

ALICE O² data model proposal

Mikolaj Krzewicki Sylvain Chapeland Roberto Divia
Matthias Richter David Rohr

for the CWG4 data model group.

February 15, 2017

Abstract

This note presents a proposal for the ALICE O² 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 (O²) 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 O² system (called O² devices) is taken care of by the ALICE-FAIR (Alfa) framework [3]. Since this is the only 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 O² system consists of a set of data buffers originating from the detector hardware (raw data) and the processing devices (derived data). The data fragments are logically grouped into (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 and qualify 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. O² 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) is sometimes preferred to high performance low-level data structures. Transparent support for high level data structures is part of the proposed data model.

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

In order to ensure consistent navigation within a time frame, each data fragment is described by a small metadata block containing the information about the content type of the payload, its origin and serialization strategy. In addition, the metadata block can be extended by the processing devices with additional information without the need to modify the payload.

2 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 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 non-deterministically. 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 [6]) is an important feature when dealing with multiple data buffers as it allows, in principle, to avoid the cost associated with serializing data into a single IO buffer before scheduling its transport. A form of vectored IO is provided in FairMQ as multi-part messages. A multi-part message consists of multiple independent buffers and is delivered atomically as a single (multi-part) message while preserving the ordering of the buffers.

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.

3 O² message structure

The O² 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.

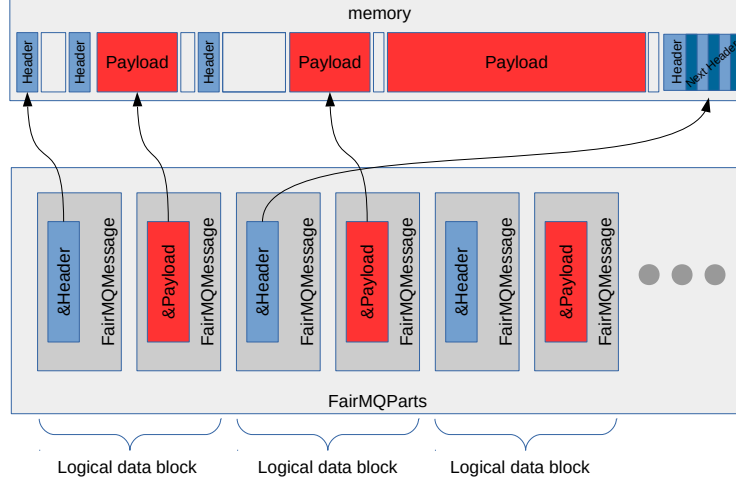


Figure 1: The O^2 message structure. Logical data blocks consisting of payload and metadata parts are contained in a multi-part FairMQ message (bottom). The buffers associated to the message parts do not have to be contiguous in memory (top).

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 4) with minimal cost and without modifying the payload downstream from the data producer.

4 Metadata format

The O^2 metadata consists of a contiguous buffer containing a sequence of headers (header stack). The byte representation of a header consists of a 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.

- 114 • Flag whether another header follows this one in the stack.
- 115 • Verify the version of the header.
- 116 • Signal the type of the header.
- 117 • Signal the packing/serialization scheme of the metadata body carried
- 118 by the header.

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

- 123 • The functional data description uniquely determining the data type
- 124 contained in the payload.
- 125 • The payload serialization method (e.g. ROOT, FlatBuffers, none)
- 126 complementing the data description field.
- 127 • The origin of the data to identify the producer (e.g. detector system
- 128 or a software subsystem).
- 129 • A data type dependent 64 bit specification. This can be used by the
- 130 detectors to store e.g. the fine grained equipment ID like the link
- 131 number for raw data or cluster finder instance for clusters¹.
- 132 • The payload buffer size².

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

136 4.1 Header memory layout

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

```

139 struct BaseHeader
140 {
141     uint32_t    magicString;
142     uint32_t    headerSize;
143     uint32_t    flags;
144     uint32_t    headerVersion;
145     uint64_t    headerDescription;
146     uint64_t    headerSerialization;
147 };
148
149 struct DataHeader : public BaseHeader
150 {
151     uint64_t    dataDescription[2];
152     uint32_t    dataOrigin;
153     uint32_t    reserved;
154     uint64_t    payloadSerializationMethod;
155     uint64_t    subSpecification;
  
```

¹Based 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.

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

```

156     uint64_t payloadSize;
157 };

```

158 4.2 Header interface

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

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

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

167 4.2.1 Example

168 The user may check if a given buffer contains the desired header informa-
169 tion in a type safe way. If the desired header is of type DataHeader, a call
170 to:

```

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

```

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

174 5 Data formats

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

182 5.1 Data in memory

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

188 Outside of the critical path where relatively a low volume of data needs
189 to be transported between devices, serialization schemes possibly impose
190 acceptable overhead. The data model supports serialization schemes and
191 facilitates the handling of serialization and deserialization of data trans-
192 parently to user code.

193 5.2 Raw data formats

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

5.3 Reconstructed data formats

Reconstructed data

5.4 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 O² design only a dedicated device would handle persistent storage making the translation steps from and to persistent storage format transparent to other devices which only need to deal with the in-memory format.

5.5 Quality control formats

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

5.6 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 O² 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.

6 Interfaces

The base functionality of a device running in the O² system is contained in the FairMQDevice class. This is a generic class providing all the necessary logic to support the O² 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 O² specific message layout. 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 O² 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.

6.1 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/alicetdrun3/Notes/Run34SystemNote/detector-read-alice/ALICErun34_readout.pdf.
- [5] URL: <https://root.cern.ch/>.
- [6] URL: http://www.unix.org/version3/ieee_std.html.