Byte Blocks

Ethan Pailes

December 14, 2015

1 Abstract

Every programmer has had serialize in-memory data for storage at some point in their career. A myriad of technologies for such a common task have cropped up, but the choices for add-hoc description of low level data are relatively sparse. Byte Blocks aims to fill this gap without introducing significant semantic complexity or forcing existing code bases to change.

This paper provides an outline of Byte Blocks features, design choices, and implementation details. In addition it provides an example of how Byte Blocks can be used to interact with an existing production interface: the Apache Kafka wire protocol.

2 Introduction

Given the number of serialization technologies already in existence what justifies adding a new one to the mix?

There are several answers to this:

- Schemaless text formats like JSON or XML fit the bill for many applications, but they contain limitations.

  One of the keys to the success of text formats is their trade off of space for readability. In an era where memory and disks are cheap this seems reasonable, but it is not at all obvious that this trade off must be made. With the proper tooling low level formats can become just as accessible as text formats.

  To avoid even more cruft schemaless text formats have to leave types behind. This might be an advantage when the goal is moving fast, but is undesirable for systems with a long life span.

- There is a dearth of ad hoc data description languages, that is most data description languages concern themselves with the logical representation of data and try to hide the actual storage format from the user. In many case this hidden complexity is a good thing, but is unfortunately incompatible with the main task of programmers: gluing systems together. In order to hide the actual representation of data from the user a data framework must control both ends of any interaction. Unfortunately this assumption aligns with reality far less often than language designers would hope.

The insight that text formats can be superseded by low level data formats provided sufficient tooling is certainly not new. Google Protobufs is a low level data format which has been successful because of the tooling it provides. Similarly ad hoc data description is not new. PADS is a language which approaches the problem of describing feral data via algebraic data types.

Data serialization is never the most interesting problem that a programmer is working on. It is error prone and contains annoying subtleties, but it is not a particularly difficult problem. With this in mind Byte Blocks tries to get out of the user's way as much as possible. The user should not have to come to Byte Blocks, Byte Blocks should come to the user. The more time users spend thinking about Byte Blocks the worse a job it has done as a language.

3 Related Work

3.1 Google Protobufs

Of the existing technologies for data serialization Google Protobufs[1]is the most similar to Byte Blocks in terms of architecture and user experience. Protobufs compiles to several different target languages, generating library code to manipulate packed data structures which are described in the proto language. It is designed in such a way that it can plug into existing code bases cleanly (a single make rule is all that is needed to incorporate protobufs into a build system).

The main difference between Byte Blocks and Protobufs is that Byte Blocks aims for a clear correspondence between a given definition and the on-disk representation, while Protobufs assumes that it controls both the serialization and deserialization. The assumption allows protobufs to play games with run length encoding in order to save space. For example a four byte protobufs integer can take anywhere between one and five bytes on-disk.

Another example of how the correspondence between a byte blocks file and the on-disk representation it generates is not obvious is the way that protobufs versions fields. In order to help manage version compatibility protobufs demands that each field be given a version number. This is really good for enterprise applications, but results in a disk layout which is not at all obvious.

3.2 PADS

PADS[2]is an established ad hoc data description language hosted in C, OCaml, and Haskell. It uses algebraic data types to provide an expressive way to describe ad hoc data. PADS and Byte Blocks share the common goal of describing the on-disk or physical representation of data first and generating a sensible logical representation. This differentiates them from the vast majority of data-description languages which are primarily concerned with a clean logical representation of data.

The main drawback to PADS is the fact that it is a hosted language. This means that the language must be reimplemented if it is to add support for another target language. By contrast the architecture of Protobufs and Byte Blocks makes it easier to simply add a module to the compiler in order to add support for a new language. Additionally interfacing with a library is likely more familiar to most developers than configuring an embedded DSL.

[1]"Protocol Buffers — Google Developers." Google Developers. Web. 9 Dec. 2015. https://developers.google.com/protocol-buffers/?hl=en

[2]Fisher, Kathleen, and David Walker. "The PADS Project." Proceedings of the 14th International Conference on Database Theory - ICDT '11. Print.

4 Byte Blocks Design

4.1 What You Give: Syntax & Language Features

In the spirit of Byte Blocks coming to the user, the syntax is simple. Any byte blocks file is simply a list of blocks, each of which consists of a name and a list of fields. Each field is a name, type tuple. The following is a block which contains one example of each of the currently supported types.

```
block example_block
  bit_field_example : 16ub
  named_type_example : type_name
  sum_type_example : tag 16ub foropts 32ub | type_name
  array_example : array 32ub 16ub
end
```

The above block has a name of example_block, and four fields named bit_field_example and so on. To the right of the colon in each field is a type expression.

Bit Fields are the most primitive type in Byte Blocks. A bit field expression consists of some integer number followed by two characters indicating signedness and endianness respectively. The integer number indicates the length of the bit field. On disk bit fields take up a number of bits corresponding to the leading integer. The memory footprint of a bitfield is whatever is required for convenient manipulation by the host language. At this point all bitfields are assumed to be integer numbers. A more mature implementation would provide options for programmers to deal with bit fields which logically represent packed boolean flags or finite sets. As is the user must manually take care of such considerations in the host language. Additionally, bit fields are currently really byte fields because they only allow fields of a length which aligns along byte boundaries.

Named Types are introduced by every block declaration. This allows blocks to be composed cleanly. The above block would not compile unless another block with the name type_name had already been defined. On disk the representation of a named type is exactly the same as the representation of the block which introduced the type. That is to say Named blocks are a zero-cost abstraction over space.

Sum Types are expressed with the 'tag' and 'foropts' keywords. A tag is a field which indicates which element in the sum type is present. The final component of a sum type is the option list (a pipe separated list of type expressions). Currently the tag must be a bit field, and lands on disk directly before whichever of the options it selects. This is not as expressive as desired for ad-hoc sum types, as tags are often found in places besides directly before the actual sum type. See the future work section for further discussion.

Array is a higher order type which indicates repeated data. On disk arrays are represented by a length prefix (32ub in the above example) followed by the content type (64ub in the above example) repeated a number of times determined by the length prefix. Arrays might benefit from the same expressiveness expansion that is planned for Sum Types, though directly adjacent length prefixes seem to be more common than adjacent tags. Byte Blocks would benefit from other higher order types such as lists (null terminated sequences as opposed to length prefixed) and maps.

4.2 What You Get: Generated Code

The only currently supported target language is C, so all discussion of generated code will be

constrained to C code. Even though C does not contain methods the generated functions will be referred to as such due to the fact that each one is tightly coupled with a particular blob of state. In this sense it makes the most sense to think of the library functions generated for a given block as static methods of a single class.

Data Structure Blocks translate in a fairly direct manner into C structures. Bit fields are translated into the appropriate fixed length integer type from stdint.h, arrays are represented with pointers and length fields, and sum types consist of a tag field and a union.

Size is a utility method which takes a block structure as input and returns the number of bytes that that block will take up on disk. For constant size blocks (blocks with neither an array element nor a sum type) it returns in constant time. In fact because the generated library is a header file it is reasonable to expect size calls to be completely optimized away in such cases.

Pack is a method which takes an input block structure and packs it into a provided byte buffer. Pack is unsafe in that it assumes that the length of this buffer is at least _size of the input block. To improve usability in future versions a wrapper method which allocates the appropriate buffer before calling pack will be added.

Unpack (New) is a method which takes an input byte buffer and a target block structure. It reads from the byte buffer into the block structure, allocating the appropriate memory for arrays (if any exist) as it does so. It assumes that these arrays are not populated (passing a populated block structure into unpack results in a memory leak).

Write is a method which takes a block structure and a file handle and writes the packed block structure to the file. Write is a relatively brainless client of both _size and _pack. It allocates and deallocates a buffer if the block is not constant size. To alleviate this cost future versions of Byte Blocks will generate a batch method which only requires a single allocation.

Read (New) is a method which takes a source file handle and a target block structure. It reads a block from the file into a dynamically growing buffer, only actually parsing length prefixes and sum type tags. It is smart enough to figure out when there is a contiguous constant space set of fields between two dynamic fields (arrays and sum types). In such cases read will read in the whole block at once. Read is a client of unpack, so it allocates memory. Future version of Byte Blocks will involve a batch read as well as an event driven version which allows the user to register a handler function which gets called whenever a new record comes in.

Free is a method which does the right thing when it comes to memory safety. Any time the user calls _unpack_new or _read_new it is necessary to subsequently call _free in order to achieve memory safety. In the case of constant space blocks this method is a no-op.

5 Future Work

There is plenty of work still to be done on Byte Blocks, most of it in expanding the expressiveness of the language. The goal of Byte Blocks is to provide a way for programmers to describe low level data formats without spending too much time thinking. To this end there are several improvements worth adding to the language.

5.1 More Expressive Sum Types

Currently byte blocks assumes that the tag of a sum type is located directly before the data. This is clearly not the only way that sum types can be expressed in a data format, and is not even common enough to justify the assumption. To rectify the situation it would make sense to introduce fields into scope further on down the block so that they can be used as tags. For example:

```
block better_sumty
  the_tag : 32ub
  some_other_field : 8ul
  yet_another_field : 64s
  the_sumty : tag the_tag foropts 32sb | 64sb end
```

5.2 Syntax Improvements

Language ergonomics is very important for Byte Blocks, and the current array notation was chosen more for generality than for ergonomics. For this reason adding sugar so that

```
array_field : array 16ub 32ul
```

could be rewritten as

```
array_field : 32ul[16ub]
```

makes sense. Both of these type expressions read as, "an array of 32 bit unsigned little endian integers indexed by a 16 bit unsigned big endian integer," but the second one seems more obvious.

Another issue with Byte Blocks syntax is its lack of comment notation. Programming without any sort of comment mechanism quickly becomes cumbersome, so adding support for comments to the parser is an important step on the path to usability.

5.3 Compiler Error Improvements

The Byte Blocks compiler is currently rather prickly when it comes to reporting errors. It will identify what sort of problem there was, but does not make any effort to inform the user where in their source the issue came from. Even more important than nice syntax is a compiler which helps rather than hinders. Moving the Byte Blocks compiler from the later to the former is a priority.

5.4 Pretty Printer for Data

One of the main advantages that formats like JSON and XML have is that developers can very easily crack open a file and take a look at the data. Without a similar capability Byte Blocks will never gain traction. Walking through file contents with hexdump is not good enough. Fortunately, it seems relatively straight forward to have Byte Blocks generate a binary which can pretty print byte blocks data. Editing the file in place would still not be feasible, but reading the data would become much pleasanter.

6 Appendix A: Talking to Kafka

Apache Kafka is a distributed message queuing system. Clients send messages, essentially bags of bytes, into a cluster of Kafka servers, and pull them back out again. This interaction followes a wire protocol published on the Kafka website.[3]

Byte Blocks can be used to describe this protocol. The following is a description which is sufficient to use to query a Kafka cluster for metadata.

6.1 Byte Blocks Description of Kafka Metadata API

One quirk of the wire protocol is that all integer numbers are signed rather than unsigned. This is due to the fact that Kafka is written in Scala, and the JVM only knows about signed numbers.

All Kafka requests and responses are preceded by a 32 bit length prefix. Thus we can describe all requests as a byte array with a 32sb length prefix.

```
block wrapper
  msg : array 32sb 8u
end
```

The wire protocol includes some primitive types for describing strings and byte collections. It also include a primitive array type which has a 32sb prefix, but Byte Blocks does not yet support user defined higher order types, which means we can't make an alias.

```
block bytes
  b : array 16sb 8u
end
block string
  s : array 16sb 8u
end
```

All Kafka requests are preceded by the same header. The api_key field indicates which sort of request is being made, the api_version is currently always zero (future versions of Byte Blocks will include support for literals), and the correlation_id is a token which is used to match up requests with their responses. The client_id is a string fields which allows developers and system administrators to debug Kafka logs.

```
block request_message_hdr
  api_key : 16sb
  api_version : 16sb
  correlation_id : 32sb
  client_id : string
end
```

A meta data request is simply the standard message header followed by a list of topics that the client is interested in.

[3] https://cwiki.apache.org/confluence/display/KAFKA/A+Guide+To+The+Kafka+Protocol

```
block metadata_request
  hdr : request_message_hdr
  topics : array 32sb string
end
```

The Kafka server responds with a nested document containing telemetry on the cluster. Kafka is designed to be as decentralized as possible, so there is no one central server for the cluster. All servers in the cluster are referred to as brokers because they all server as valid entry points to a conversation with the cluster. Kafka clusters contain some number of topics meant to be used for grouping related messages. Each topic is composed of some number of partitions, which are simply giant FIFO queues running as fast as they can in parallel. Kafka is not able to avoid centralization entirely, so each partition has one broker which has been assigned to lead it. Even though the actual data for the partition might exist on a separate broker all traffic flows through the leader.

The meta data response, described as a series of nested blocks below provides information that a client needs to interact with the cluster.

```
block broker
  node_id : 32sb
  host : string
  port : 32sb
end
block partition_metadatum
  partition_error_code : 16sb
  partition_id : 32sb
  leader : 32sb
  replicas : array 32sb 32sb
  isr : array 32sb 32sb
end
block topic_metadatum
  topic_error_code : 16sb
  topic_name : string
  partition_metadata : array 32sb partition_metadatum
end
block metadata_response
  correlation_id : 32sb
  brokers : array 32sb broker
  topic_metadata : array 32sb topic_metadatum
end
```