avoid expressing those explicitly in the workflow.

Integrating with pre-existing devices

It can actually happen that you need to interface with native FairMQ devices, either for convenience or because they require a custom behavior which does not map well on top of the Data Processing Layer.

This is fully supported and the DPL provides means to ingest foreign, non-DPL `FairMQDevice` produced messages into a DPL workflow. This is done via the help of a "proxy" data processor which connects to the foreign device, receives its inputs, optionally converts them to a format understood by the Data Processing Layer, and then pumps them to the right Data Processor Specs.

This is done using the `FairMQRawDeviceService` which exposes the actual device on which an Algorithm is running, giving the user full control.

Customisation of default behavior

The DPL aims at solving with the default behavior the majority of common usecases. For all the special cases to be handle, we however provide multiple customisation entrypoints, in particular to:

- Change the policy for when an input record can be processed and what to do with the record afterwards. This is referred as `CompletionPolicy`.
- Change the behavior of the FairMQ channels. This is referred to as `ChannelConfigurationPolicy`.
- Change the command line options of the workflow driver.

In all cases this is done by providing a:

```
void customize(...)
```

free function which takes as argument the group of policies to be applied to customise the behavior.

Support for analysis

While not part of the initial design goal, we plan to extend DPL in order to support analysis. In particular we are evaluating a mode in which users can natively get a ROOT `RDataFrame` with an API similar to the `InputRecord` API.

Current Demonstrator

A demonstrator illustrating a possible implementation of the design described above is now found in the dev branch of AliceO2, in the Framework folder.

In particular:

- `Framework/Core` folder contains the `DataProcessorSpec` class and related.
- `Framework/Core/test` folder contains a few unit test and simple example workflows.
- `Framework/TestWorkflows` folder contains a few example workflows.
- `Framework/DebugGUI` folder contains the core GUI functionalities.
- `Framework/Utils` folder contains utilities and helpers for the creation of workflows.

There is also a few demonstrator available in particular:

- https://github.com/AliceO2Group/AliceO2/tree/dev/Detectors/TPC/workflow demonstrates the usage of DPL for TPC clusterisation and track reconstruction.
- https://github.com/AliceO2Group/AliceO2/blob/dev/Framework/TestWorkflows/src/o2SyncReconstructionDummy.cxx provides a skeleton for the synchronous reconstruction.
- https://github.com/AliceO2Group/AliceO2/tree/dev/Steer/DigitizerWorkflow provides a workflow than can do TPC digitisation.

Interesting reads

- MillWheel: Fault-Tolerant Stream Processing at Internet Scale : paper about Google previous generation system for stream processing
- Concord : Similar (to the above) stream processing solution, OpenSource.

Document history

- v0.9: proposal for approval at the O2 TB - 19th June 2018