ALICE O^2 data model and data layout proposal

CWG4

March 23, 2017

Abstract

This note presents a proposal for the ALICE O^2 data model and the in-memory data layout. 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.

2 Data model requirements

The data processed by the devices distributed in the O^2 system comprise buffers originating from the detector hardware (raw data) and the processing devices (containing derived data). The data fragments belonging to a logical data 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. O^2 devices dedicated to quality control (QC) and physics analysis should use the same set of interfaces as devices 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 layout.

The mix of various data formats and their binary representations calls for a well defined and flexible metadata scheme which will allow for efficient description of each data part and efficient navigation through the data set.

3 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 O^2 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 (usually 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 by (hidden) references to internal metadata (e.g. a virtual method table).

In the O^2 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 e.g. 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 details 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 also not usually covered by the standard resource allocation mechanisms provided by the language and the resource 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 separate processes access the data inside a buffer starting 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 and lifetime management need to be performed by the transport layer explicitly.

Additional constraints on data layout and representation are given by the machine architecture. The approach described here is only possible if all machines involved in the processing chain have the same internal binary data representation with regards to endianness and data alignment. In this case data produced on one machine can be accessed by an other directly in the transport buffer. Otherwise the data need 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, at a cost.

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 (and in general in-buffer) representation of data should allow a zero-copy approach. This is only possible when the data contained in the buffer are flat, i.e. are of POD type (plain old data): a scalar or a class type with no virtual base classes of functions that has no members of reference type or an array of such type. For in-depth POD description 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 and life-time management should be expressed in the data model programming interface consistently.

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. In the O^2 system 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. On the output, the messages might be load balanced among the connected endpoints, depending on the messaging pattern chosen at runtime. Messaging queues (like FairMQ) present a

relatively high level interface, where connection details are not exposed, the application is completely agnostic about the origins or destinations of data as that is considered runtime configuration. Devices receiving data are presented with an input queue where messages originating from different sources are interleaved in an non-deterministic order. Output data might non-deterministically be load balanced among endpoints depending on the messaging pattern. For dealing with multiple logically associated data buffers, additional 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 it's 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 scheduled and transported atomically while preserving the ordering of the references. The multi-part approach, in addition to potentially 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.

In the case zero-copy vectored IO is not supported by the transport layer, compromises have to be made. One possibility is to construct a single message by serializing the contents of all the associated buffers into a single message buffer, this step can be performed transparently to the application.

Another option, if a zero-copy approach is needed, is to prevent the queues from multiplexing and load balancing messages to/from different endpoints by requiring socket-like point-to-point topologies; this, however, introduces a dependency of the data model on runtime configuration and excludes most of the high level features provided by current FairMQ backends, most notably the messaging patterns. Other options may of course exist.

5 O^2 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. In order to consistently support buffers containing raw data, serialized representations and flat structures while allowing for a zero-copy approach, the metadata is placed in a separate message, logically associated to the actual payload.

The O^2 message consists of a sequence of metadata-payload pairs of messages referenced from a multi-part message vector 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 trans-

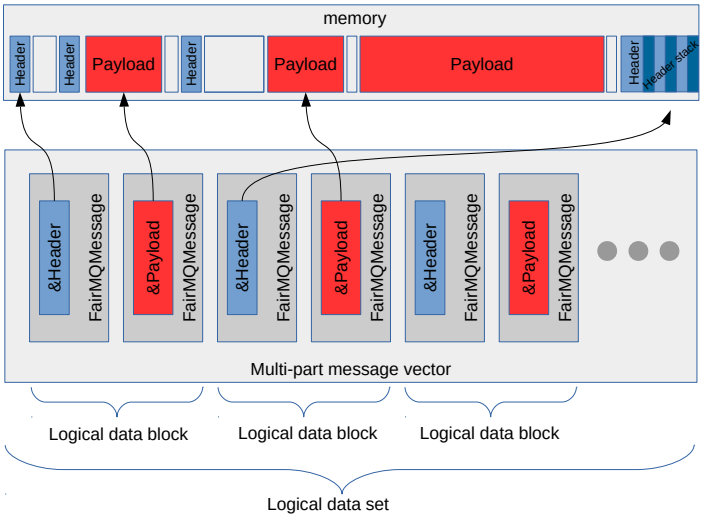

Figure 1: The O^2 message structure containing a data set. Logical data blocks consisting of payload and metadata parts are contained in a multi-part FairMQ message (bottom). The set of buffers associated to the message parts do not have to occupy a contiguous memory block (top).

port level, efficient navigation is possible as only the (small) metadata parts need to be inspected at well defined locations in the message vector.

- Metadata can be inspected separately from the payload, there is no need to potentially de-serialize the payload.
- 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 each header in the stack consists of a header 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 binary representation of BaseHeader is fixed and can not be extended to guarantee consistent decoding of header information in the future. All headers shall begin with 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 clusters[1].
- The payload buffer size[2].

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.

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;
```

[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.

[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.

```
  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 coming 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.

7.3 Reconstructed data formats

7.4 Persistent storage of time frame data

The in-memory data representation of header and payload buffers contains in principle 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.

In the modular O^2 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.

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.

Appendices

A Interfaces

A minimal reference set of interfaces providing type-safe construction, access and navigation of the metadata-payload protocol of the O^2 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.

A.1 Metadata access interface

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.

A.1.1 Examples

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.

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:

```
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.

A.2 Message construction and 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 atomic message delivery. FairMQ is, however, unaware of the O^2 specific data layout concerning the metadata and payload protocol. 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 consistently. A strongly typed interface ensures consistency in the handling of the data and associated metadata.

A.3 Examples

To construct a message, the AddMessage(...) family of methods should be used to guarantee 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 vector 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 associated payload are decoded from their binary representations 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.