DUNE FD DAQ Serialization

Brett Viren

October 11, 2019

Contents

1 Introduction and requirements **1**
- 1.1 Common requirements . . . 2
- 1.2 Requirements on transient serialization . . . 2
- 1.3 Requirements on persistent serialization . . . 2
- 1.4 Other criteria . . . 2

2 Options **2**
- 2.0.1 Hand crafted . . . 2
- 2.0.2 Casting . . . 3
- 2.0.3 Boost.serialize . . . 3
- 2.0.4 Cereal . . . 4
- 2.0.5 Protocol Buffers . . . 4
- 2.0.6 ZeroMQ messages . . . 5
- 2.0.7 ROOT . . . 5
- 2.0.8 HDF5 . . . 6
- 2.0.9 Etc. . . . 6
- 2.0.10 Other options . . . 6

3 Summary Recommendation **6**
- 3.1 Transient . . . 6
- 3.2 Persistent . . . 6

1 Introduction and requirements

The serialization of in-memory data structures to an array of bytes and back again (deserialization) is required in a number of contexts in the DAQ. These contexts roughly fall into two categories: transient and persistent serialization. The transient category refers to contexts where the serialize data is short lived. For example, when sending trigger information, run control commands or monitoring logs between processes over the local network. The persistent category refers to when the serialized data is written to (or read from) non-volatile data storage such as when "events" are saved to DAQ "raw data" output files. These two categories have some unique as well some common requirements.

1.1 Common requirements

- Should limit number of serialization methods. Serialization methods shall be deliberately selected for any given context and contexts should share the same methods where appropriate. (Basically the purpose of this document).

- Must be platform independence. Data shall be properly serialized or deserialized regardless of which of the supported architectures and operating systems are used.

1.2 Requirements on transient serialization

- Should support schema evolution. All software applications participating in parley transient serialization may be simultaneously upgraded if a schema change is required and thus strong support for schema evolution is not required. However, the serialization should provide some support so that version sheer between data and software can be at least detected.

1.3 Requirements on persistent serialization

- Must support schema evolution. It must be possible to read old data with new software. It should be possible to partially read new data with old software.

- Deserialization for life. It must be possible to read all data collected by the DAQ for at least the lifetime of the collaboration. Deserialization implementations must be fully in control of the collaboration (eg, as provided with FOSS implementations).

1.4 Other criteria

Depending on the exact context, transient serialization must optimize for latency and low computational resources for encode/decode. Persistent serialization should optimize by minimizing storage space (eg, employing compression) and supporting persistent structure to facilitate, eg, partial file reads.

The rest of this document presents a select set of serialization options and provides pros and cons. It ends with summary recommendation.

2 Options

2.0.1 Hand crafted

Not a system per se, but one possible strategy is to "hand-craft" serialization and deserialization code. In principle, all following systems may be considered hand-crafted. This DIY strategy is akin to starting a serialization system from scratch.

Pros

- requires no additional external software dependency.

- potentially could exploit features of the particular data to be serialized.

Cons

- likely amateurishly implemented compared to what is provided by established, purpose-specific projects.

- requires DAQ group to take on long term support and maintenance instead of leveraging effort from a community.

Recommendation: avoid.

2.0.2 Casting

The simplest serialization method is to cast between in-memory data structure and a byte array and back again. The byte array can then be placed into either IPC or file containers.

Pros

- no external software dependency
- very fast

Cons

- architecture differences must be explicitly handled.
- no schema evolution support (per se, one must handle this somehow in the in-memory data structure)

Recommendation: avoid unless serialization is transient and used between closely associatyed endpoints (eg, communication between threads in the same process) or for unstructured data which is inherently array-like.

2.0.3 Boost.serialize

Boost.serial is from the Boost project which develops a broadly used, high quality set of C++ library providing a large variety of functionality including algorithms and data structures destined to enter the C++ standard. Boost.serialize provides a simple method/function protocol to organize code for the serialization of C++ structures or classes. It provides an "intrusive" API whereby a `struct` or `class` is instrumented with special methods to serialize the attributes of the structure. If these attributes themselves are compound, they may be directly serialized if they too are so instrumented. Non-intrusive methods instead require free functions to be defined and for the data structure to expose sufficient attribute access.

Boost.serialize provides serialization functions for the C++ STL classes as well as *plain old data* types. It also provides support for pointer serialization such that the pointed data is serialized but once. Likewise, deserialization of multiple pointers to the same data produces just a single deserialization of the pointed data itself.

Pros

- high quality, stable code
- hooks for schema evolution

Cons

- adds Boost as an external dependency
- requires hand-instrumented low-level serialization functions

- memory resident data structure design must account for intrusiveness
- no compressed archive

Recommendation: consider for transient and partially for persistent. (lacks higher level file structure features)

2.0.4 Cereal

Cereal is a header-only C++ library which is designed to be used in C++ code in a very similar manner to Boost.serialize. It provides simpler syntax, less requirements on intrusiveness but lacks the support for pointer serialization described above.

Pros

- header-only so can be easily "vendored"
- simpler syntax than boost.serialize
- hooks for schema evolution

Cons

- no compressed archive

Recommendation: consider for transient and partially for persistent (lacks higher level file structure features).

2.0.5 Protocol Buffers

Protocol buffers (protobufs) provides object modeling. From a model, the code for a number of classes may be generated in a target language. A class provides serialization methods. Object models are defined in a simple text format different from but somewhat reminiscent of JSON. A compiler (`protoc`) is executed on this description to generate header and source file (eg, when targeting C++). Compilers to Python and other language targets exist. The compiled results may be distributed without requiring subsequent access to the compiler. A change to the description requires recompiling to produce new C++ code.

A protobuf object provides a method to determine its serialized size. This can be used to allocate a buffer which can be passed to a second method that fills that buffer with serialized bytes. Deserialization proceeds in the reverse. Given a buffer and its size, the protobuffer object can parse the bytes and populate object attributes.

Pros

- high quality code with large support and user base
- fast and with error checking
- some compression (eg, variable size values)
- no external dependency once code generation performed
- schema evolution support

Cons

- object model tends to pervade application (not a con of protobuf per se), or one must copy between application representations (some zero-copy support exists).

Recommendation: consider for all transient serialization, subject to performance testing. Consider for persistent serialization (but lacks higher level file structure features).

2.0.6 ZeroMQ messages

Like "flat binary" files, ZeroMQ messages provide rudimentary support for serialization. Unlike files, ZeroMQ messages can be subdivided into a number of "frames". Frames are effectively a sized byte string and thus serve as containers for data which is serialized in detail in some other manner. As ZeroMQ messages are effectively already serialized, they may be written to and read from files.

Pros

- general purpose, high performance
- built-in support in low overhead library

Cons

- provides few actual serialization features.

Recommendation: for ZeroMQ IPC, use framing to provide "header" values that can be used to direct the remaining "payload" data without requiring it to be decoded.

2.0.7 ROOT

ROOT files are ubiquitous but the serialization methods used can be used separate from file I/O. They work similarly to protobuf in that serialization code is generated based on an object model or by parsing C++ class definition.

Pros

- well known and supported software
- support for compression and schema evolution
- provides file structure

Cons

- large software dependency (but likely in the ecosystem in any case)
- ROOT tends to bring unwanted baggage
- binds serialization endpoints to using ROOT

Recomendation: consider it for use in persistent serialization if ROOT files will be used to hold the result.

2.0.8 HDF5

To be added.

2.0.9 Etc.

What else.

2.0.10 Other options

Not directly serialization but other projects provide potentially useful contributions to this issue.

1. Kaitai

   Kaitai provides a binary serialization description language Kaitai Struct and a compiler (`ksc`) that can generate source code in various target languages providing a parser binary data following the description. A number of supported formats exist including protobuf.

3 Summary Recommendation

3.1 Transient

Recommendations:

- use ZeroMQ message framing to provide "header" info where otherwise the full payload would have to be decoded.
- use protobuf for serialization of highly structured objects (eg, trigger info, CCM). Serialize to/from a ZeroMQ frame.
- consider boost.serial or cereal as alternative.
- use casting for large bulk-array data such as front-end computer readout via buffer interface service. Transport result in ZeroMQ frame. Break up very large arrays for transport into multiple messages.

3.2 Persistent

This section first needs clarity on DAQ's persistent event model and technology (eg, network FS vs key/value store).