ALICE O^2 data model proposal

Mikolaj Krzewicki Sylvain Chapeland Roberto Divia
Matthias Richter David Rohr

for the CWG4 data model group.

February 26, 2017

Abstract

This note presents a proposal for the ALICE O^2 data model. A base data layout and metadata format that allow for efficient resource use are proposed. Application of the data model to online/offline data processing and quality control is discussed.

1 Introduction

The ALICE online-offline (O^2) computing system [1, 2] is a computing facility and a software framework designed for the processing of the ALICE data in the upcoming LHC Run 3. The design aims at high data throughput and parallelism using a multiprocess model. It does not, however, exclude the use of multithreading and other forms of concurrent processing inside of individual processes.

The data exchange between processes running within the O^2 system (called O^2 devices) is taken care of by the ALICE-FAIR (Alfa) framework [3]. Since this is the main communication mechanism foreseen for data exchange, it effectively serves the role of an API between the devices. The Alfa framework provides data transport and synchronisation primitives via the FairMQ message queue library. FairMQ messages consist of raw memory buffers which are asynchronously queued and atomically delivered.

The online data processed by the O^2 system consists of data buffers originating from the detector hardware (raw data) and the processing devices (containing derived data). The data fragments belonging to a logical unit are grouped into a data set, e.g. a (sub-) time frames. A (sub-) time frame contains raw data associated to a period of data taking (typically several tens of ms, as dictated by the heartbeat trigger [4]) and/or the results of the processing of these data. In addition, any data that might be necessary to describe the data set can be added to the logical group.

In the unified online-offline software model also derived data is handled within the same software framework. O^2 devices dedicated to quality control (QC) and physics analysis tasks should use the same set of interfaces as components in the synchronous data reconstruction and calibration chain. The requirements for the data used in these tasks tend to be different from the online components: high level abstractions and ease of use (of e.g. ROOT[5] objects), despite the additional overhead, is sometimes preferred to high performance low-level data structures. Transparent support for high level data structures is part of the proposed data model.

2 Zero-copy approach

A single time frame data volume is expected to be of the order of tens of gigabytes. The data model should facilitate data exchange approaches that minimize resource use, i.e. avoid unnecessary copies of data and serialization/de-serialization overhead.

Traditionally, in monolithic designs where all processing is performed inside a single process, data exchange between logical processing units (i.e. subroutines and threads) is supported at the programming language semantics level using parameter passing (by value or by reference). The ubiquitous virtual memory model supported by most current computing architectures allows subroutines to efficiently exchange data by absolute references (pointers) to resources available in the uniform process virtual memory space. This memory model also enables fully transparent and efficient compiler support of higher level abstractions like virtual inheritance and type safety in the C++ language.

In the O^2 multi-processing model data needs to be exchanged between devices running as separate processes. In the case of devices separated by a network interface (running on different machines in a network) a data copy over the network is inevitable. The copy overhead can, however, be minimized using modern networking technologies like remote direct memory access (RDMA).

For processes running on the same machine it is possible to use shared memory segments which can be accessed directly by more than one process. This memory region is usually not covered by the standard resource allocation mechanisms provided by the language and needs to be addressed explicitly. A data copy to this memory can be avoided by constructing and manipulating the data directly there.

Shared memory segments are mapped by the operating system at arbitrary base addresses within the process virtual memory. Data access in such a memory segment therefore cannot rely on absolute references as they are valid in one process only. That excludes structures containing pointer data members and higher level language features (like virtual inheritance) as these in general are implemented using absolute references. This limitation applies to data shared via messaging systems as well, as the base address of a message buffer will be different in each process.

In cases where additional overhead is acceptable higher-level data structures can be serialized into a buffer or shared memory and then de-serialized by the receiving process. In the case of multiple devices processing the same data the same buffer is deserialized by each consumer separately. Various libraries exist that provide this functionality and try to optimize various aspects of the process, e.g. google flatbuffers[6] allow zero-copy access to data in the serialized buffer with only a minimal (indirection) overhead to ensure portability and schema evolution. Serialization, however always incurs data duplication as additional resources need to be allocated for the serialized buffer.

In view of the large volume of data that needs to be transported and processed by the multi-process based O^2 system this cost should be minimized. To achieve optimal performance the in-memory representation of data should allow a zero-copy approach. This is only possible when individual data units are contiguous in memory and do not refer to, or use, absolute addresses (pointers) or abstract language features (virtual inheritance). Should maybe conjure up a proper definition, e.g. POD type sans reference type members etc.

3 O^2 data flow

In this section: overview of the main data flow and the requirements on the data layout and formats needed to achieve the flow:

3.1 First level processors

- CRU data in FLP memory.
- Subtimeframe building.
- FLP processing including.

3.2 Event processing nodes

- Time frame building.
- reconstruction and calibration.
- compressed time frames.

3.3 Quality control (QC)

3.4 Asynchronous processing on the O^2 farm

- Event summary data.
- AOD format.
- Persistent data format(s).

3.5 Analysis facilities

- Data analysis requirements.

3.6 Requirement summary

4 Message queue based communication

The move to a multi-process, message queue based system is a large departure from the current monolithic ALICE offline data and computing model. It is, however, conceptually close to the modular processing model of the ALICE HLT. The decentralized messaging queue concept of FairMQ simplifies it by integrating the data transport layer into a library and eliminating the separation between dedicated transport and processing devices (or processes). In contrast to the managed and synchronized data flow in the HLT framework, message queue based systems are fully asynchronous. Messages are asynchronously queued on both the sending and receiving ends and multiplexed between the different connections without queue ordering guarantees. Devices receiving data are presented with an input queue where messages originating from different sources are interleaved in an non-deterministic order. For dealing with multiple logically associated data buffers, additional lower level abstractions are needed to maintain their association across transport boundaries without incurring excessive overhead.

Vectored IO (also referred to as scatter/gather IO in the POSIX specification in [7]) is an important feature when dealing with multiple data buffers as it allows, in principle, to avoid the cost associated with serializing a data set into a single IO buffer before scheduling it's transport. A form of vectored IO is provided in FairMQ as multi-part messages. A multi-part message consists of a structure (a vector) referencing multiple independent buffers. The multi-part message (and the referenced buffers) is transported atomically while preserving the ordering of the references.

The multi-part approach, in addition to minimizing the resource strain associated with IO buffer construction, also by construction reduces the need for (re-)synchronisation and event building; data fragments once associated to a single logical unit (e.g. a time frame) remain that way throughout the entire chain regardless of the data transport topology. Another benefit is that additional data parts can be attached or removed by other processing devices without copy overhead at any point of the processing chain.

5 O^2 message structure

In order to ensure consistent navigation within a data set (e.g. a time frame), each data fragment is described by metadata containing the information about e.g. the content type of the payload, it's origin and serialization strategy.

The O^2 message consists of a sequence of metadata-payload pairs of message parts contained within a multi-part message. Each payload is described by metadata contained in a separate message part. Since FairMQ preserves the ordering of the parts, the natural choice is to precede each data part with the associated metadata part in the message, as illustrated in figure 1.

Storing the metadata and data payload in separate buffers offers several advantages:

- Since the metadata is separated from the payload already at transport level, efficient navigation is possible as only the (small) metadata parts need to be inspected.
- The size of the metadata buffer is not fixed enabling a scheme with flexible metadata content.
- The content of the data buffers, once produced by the hardware or a processing device, is immutable to other devices. Since the metadata is encapsulated in a separate buffer, it is possible to add additional information to the metadata (e.g. quality control status, contained in an additional header in the header stack, see section 6) with minimal cost and without modifying the payload downstream from the data producer.

6 Metadata format

The O2 metadata consists of a contiguous buffer containing a sequence of headers (header stack). The byte representation of a header consists of a

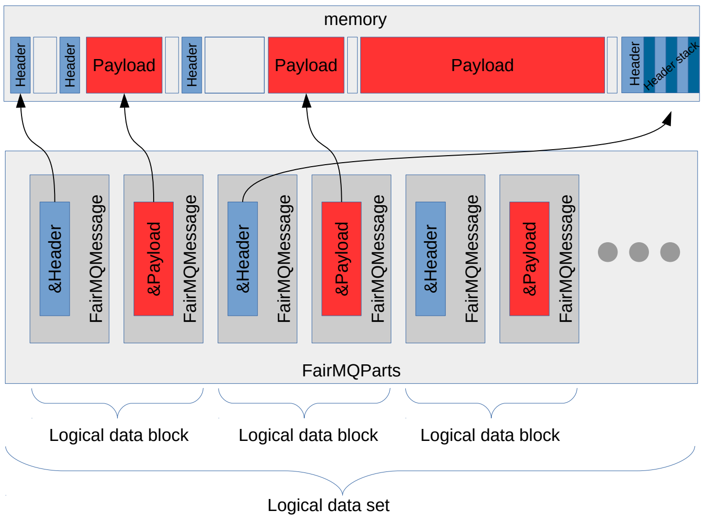

Figure 1: The O^2 message structure containing a data set. Logical data blocks consisting of payload and metadata parts are contained in a multi-part FairMQ message (bottom). The buffers associated to the message parts do not have to be contiguous in memory (top).

user defined body following a BaseHeader struct containing fields needed to:

- Verify that the following data (header body) belongs to an O^2 header.
- Define the size of the entire header.
- Flag whether another header follows this one in the stack.
- Verify the version of the header.
- Signal the type of the header.
- Signal the packing/serialization scheme of the metadata body carried by the header.

The header stack should contain at least the standard DataHeader struct describing the basic payload properties common to all payloads. The DataHeader representation starts with the BaseHeader followed by bytes representing:

- The functional data description uniquely determining the data type contained in the payload.
- The payload serialization method (e.g. ROOT, FlatBuffers, none) complementing the data description field.
- The origin of the data to identify the producer (e.g. detector system or a software subsystem).
- A data type dependent 64 bit specification. This can be used by the detectors to store e.g. the fine grained equipment ID like the link number for raw data or cluster finder instance for clusters[1].

[1]Based on HLT experience: most data types use some kind of fine grained ID. This field has been added here to avoid the overhead of a full header for what in most cases would be just one (64 bit) field.

- The payload buffer size[2].

Other headers can be defined similarly to DataHeader and included in the header stack. Examples include trigger information for triggered detector data, object name for ROOT objects used in quality control etc.

6.1 Header memory layout

The memory layout of the DataHeader struct is illustrated by the following definition:

```
struct BaseHeader
{
  uint32_t  magicString;
  uint32_t  headerSize;
  uint32_t  flags;
  uint32_t  headerVersion;
  uint64_t  headerDescription;
  uint64_t  headerSerialization;
};

struct DataHeader : public BaseHeader
{
  uint64_t dataDescription[2];
  uint32_t dataOrigin;
  uint32_t reserved;
  uint64_t payloadSerializationMethod;
  uint64_t subSpecification;
  uint64_t payloadSize;
};
```

7 Data formats

The O^2 data model does not impose any limitations on the data types exchanged between devices. The only constraint from the data transport layer is that the trasported payload must be contiguous in memory. For higher level data types it usually means that they need to be serialized which penalizes performance (to a varying degree). A trade off has to be made between flexibility and performance based on the use case for a particular data type.

7.1 Data in memory

The recommendation is to use flat POD data types where performance and/or memory usage is critical. Flat data means natually contiguous data that does not require a serialization step and does not contain any process specific run time dependent information like virtual function table pointers or pointer/reference members.

Outside of the critical path where relatively a low volume of data needs to be transported between devices, serialization schemes possibly impose acceptable overhead. The data model supports serialization schemes and

[2]This is not strictly necessary online as the transport framework keeps track of the buffer sizes. Keeping this information is the header is useful for persistent storage and debugging purposes.

facilitates the handling of serialization and deserialization of data transparently to user code.

7.2 Raw data formats

Raw data formats will be determined by the readout hardware of the detectors.

7.3 Reconstructed data formats

Reconstructed data

7.4 Persistent storage of time frame data

The in-memory data representation of header and payload buffers contains enough information to be stored on-disk directly as a sequence of buffers. The data represented by POD data are, however, not suitable for long term storage and need to be transformed to a portable and extensible format.

In the modular O^2 design only a dedicated device would handle persistent storage making the translation steps from and to peristent storage format transparent to other devices which only need to deal with the in-memory format.

7.5 Quality control formats

The Quality control (QC) devices will produce ROOT based data.

7.6 AOD format

The contents of the AOD data used for end user physics data analysis is being prepared by the physics community. Work is being done on developing the in-memory layout that will support efficient processing, pruning and distribution in the message based O^2 environment. The struct-of-arrays approach that is currently investigated also allows for transparent extensions of the data content. The persistent storage of AOD still needs to be investigated.

8 Interfaces

8.1 Metadata access interface

Data members representing (in principle) arbitrary integers (headerSize, payloadSize, subSpecification) or bitfields (flags) are directly accessible and settable as integers.

Access to other members is protected by strongly typed interfaces. Initialization can only be performed using predefined constants, consistency is then assured by the type system at compile time.

Access to header information contained in a metadata buffer is also implemented in a type safe way.

8.1.1 Example

The user may check if a given buffer contains the desired header information in a type safe way. If the desired header is of type DataHeader, a call to:

```
DataHeader* header = Header::get<DataHeader>(buffer);
```

will yield a valid pointer if a header of type DataHeader is part of the header stack inside the buffer pointed to by the buffer pointer.

8.2 Message navigation

The base functionality of a device running in the O^2 system is contained in the FairMQDevice class. This is a generic class providing all the necessary logic to support the O^2 system and data model, like control abstractions, a state machine and multi-part messaging. FairMQ support for the multi-part messaging is, however, unaware of the O^2 specific message layout and the header to payload association. In order to enforce a consistent handling of the metadata the necessary interface is implemented on top of FairMQDevice as O2device.

Each device in the O^2 system should inherit from the O2device class in order to be able to use the data model. Interfaces are provided that insure consistency in the handling of the data and associated metadata.

8.3 Examples

To construct a message, the AddMessage(...) family of methods should be used to insure correct association of metadata to payloads, e.g. in the simplest case:

```
bool AddMessage( FairMQParts& message,
                 AliceO2::Header::Block&& headerStack,
                 FairMQMessagePtr payloadPart );
```

where in a single call the header stack to payload association is made and data is appended to the message.

After the message is constructed a call to the standard FairMQ send method suffices:

```
int64_t Send( const FairMQParts& parts,
              const std::string& channel );
```

This queues the entire message for atomic delivery via a specified data channel.

For decoding the message on the receiving end functionality is provided to access the metadata and it's associated payload in a consistent way. The user implements a function that takes the metadata and the payload buffers as input, and uses that function via e.g. a call to:

```
template <typename T>
bool ForEach(
       O2message& parts,
       bool (T::*memberFunction)(
         const byte* headerBuffer, size_t headerBufferSize,
         const byte* dataBuffer, size_t dataBufferSize
       )
     );
```

Inside the user provided function the metadata and the payload are decoded and processed.

References

[1] Buncic P. and The ALICE Collaboration. *Technical Design Report for the Upgrade of the Online-Offline Computing System.* Tech. rep. ALICE-TDR-019 CERN-LHCC-2015-006. 2015.

[2] *The Alice O2 software.* URL: `https://github.com/AliceO2Group`.

[3] M. Al-Turany et al. “ALFA: The new ALICE-FAIR software framework”. In: *Journal of Physics: Conference Series* 664.7 (2015), p. 072001. URL: `http://stacks.iop.org/1742-6596/664/i=7/a=072001`.

[4] P. Vande Vyvre A. Kluge. *The detector read-out in ALICE during Run 3 and 4.* Tech. rep. DRAFT ALICE-TECH-2016-001. URL: `http://svnweb.cern.ch/world/wsvn/alicetdrrun3/Notes/Run34SystemNote/detector-read-alice/ALICErun34_readout.pdf`.

[5] URL: `https://root.cern.ch/`.

[6] URL: `https://google.github.io/flatbuffers/`.

[7] URL: `http://www.unix.org/version3/ieee_std.html`.