

ALICE O² data model proposal

CWG4

March 9, 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 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 O² 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 frame. A (sub-) time frame contains raw data associated to a period of data taking (typically several tens of ms corresponding to O(1000) interactions, 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. 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), despite the additional overhead, is sometimes preferred to high performance low-level data structures. Transparent support for high- and low level data structures is part of the proposed data model.

2 Zero-copy approach

A single uncompressed time frame data volume is expected to be of the order of tens of gigabytes which constitutes a significant fraction of the amount of memory foreseen for the machines in the O² system. The data model facilitates 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). Subroutines can efficiently exchange data by passing references (pointers) to resources available in the uniform virtual memory space available to a process. A flat process memory model also enables fully transparent and efficient compiler support of higher level abstractions like virtual inheritance in the C++ language, usually implemented as (hidden) references to internal metadata (e.g. a virtual method table).

In the O² multi-processing model data needs to be exchanged between devices running as separate processes. This scenario is not covered by the core language (in the case of C++) and needs to be addressed using additional abstraction layers. In the most general case the data to be shared is copied into a transport buffer a copy of which is made available to the receiving process by the operating system using various interprocess or network communication mechanisms e.g. pipes or (network) sockets. If the data structure is complex, i.e. not contiguous and/or refers to other data structures (contains pointer/reference members) it needs to be copied (serialized) to a contiguous buffer including the metadata needed to recreate the structure in the receiving process memory. The buffer then has to be de-serialized by each individual consumer.

Various libraries exist that provide this functionality and try to optimize various aspects of the serialization and de-serialization 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 CPU overhead and data duplication as additional resources need to be allocated for the target buffer.

In the case of devices separated by a network interface (e.g. 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 utilizing remote direct memory access (RDMA).

For processes running on the same machine it is possible to use shared memory which can be accessed directly by more than one process. Shared memory segments are mapped by the operating system at arbitrary base addresses within the process virtual memory. Data structures in such memory segments cannot contain absolute references (i.e. pointers) as they depend on the address space layout which in principle is different for each process. That excludes straightforward use of structures that contain reference data members and/or higher level language features (like virtual inheritance) as these in general are implemented using references to internal, process specific metadata. Shared memory regions are usually not covered by the standard resource allocation mechanisms provided by the language and memory management needs to be handled via additional libraries.

From the communication, or data transport perspective the use of either transport buffers for socket-like communication or special memory regions like shared memory is similar. In both cases a data buffer is accessed by separate processes at an arbitrary base address, imposing similar limitations on the data representation. The choice is to either serialize/de-serialize high level data for transport when the cost of this process is acceptable, or avoid data duplication and the CPU overhead by constructing the data in a fully self-contained format directly in a buffer that is suitable for the selected transport mechanism. In the latter case, since the transport mechanism is chosen at run time, all memory allocations need to be performed via the transport layer explicitly.

Additional constraints on data layout and representation are given by the machine architecture. True zero-copy approach is only possible if all machines involved in the processing have the same internal binary data representation with regards to endianness and data structure alignment as the agreed format. In this case data produced on one machine can be accessed by an other directly from the transport buffer. Otherwise data needs to be converted to a suitable binary representation; this can in principle be handled transparently by either the transport layer or the data access interface.

In view of the large volume of data that needs to be transported and processed by the multi-process based O^2 system the data exchange 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 POD type (plain old data), i.e. a scalar or a class type with no virtual base classes or functions that has no members of reference type or an array of such type. For full POD definition see e.g.[7].

A zero-copy approach introduces an explicit relationship between the data model and the transport layer. The need for explicit transport-driven memory allocation should be expressed in the data model programming interface consistently.

3 O^2 data flow

In this section: overview of the main data flow and the requirements of 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 quality control.

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 O² 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 of a connection between processing units (devices) 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 a 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 [8]) 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 its transport. Vectored IO is provided in FairMQ in the form of 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 and/or data serialization, also by construction reduces the need for (re-)synchronisation and event building; data fragments once associated to a single data set (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 to or removed from the data set by other processing devices without copy overhead at any point in the processing chain.

5 O² message structure

Each buffer associated to a single data set is described by metadata containing the information about e.g. the content type of the associated

buffer, 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. In order to allow efficient payload classification, each buffer 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.

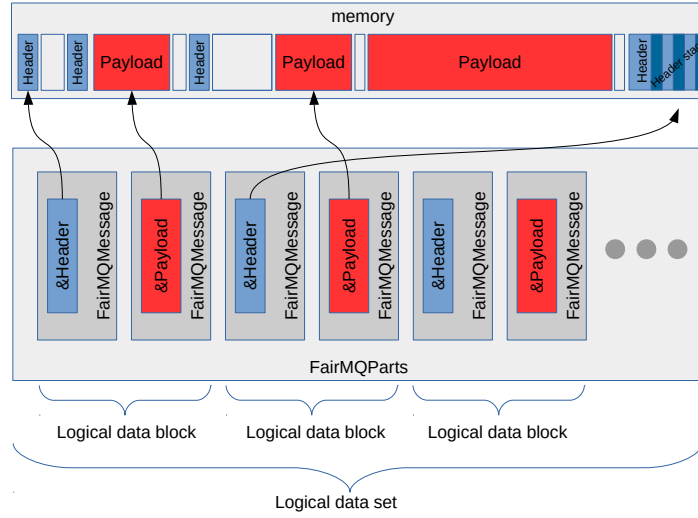


Figure 1: The O^2 message structure containing a data set. Logical data blocks consisting of payload and metadata parts are contained in a multi-part FairMQ message (bottom). The set of buffers associated to the message parts do not have to occupy a contiguous memory block (top).

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 at well defined locations in the message vector.
- The size of the metadata buffer is not fixed enabling a scheme with flexible metadata content.
- The content of the payload 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 user defined body following a BaseHeader struct containing fields needed to:

- Verify that the following data (header body) belongs to an O² 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 binary representation of BaseHeader is fixed and can not be extended to guarantee consistent decoding of header information in the future. All metadata needs to follow the bytes representing the BaseHeader. The BaseHeader struct shall never be used directly, it only aids the construction and decoding of metadata.

6.1 DataHeader

The header stack of all the data blocks 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 clusters¹.
- The payload buffer size².

The binary format of DataHeader is fixed to ensure efficient access without the need for special decoding steps. The contents can only possibly be extended by appending additional data members and incrementing the version number to ensure backward compatibility.

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

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.

6.2 Header memory layout

The memory layout of the BaseHeader and the derived DataHeader structs is compatible with 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
{
    BaseHeader 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 data must be contiguous in memory. In addition, if a zero-copy behaviour is desired, the memory buffer for the payload should be allocated by the transport layer in a memory region appropriate for the chosen transport method. 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 in the data path where performance and/or memory usage is critical, e.g. synchronous reconstruction and calibration, which needs to access and process the bulk of the data. Outside of the critical path a low volume of data needs to be transported between devices and serialization schemes possibly impose acceptable overhead. The data model supports serialization schemes and facilitates the handling of serialization and de-serialization of data transparently to user code.

7.2 Raw data formats

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

290

7.3 Reconstructed data formats

291

7.4 Persistent storage of time frame data

292

The in-memory data representation of header and payload buffers contains sufficient information to be stored on-disk directly as a sequence of buffers. Additional information can trivially be added to facilitate persistence like checksums, schema evolution information and indexes. The data that is POD in memory however, is not suitable for long term storage and need to be transformed to a portable format and possibly compressed.

298

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.

302

7.5 Quality control formats

303

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

304

7.6 AOD format

305

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.

312

8 Interfaces

313

A minimal reference set of interfaces providing type-safe construction, access and navigation of the metadata-payload protocol of the O² message is developed. This complete, but minimal working set is meant to be augmented by higher level interfaces based of the needs of the community.

317

318

8.1 Metadata access interface

319

Construction and access to data contained in the header structures is protected by strongly typed interfaces. Initialization can only be performed using predefined constants unless the data member being set allows for arbitrary values explicitly; consistency is then ensured by the type system at compile time.

324

8.1.1 Examples

325

Construction of a header stack containing an arbitrary number of headers is provided by the Header::Stack class. The constructor is a variadic template, taking as arguments references to the headers or that are to be placed in the header stack. Header::Stack is a move-only type to guarantee resource safety.

329

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

332


```

333     DataHeader* header = Header::get<DataHeader>(buffer);
334     will yield a valid pointer if a header of type DataHeader is part of the
335     header stack inside the buffer pointed to by the buffer pointer.

```

336 8.2 Message construction and navigation

337 The base functionality of a device running in the O² system is contained in
338 the FairMQDevice class. This is a generic class providing all the necessary
339 logic to support the O² system and data model, like control abstractions, a
340 state machine and multi-part messaging. FairMQ support for the multi-
341 part messaging is, however, unaware of the O² specific message layout
342 and the header to payload association. In order to enforce a consistent
343 handling of the metadata the necessary interface is implemented on top
344 of FairMQDevice as O2device.

345 Each device in the O² system should inherit from the O2device class
346 in order to be able to use the data model consistently. A strongly typed
347 interface ensures consistency in the handling of the data and associated
348 metadata.

349 8.3 Examples

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

```

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

```

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

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

```

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

```

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

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

```

368     template <typename T>
369     bool ForEach(
370         O2message& parts ,
371         bool (T::*memberFunction)(
372             const byte* headerBuffer, size_t headerBufferSize ,
373             const byte* dataBuffer, size_t dataBufferSize
374         )
375     );

```

376 Inside the user provided function the metadata and the payload are de-
377 coded 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://en.cppreference.com/w/cpp/concept/PODType>.
- [8] URL: http://www.unix.org/version3/ieee_std.html.