Implementing XBF: An Efficient Self-Describing Binary Format

David Krauthamer
Electrical and Computer Engineering Department
Stevens Institute of Technology
Hoboken, USA
dkrautha@pm.me

Dov Kruger
Electrical and Computer Engineering Department
Stevens Institute of Technology
Hoboken, USA
dkruger@stevens.edu

***Abstract*—The most common data interchange formats used on the web are self-describing, and include metadata with every entry sent. Additionally, they are also encoded in plain text. Both of these traits are inefficient, and data transfer size and speed can be massively improved with a new format. This paper seeks to define and implement a specification for a new data interchange format that remains self-describing, but heavily deduplicates metadata and transfers data in binary. This allows it to be as fast, or faster, than other data interchange formats, even those that utilize binary encodings, while requiring overall less bytes sent on the wire.**

***Index Terms*—JSON, XML, Binary Encoding, Data Interchange**

I. Introduction

On the web, data is largely interchanged in one of two formats, Extensible Markup Language (XML) [1] or JavaScript Object Notation (JSON) [2]. These formats have two key features that have led to their popularity: self-description and human-readability. These features have inherent inefficiencies, which XBF (Xtensible Binary Format) intends to solve through carefully chosen compromises.

A. Self-Description

A self-describing format contains metadata and does not require a schema for incoming data, and the names and types of the data being sent can be determined from the data itself. The following is an example of XML:

```
<person>
        <name>John Jackson</name>
        <age>25</age>
</person>
```

This example represents a person object with two fields, a string name and integer age. The downside to being self-describing is wasted characters on the metadata describing the object. In this particular example, 14 characters of the 59 total characters actually contain the desired data, the rest are just describing the metadata.

B. Human-Readability

A human-readable format is one where data is encoded entirely as ASCII or UTF-8 characters, and formatted in such a way that a person could easily read or write it without the need of a machine. Human-readability brings with it two inefficiencies, the need to parse text data into a binary format that a computer understands, and additional overhead from the inherent amount of space it takes to encode data in text. For example, a 32-bit number is stored in 4 bytes natively, but a variable amount of space when in ASCII. If the number being stored is 1, then ASCII only takes a single byte (only a single digit). Once more than 4 digits is required (numbers over 9999), ASCII will always be less efficient than a natively encoded number. In a worst case scenario, 32-bit numbers can store a number up to around 4.2 billion, which would take 10 bytes to store in plain text.

C. Similar Formats

There are other self-describing formats that improve on JSON and XML in various ways that will now be examined.

1) Comma Separated Values (CSV): CSV [3] is a human-readable format ideal for sending mass amounts of data. CSV avoids sending metadata repeatedly by having a header at the top of the file which describes the names of each of the columns in the rows that follow. These rows must always contain the same number of elements as the number of columns defined in the header, which makes CSV less flexible compared to JSON and XML, which can have objects that contain objects or lists of objects.

2) Concise Binary Object Format (CBOR): CBOR [4] is a binary format which seeks to have interoperability with JSON, a fairly small message size, and tiny required code size for an encoder and decoder. Additionally, the format seeks to be future-proof and extensible such that older decoders can still decode new extended data.

3) MessagePack: MessagePack [5] is a binary format inspired by JSON but with its own type system and encoding and decoding format. All data is represented in a binary format, big-endian where necessary, and supports custom extension types that can be defined by applications.

D. Shortcomings

Each of the formats examined has its own set of shortcomings that can be improved on. CSV has the right idea of sending as little metadata as possible, but has a limited type system as a result. JSON and XML are human-readable, and thus inefficient by design. CBOR prioritizes implementation code size more than a small message size, when it should be possible to achieve both. MessagePack doesn't deduplicate metadata where possible, and sends type information with every value sent. Additionally, MessagePack utilizes big-endian formatting, while most modern processors use little-endian, requiring a conversion to take place.

Given these shortcomings, there are a number of ways they can be improved on:

- When possible, send metadata only a single time (mainly for homogenous lists).
- Allow sending metadata to be optional, such as if a client already has it.
- Utilize little-endian to reduce conversions.
- Prioritize a simple type system.

II. Format Design

A. Primitive Types

There are a total of 17 primitive types:

- Boolean
- U8, U16, U32, U64, U128, U256
- I8, I16, I32, I64, I128, I256
- F32, F64
- Bytes
- String (UTF-8)

All integers will be sent in little endian format, with the least significant bit first. For signed integers, they will be represented in two's complement format. These two formats are chosen because they are the most common way integers are interacted with in modern x86 and ARM processors, and will minimize the amount of conversion required when sending and receiving data. Similarly, floating point numbers will be sent as 32 or 64 bit IEEE 754 floating point numbers.

Strings will be sent as a sequence of bytes that correspond to UTF-8 code points. They will first send their length as an unsigned 64-bit integer (in little endian format), followed by the corresponding number of bytes contained within the string. Bytes have the same specification as strings, but with the exception that they do not have to be a valid sequence of UTF-8 code points. Lengths were chosen to be 64-bits to simplify the format, but at the cost of efficiency when sending many small sequences.

B. Primitive Metadata

Primitive metadata will be sent as a single byte discriminant value. After receiving one of these discriminant values, the client will read the following byte(s) and interpret them as the type given by the discriminant. The discriminant for each primitive type is show in Table I

Strings will always be the final value in the list. The value given to strings is used by Vectors and Structs to determine what their discriminant value will be.

C. Primitive Examples

In the following examples, each type will first be shown with its discriminant value followed by its actual value, with the byte representation underneath them.

TABLE I
Primitive Metadata Discriminants

Type	Discriminant
Boolean	0
U8	1
U16	2
U32	3
U64	4
U128	5
U256	6
I8	7
I16	8
I32	9
I64	10
I128	11
I256	12
F32	13
F64	14
Bytes	15
String	16

1) *16-Bit Unsigned Integer:*

Disc	1024	
2	0	4

2) *String:*

Disc	length								h	e	l	l	o
16	2	0	0	0	0	0	0	0	104	101	108	108	111

D. Vector Type

Vectors are a homogenous list of values that has a known, variable length. Vectors will first include their length as an unsigned 64-bit integer (the same as Strings and Bytes), followed by the corresponding number of elements. The type contained within a Vector is not sent to the client. That information is carried in the metadata.

E. Vector Metadata

Similarly to primitives, a single byte discriminant value signifying a Vector is incoming is sent. This discriminant value will be 1 greater than that of the discriminant value for Strings. Following this, metadata information for the internal type contained within the Vector will be sent. This process may continue recursively with nested types of Vectors and Structs. The length of a vector or the data contained within the vector must not be sent.

F. Vector Example

Vector of 16-Bit Unsigned Integers
DV = Discriminant of a Vector
D16 = Discriminant of a 16-Bit Unsigned Integer

DV	D16	length								42		1024	
17	2	2	0	0	0	0	0	0	0	42	0	0	4

G. *Struct Type*

A Struct is an aggregate type containing a name as well as named fields. A struct may not contain duplicate field names. Should a Struct be sent like this anyway, it is considered malformed and may not be constructed on the receiving end. Fields of a Struct are sent in sequence in the order they are listed in the Struct's metadata. When a struct is serialized it does not send any sort of name information (such as its name or field names), how many fields it has, nor does it send any type information about its fields. That information is carried in the metadata.

H. *Struct Metadata*

A discriminant value will first be sent, similarly to primitives (following the same size requirement). This discriminant value is 1 greater than that of the discriminant value for Vectors.

Following this, the name of the Struct will be sent, using the same format as primitive strings are sent (unsigned 64-bit length and then the bytes). Next, send the number of fields contained within the Struct as an unsigned 16-bit integer. A 16-bit integer is used because it's very unlikely for a struct of more than 65 thousand fields to be necessary. Finally, the fields of the Struct will be sent, first the name of the field as a String, then immediately after the metadata for the type of the field. This process may continue recursively with nested types of Structs or Vectors. These name and type pairs will be sent until there are no more fields left in the Struct.

I. *Struct Example*

The following pseudocode will be used as an example for the description of a struct:

```
struct Point {
        first: i8,
        last: i8
}

let my_point = Point {
        first: 42,
        last: -5,
};
```

The struct has a name Point, and two fields, named first and last, both of type i8. The goal is to serialize the variable `my_point`, which is an instance of Point with the values 42 and -5 for the members first and last respectively. The struct's metadata will be sent as follows:

DS = Discriminant of a Struct
DX = Discriminant of field ”first”
DY = Discriminant of field ”last”

DS	struct name length								P	o	i	n	t	fields	
18	5	0	0	0	0	0	0	0	80	111	105	110	116	2	0

first name length								f	i	r	s	t	DX
5	0	0	0	0	0	0	0	102	105	114	115	116	7

last name length								l	a	s	t	DY
4	0	0	0	0	0	0	0	108	97	115	116	7

The values inside the struct will be sent in the same order as they are listed in the struct's metadata:

first	last
42	251

III. Implementation

The reference implementation for XBF [6] is written in Rust [7]. Rust was chosen for the following reasons:

- Rust is statically typed and compiled, and allows for similarly efficient code generation to C/C++.
- A powerful type system with tagged unions built-in [8] such that dynamic dispatch and virtual method tables weren't required, and structural pattern matching was used instead.
- Rust provides statically checked memory safety through the concept of ownership [9].
- A built-in testing system in the standard library [10].
- Errors are represented as values rather than throwing exceptions [11].

A. *Values*

At the top of the hierarchy of types is a generic value, named the XbfType. This is a tagged union of all possible types, consisting of Primitives (XbfPrimitive) III-B, Vectors (XbfVec) III-C, and Structs (XbfStruct) III-D. The corresponding metadata is the XbfMetadata, a tagged union of metadata types of each of the aforementioned subtypes. In order to perform a downcast on either of these top level types, Rust's structural pattern matching must be used.

XbfType		
XbfPrimitive	XbfVec	XbfStruct

XbfMetadata		
XbfPrimitiveMetadata	XbfVecMetadata	XbfStructMetadata

B. *XbfPrimitive*

The implementation of primitives is a tagged union of all possible primitive types, mapping the XBF type to the Rust native equivalent as shown in Table II. The exceptions are the 256-bit integer types, which do not have an analog, and are instead represented by an array of four unsigned 64-bit integers.

TABLE II
XBF TYPE MAP

XBF Type	Rust Type	C++ Equivalent Type
Boolean	bool	bool
U8	u8	uint8_t
U16	u16	uint16_t
U32	u32	uint32_t
U64	u64	uint64_t
U128	u128	uint64_t[2]
U256	[u64; 4]	uint64_t[4]
I8	i8	int8_t
I16	i16	int16_t
I32	i32	int32_t
I64	i64	int64_t
I128	i128	uint64_t[2]
I256	[u64; 4]	uint64_t[4]
F32	f32	float
F64	f64	double
Bytes	Vec<u8>	std::vector<uint8_t>
String	String	std::string

Metadata for integers is an enumeration of the possible primitive metadata discriminant values, ranging from 0 to 16, represented as unsigned 8-bit integers.

C. *XbfVec*

XbfVec is implemented as a native dynamic array of XbfType values, and a Rc (reference counted, shared pointer) to the metadata of its inner type, represented as the base XbfMetadata. In order to ensure XbfVec remains homogenous, it requires the metadata to be passed in at construction time. If all the values given to the constructor do not have the same metadata as the metadata passed in, the constructor returns an error type.

XbfVec			
Metadata	Vec<XbfType>		
Rc	*XbfType	Length	Capacity

The metadata is implemented with a Rc because Rust's tagged unions are not allowed to be recursive, as then they could be infinitely sized on the stack. A pointer type has a known sized value, and as such the tagged union can then be created on the stack by the compiler. The recursion comes from the base XbfMetadata type, which can contain XbfVecMetadata, which contains an XbfMetadata, and the recursion can continue.

XbfVecMetadata could have been implemented with an owning pointer (known as a Box<T>), however that would mean that the metadata couldn't be shared among many instances of an object. When XbfVec is deserialized, it passes the internal metadata to each of the inner values it deserializes and constructs, which causes the reference count to increase and all the inner values to share the same heap allocated metadata. While this does not affect the on the wire performance, this does reduce memory usage for a user of the library.

D. *XbfStruct*

XbfStruct is implemented in a very similar manner to XbfVec, consisting of a Rc to the metadata of the struct, and a Box<[XbfType]> containing the fields. This type is known as a boxed slice. In Rust, a slice is a pair of a pointer to contiguous memory and a length. When a slice is boxed, it is allocated on the heap, and the box object is the sole owner of that heap allocation, and will de-allocate it when the box goes out of scope. The reason a boxed slice was chosen instead of a dynamic array was to save on memory usage.

XbfStruct		
Metadata	Box<[XbfType]>	
Rc	*XbfType	Length

A Vec<T>(dynamic array) in Rust consists of a pointer to memory, a capacity of how much memory is available at that address, and a length of how many elements have been properly initialized. The amount of fields of a struct are not allowed to change, so a boxed slice can be used instead of a dynamic array, thereby saving the size of a pointer of memory for every XbfStruct created.

The metadata of structs (XbfStructMetadata) is the most complex of the three metadata types. It consists of boxed string slice and an IndexMap [12] mapping boxed string slices to XbfMetadata types.

In Rust there is an important distinction between an owned String, and a string slice [13]. A string slice is similar to the previously mentioned slices, except it consists of valid UTF-8 code points. It is a view into a block of memory, and its size cannot be modified. If growable string is required, that's where the String type is used. It makes the same UTF-8 guarantee as a string slice, but has the same properties as a Vec where it keeps track of its capacity, and will reallocate if its capacity is exceeded, similar to a dynamic array. As with the decision to use a boxed slice in XbfStruct, a boxed string slice is utilized to save on memory usage, as the metadata for a struct won't need to change, thereby making a capacity field irrelevant.

String		
*u8	Length	Capacity

&str (string slice)	
*u8	Length

An IndexMap is a hash table with consistent order and fast iteration. Ordinarily a hash table does not keep track of the order it's fields are inserted, which is a problem when serializing the metadata, as the fields could be sent in any order. A dynamic array of objects consisting of the field name and type could be used, however this leads to $O(n)$ searches for a given field. In order to retain $O(1)$ searches, an IndexMap was used. It requires extra memory overhead to keep track of the insertion order of keys and values, but ensures that iteration is fast and in a consistent order.

XbfStructMetadata		
Box<str>		IndexMap<Box<str>, XbfMetadata>
*u8	Length	Fields omitted for brevity

E. Implementation Testing

The reference implementation was developed in a test driven manner, taking advantage of the built-in testing facilities of Rust. Individual unit tests would first be written with the desired behavior for an object or function, and then code would be written to satisfy the test. This approach leads to more reliable software. When adding a new feature, if anything from old features is broken tests will fail, indicating something has gone wrong. Unit tests also caught a number of implementation bugs which were noticed immediately because of the immediate feedback.

In addition to the unit tests, a number of integration tests, tests designed to test the public API (Application Programming Interface), were implemented to ensure the experience of using the library was as expected. The combination of these approaches has led to 100 percent code coverage, such that every line of source code of the library has been executed at least once by a test.

IV. Evaluation

XBF was evaluated against a number of other self-describing data interchange formats.

- CSV [14]
- MessagePack [15]
- CBOR [16]
- JSON [17]
- XML [18]

The test consisted of two components, a client and a server, both written in Rust. The server downloaded a year of Sony stock data from Yahoo Finance [19] in CSV format, and parsed it into a list of objects. Next, the original size of the CSV and a calculated value for how much memory is taken up by the parsed list was logged. Finally, the server waited for connections, and depending on the request received, serialized the list of objects into the requested format and sent it to the client. The serialized data was not cached, and the serialization was performed for each request.

The client performed the time measurement, as well as initiated connections to the server. For each data format, the client sent 100 requests to the server, and recorded the time it took from initiating the connection to receiving all the data. The average of the time for these 100 requests was then recorded, along with how large each response received was in bytes.

The server was a DigitalOcean Droplet [20] running Ubuntu 20.04.5 located in New York City. The client was a laptop running openSUSE Tumbleweed located in Hoboken, NJ.

V. Results

Original CSV data size recorded by the server: 17,160 bytes.
Native data size recorded by the server: 14,558 bytes.

TABLE III
Average Time and Bytes Read

Format	Avg Time (ms)	Bytes Read
CSV	18.93	16,411
MessagePack	11.22	15,565
CBOR	16.95	25,507
JSON	21.91	31,180
XML	21.87	43,699
XBF	11.32	14,686

TABLE IV
Format Overheads

Format	Overhead (bytes)	Percent Overhead
CSV	1,823	11.1
MessagePack	977	6.22
CBOR	10,919	42.8
JSON	16,592	53.2
XML	29,111	66.6
XBF	98	0.67

See Table III and IV for additional results. Overhead is calculated as native data size recorded by the server subtracted from bytes sent for a format.

VI. Conclusions

A. CSV Size Discrepancy

The first result to address is the difference between the number of bytes in the original CSV file and the number of bytes received by the client when asking for CSV data. Upon examination, it was found that the original CSV always contained 8 total digits of precision, whereas when it was serialized only the number of digits required to represent a number was sent. As an example, in the original data the number 83.5 would be represented as 83.500000, but when serialized would omit the trailing zeroes, as they were unnecessary. This difference only serves to increase the performance of CSV in the test, so was left as is.

B. Time Performance

XBF performed significantly faster than all other formats except for MessagePack, in which both formats performed in about the same amount of time. This is likely due to both formats working with binary data, which avoids needing to do complex and expensive conversions between plain text and native binary formats, as well as requires sending less data in general.

The XBF reference implementation prioritized correctness, memory usage, and performance, in that order. Most optimization time was put into reducing the number of heap allocations made as they are relatively expensive. Given more time the program could be profiled to discover where the most execution time is spent, and optimizations made based on that to improve runtime performance even further.

C. Size Performance

Of all the formats tested, XBF was able to send the least amount of bytes to express the same amount of data. This is despite some of the optimizations XBF is missing compared to the next most efficient format, MessagePack. MessagePack allows for the conversion of numbers to as small of a size as possible before being sent over the wire. For example, if an unsigned 64-bit number with a value of 1024 was to be sent, it would be downgraded to an unsigned 16-bit number as no information is lost from doing so. MessagePack also has different string and array types with different integer length types, allowing fewer bytes to be sent when these variable length types don't have many elements. As an example, an array with less than 65,536 elements would be sent as an array 16, instead of an array 32. Both of these optimizations weren't implemented in the XBF specification or reference implementation for the sake of simplicity, but could be added in a future revision.

XBF requiring the least amount of bytes, as well as having the smallest overhead, indicates that the primary technique of metadata deduplication worked as intended. This could be implemented in other formats as well, including the human-readable formats such as JSON and XML.

In its current state, XBF would likely make a good candidate for use in low bandwidth scenarios, such as IoT (Internet of Things) systems. It could also have applications in extremely high volume situations, where even a difference of a few kilobytes in the amount of data sent could add up to gigabytes saved very quickly.

References

[1] "Extensible markup language (xml) 1.1 (second edition)," Aug. 2006. [Online]. Available: https://www.w3.org/TR/2006/REC-xml11-20060816/

[2] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format," RFC 8259, Dec. 2017. [Online]. Available: https://www.rfc-editor.org/info/rfc8259

[3] Y. Shafranovich, "Common format and mime type for comma-separated values (csv) files," Oct. 2005. [Online]. Available: https://www.rfc-editor.org/rfc/rfc4180

[4] C. Bormann and P. Hoffman, "Concise binary object representation (cbor)," Dec. 2020. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8949.html

[5] S. Furuhashi, "Messagepack specification," apr 2021. [Online]. Available: https://github.com/msgpack/msgpack/blob/master/spec.md

[6] D. Krauthamer, "Xtensible binary format (xbf)." [Online]. Available: https://github.com/XtensibleBinaryFormat/XBF/tree/main/xbf_rs

[7] "Rust." [Online]. Available: https://www.rust-lang.org/

[8] S. Klabnik and C. Nichols, *The Rust Programming Language*, Dec. 2015, ch. 6. [Online]. Available: https://doc.rust-lang.org/book/ch06-00-enums.html

[9] ——, *The Rust Programming Language*, Dec. 2015, ch. 4. [Online]. Available: https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html

[10] ——, *The Rust Programming Language*, Dec. 2015, ch. 11. [Online]. Available: https://doc.rust-lang.org/book/ch11-00-testing.html

[11] ——, *The Rust Programming Language*, Dec. 2015, ch. 9. [Online]. Available: https://doc.rust-lang.org/book/ch09-00-error-handling.html

[12] bluss, "Indexmap." [Online]. Available: https://github.com/bluss/indexmap/tree/master

[13] S. Klabnik and C. Nichols, *The Rust Programming Language*, Dec. 2015, ch. 8.2.

[14] A. G. (BurntSushi), "rust-csv." [Online]. Available: https://github.com/BurntSushi/rust-csv

[15] E. S. (3Hren), "Rmp - rust messagepack." [Online]. Available: https://github.com/3Hren/msgpack-rust

[16] Enarx, "ciborium." [Online]. Available: https://github.com/enarx/ciborium

[17] D. T. (dtolnay), "Serde json." [Online]. Available: https://github.com/serde-rs/json

[18] J. T. (tafia), "quick-xml." [Online]. Available: https://github.com/tafia/quick-xml

[19] yahoo! Finance, "Sony stock history." [Online]. Available: https://query1.finance.yahoo.com/v7/finance/download/SONY?period1=1659398400&period2=1690934400&interval=1d&events=history&includeAdjustedClose=true

[20] "Droplets." [Online]. Available: https://docs.digitalocean.com/products/droplets/