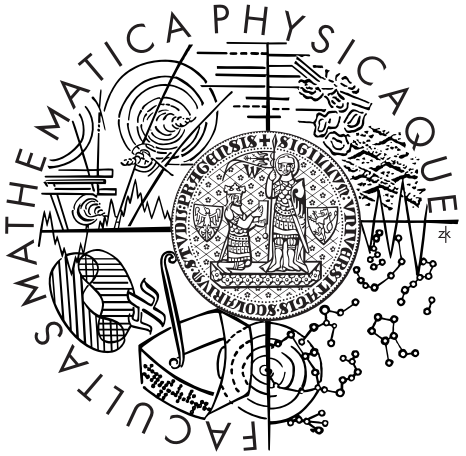
Charles University in Prague

Faculty of Mathematics and Physics

BACHELOR THESIS



Pavel Kalvoda

Implementace a evaluace protokolu CBOR

Department of Software Engineering

Supervisor of the bachelor thesis: RNDr. David Bednárek, Ph.D.

Study programme: Computer Science

Specialization: General Computer Science

Prague 2015

First and foremost, I would like to thank my supervisor, Dr. David Bednárek, for his steady support and the overwhelming amount of his feedback that has been invaluable for improving this thesis.

None of my work would be possible without the foundations build with the monumental effort of all my predecessors and peers, both known and unknown. I would like to thank the open source community for the tools and services I was provided with.

I want to thank thank my proof readers, Julia Hezinova and David Hewitt, for their tireless help with this text. I also owe my gratitude to Petr Bělohlávek, my friend and fellow student, for the sheer amount of his help.

Finally, I am grateful to my family, who have supported me through my studies. *Thank you!*

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague, May 21, 2015 signature

Název práce: Implementace a evaluace protokolu CBOR

Autor: Pavel Kalvoda

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. David Bednárek, Ph.D.

Abstrakt: CBOR (RFC 7049) je nový protokol pro binární serializaci dat. Nevyžaduje definici schémat a je podobný JSON či BSON. Přichází s několika novými koncepty, především explicitní podporou proudového zpracování a rozšiřitelnou sémantikou. Tato práce předkládá robustní generickou implementaci pro C odpovídající standardu a její rozhraní pro jazyk Ruby. Implementace i sám protokol jsou kriticky analyzovány na teoretickém i experimentálním základu. Z měření plyne, že implementace výkonem překoná relevantní alternativy, přestože zachovává bohaté rozhraní. CBOR nabízí srovnatelnou nebo lepší efektivitu a funkcionalitu vzhledem k alternativám; systémy s omezenými zdroji i ty s vysokou propustností by z něho mohly benefitovat. Některé jeho aspekty se ovšem ukázaly být nepraktické či zbytečně složité, proto jsou předloženy korekční návrhy. Je popsáno a zhodnoceno i několik dalších možných přístupů k implementaci.

Klíčová slova: CBOR, serializace dat, binární reprezentace

Title: Implementation and evaluation of the CBOR protocol

Author: Pavel Kalvoda

Department: Department of Software Engineering

Supervisor: RNDr. David Bednárek, Ph.D.

Abstract: CBOR (RFC 7049) is a recent binary schema-less data serialization protocol similar to JSON and BSON. It introduces several novel concepts, such as explicit streaming support and extensible semantics. A robust, generic, standard-compliant implementation for C is developed and presented, along with its binding for the Ruby language. Both the implementation and the protocol itself are critically evaluated by both experimental and theoretical inquiries. The measurements show that the implementation achieves performance superior to relevant alternatives while still providing a rich set of features. CBOR offers functionality and efficiency on par with or superior to the alternatives. Both high-volume applications and constrained node systems might benefit from CBOR. Some of its features, however, were shown to be needlessly complex or impractical. Suggestions addressing these deficiencies are presented. Several other approaches to implementing CBOR are proposed and discussed.

Keywords: CBOR, data serialization, binary representation

Contents

Chapter 1

Introduction and related work

CBOR (RFC 7049) is a recent binary schema-less data serialization protocol similar to JSON and BSON. It introduces several novel concepts, such as explicit streaming support and extensible semantics.

In this chapter, we present a short overview of this thesis, followed by an inquiry into the technical and historical background of the CBOR format. We explain the motivation for introduction of a new data serialization format, as well as a detailed comparison with similar relevant standardized formats.

The aforementioned background information will then serve as a basis for explaining our motivation and setting the goals for both our implementation and the subsequent investigation of the CBOR format itself.

The second chapter consists of an in-depth analysis of the CBOR format, its existing implementations, and several notable implementations of other formats. In the third chapter, we describe the design and technical details of our implementation. The fourth chapter showcases how the implementation may be used. The fifth chapter consists of benchmarks and evaluations of the implementation. In the sixth chapter, we discuss the findings from the previous chapters and propose possible improvements, as well as address the suitability of CBOR for different domains. Finally, the seventh chapter summarizes the whole thesis and presents a perspective for the future.

1.1 An overview of the CBOR format

1.1.1 General information and background

CBOR [7] aims to be a compact, schema-less data serialization format. Being akin to MessagePack [22], it strives to provide a simple, regular representation of existing data formats used in today's Internet standards while maintaining simplicity of encoders and decoders.

1.1.2 Design objectives

The following extract from the original RFC [7, s. 1.1] provides a succinct overview of the design goals:

1. The representation must be able to unambiguously encode most common data formats used in Internet standards.
2. The code for an encoder or decoder must be able to be compact [...].
3. Data must be able to be decoded without a schema description.
4. The serialization must be reasonably compact [...].
5. The format must be applicable to both constrained nodes and high-volume applications.
6. The format must support all JSON data types for conversion to and from JSON.
7. The format must be extensible, and the extended data must be decodable by earlier decoders.

1.2 Comparison to similar formats

Despite the fact that the CBOR standard declares different goals and design criteria than all of the formats mentioned in this section, they still share significant overlaps in functionality and usage. A brief overview of the key similarities and differences will be helpful both to those evaluating different alternatives from

a practical standpoint as well as to those trying to understand the theoretical motivations and limits.

The formats were selected based on the following criteria:

- Genericity – is the format reusable across many different applications and domains?
- Standardization – is there a formal specification or an official reference implementation available?
- Adoption – is the format used in real-world applications?
- Absence of schemata – does the format require payloads to be defined upfront?[1]

We intentionally include both binary and text-based formats, as many of them are commonly used together either in cooperation or as alternatives. XML is included predominantly because it is the de facto standard for generic cross-platform data serialization, with a multitude of very mature implementations readily available, and can therefore serve as a reference for the other formats. We do acknowledge that, due to its nature, it is not comparable to the JSON family in most aspects.

1.2.1 Relevant aspects

In order to meaningfully compare and contrast the formats, we have chosen a set of high-level aspects that are likely to interest most prospective users. These are

- Character and origin
- Available data types and constructs
- Extensibility
- Availability and support
- Functionality
- Common applications & domains

[1] This option is available for most of the listed formats. It is, however, not mandatory.

1.2.2 JSON

1.2.2.1 Character and origin

JSON (JavaScript Object Notation) is a simple text-based data-interchange format that originated from a subset of the JavaScript programming language. The format itself is independent from text encoding (either UTF-8, UTF-16, or UTF-32 are allowed) [12, s. 8.1].

1.2.2.2 Available types and constructs

JSON's constructs are based on the primitive types of JavaScript. These include: signed integers, floats (excluding `NaN` and `Infinity`) [12, s. 6], text strings, arrays, objects (hashmaps with string keys), and three literal values (`true`, `false`, `null`).

It is worth noting that the range of representable integers, as well as the float precision, are not defined by the standard. Application developers are often forced to circumvent this limitation by using application-specific semantics and conversions [2].

1.2.2.3 Extensibility

There is no standard mechanism for future extensions.

1.2.2.4 Availability and support

JSON is very widely supported by both ubiquitous JavaScript engines and a wide variety of standalone codecs for virtually all platforms and languages. The official web page alone lists some 178 implementations [36].

1.2.2.5 Functionality

No additional functionality is defined by the standard. Non-standard extensions for streaming exist [61].

1.2.2.6 Common applications & domains

JSON is used extensively in interactive web applications as a more efficient alternative to XML [64]. Data stores based on JSON emerged as a result of the wide adoption of web-related technologies, prominent examples being Apache CouchDB [3], RethinkDB [50], and MongoDB [5].

It is also commonly reused in other protocols and formats. For example, JSON-RPC [27] is a remote procedure call protocol based on JSON serialization.

1.2.3 XML

1.2.3.1 Character and origin

XML (Extensible Markup Language) is a part of the W3C technologies suite. It is an extensions of SGML (Standard Generalized Markup Language [33]) intended for general use.

XML is a text-based format with strong affinity towards Unicode encoding. It has a flexible, tree-like structure that can be specified and extended using *XML schemata*. All common data structures can be easily expressed using XML.

A standard mechanism of linking XML documents, as well as other types of data, is available. XML also supports verbatim, unescaped sections and comments. A binary-encoded version is partially standardized [38].

1.2.3.2 Available types and constructs

XML does not specify data types per se. Instead, it has a general hierarchical structure with text-only data. The application itself must define the mapping between the XML document tree and its data types. The format itself does not deal with specifics of e.g. numeric types.

1.2.3.3 Extensibility

The self-describing nature of the format is sufficient for expressing any structured textual data. Binary data are generally not supported.

DTDs (Document Type Definitions) and XSDs (XML Schema Definitions) allow for standardized, publicly shared declarations of element types.

1.2.3.4 Availability and support

XML is extremely widespread, with both generic and application-specific codecs available for all major platforms and languages.

1.2.3.5 Functionality

The declarative, self-describing nature of XML could be considered a functionality, since it allows almost arbitrary extensions of the data model.

Streaming is available via the loosely defined family of SAX (Simple API for XML) implementations [13].

1.2.3.6 Common applications & domains

XML is often used as a base for deriving custom formats. Examples include SVG for vector graphics [20], Office Open XML for office productivity applications [48], and SOAP for remote procedure calls.

XML is also often used for storing application configuration and saved state of applications. Many of these documents are, at least to some degree, generated automatically from other data.

Finally, early interactive web application used XML as an integral part of the XMLHttpRequest API for asynchronous communication. This usage has been largely superseded by JSON (see 1.2.2.6).

1.2.4 CBOR

1.2.4.1 Character and origin

CBOR (Concise Binary Object Representation) is a fairly recent (2013) schemaless binary data serialization format. It is designed for a wide variety of use cases.

1.2.4.2 Available types and constructs

CBOR contains primitive data types similar to those of JSON (detailed in 2.1.1), with the addition of binary data. The precision and semantics of numeric types are thoroughly defined by the specification. It also supports some commonly used

singular values, such as `NaN, null, undefined`. Content metadata are available as well.

1.2.4.3 Extensibility

Extensibility both on the standard level and the application level is a part of the core design goals of CBOR. The standard specifies how the format will be extended, including considerations for forward- and backwards-compatible applications. Content metadata 'tags'[2] can be publicly shared via a centralized repository, mimicking the DTD mechanism of XML. Rigid message definitions (CDDLs, currently a IETF draft specification [60]) are also available.

1.2.4.4 Availability and support

CBOR availability does not match that of JSON or XML. The official web page lists several implementations [8], most of which declare themselves to be either experimental or purpose-specific.

1.2.4.5 Functionality

Unlike text-based formats, CBOR can provide effective lazy decoding features. Seek-to and selective decoding is available in fixed-length documents. Streaming support is given a thorough consideration in the specification. Binary data type enables clients to embed resources efficiently, without escaping or encoding.

1.2.4.6 Common applications & domains

Although CBOR has many potential use cases (as discussed in section 6.3), there are no known existing production-stage applications yet.

1.2.5 BSON

1.2.5.1 Character and origin

BSON (Binary JSON) is a conceptual extension to JSON that was originally developed as a data serialization format for the MongoDB database [44]. It is

[2]Please refer to subsection 2.1.1.9 for more details.

binary and schema-less, while largely mirroring the structure of JSON in other areas.

1.2.5.2 Available types and constructs

BSON supports all the structures found in JSON, and adds fixed-width numerals, binary content, and several constructs specific to database applications. These include: Iterator pointer, UUID, and so called ObjectID (MongoDB's primary key type).

1.2.5.3 Extensibility

There is no standard mechanism for future extensions. No ad-hoc extensions are known at the time of writing.

1.2.5.4 Availability and support

Besides its use in MongoDB and the products related to it, BSON has codecs for several major platforms and languages. It is more widely supported than CBOR, but significantly less widespread than JSON or XML.

1.2.5.5 Functionality

BSON does not offer any additional features.

1.2.5.6 Common applications & domains

There are very few applications other than MongoDB that use BSON .

1.2.6 MessagePack

1.2.6.1 Character and origin

MessagePack (often abbreviated as MsgPack) is a lightweight, schema-free binary format. It had a strong influence on the design CBOR; many CBOR constructs share structure with their MsgPack counterparts.

1.2.6.2 Available types and constructs

MessagePack offers all the structures found in JSON, with the addition of binary content and fixed-width numerals. The length of all elements is fixed. There is an ambiguity between binary and text data that remains unresolved and implementation-specific [12, E.2].

1.2.6.3 Extensibility

There is no standard mechanism for future extensions.

1.2.6.4 Availability and support

There are mature implementations available for most major platforms and languages [23].

1.2.6.5 Functionality

Streaming is not explicitly mentioned in the specification. The reference Ruby implementation supports streaming [25] to a certain degree.

1.2.6.6 Common applications & domains

MessagePack is often used for RPC-like applications and distributed computing protocols. It is also a popular choice for compressed structured logging (including Fluentd, the *syslog*-like log collector), and is supported by Redis, the popular data structure server.

1.2.7 UBJSON

1.2.7.1 Character and origin

UBJSON (Universal Binary JSON) is yet another binary evolution of JSON. It was introduced to reduce the size of JSON payloads sent over the network while still maintaining the simplicity of JSON.

1.2.7.2 Available types and constructs

UBJSON's data model is a one to one copy of JSON. Binary data was excluded on purpose [BZ, s. 5] to maintain the isomorphic mapping.

1.2.7.3 Extensibility

There is no standard mechanism for future extensions.

1.2.7.4 Availability and support

UBJSON arguably has the least adoption of all the listed formats. Generic codecs for some major languages are available, although none of them is documented.

1.2.7.5 Functionality

UBJSON allows streaming arrays and maps. It also features *no-op*, a data item with no meaning that is intended to be used during long network polling sessions as a means of preventing automatic network timeouts.

1.2.7.6 Common applications & domains

There are no known existing production-stage applications.

1.3 Motivation

In the previous section, we have have discussed several formats that are, to some degree, similar to CBOR in terms of functionality and design. We have to ask ourselves what exactly sets CBOR apart from them, and what will be the significance of these differences for real-world applications.

To grasp the matter in a structured and succinct manner, we will present three separate sections with the reasons why exploring the CBOR format is a worthwhile endeavor.

1.3.1 Unique features

The CBOR format has a unique set of features that is not found in any other of the aforementioned alternatives. The slow adoption of seemingly similar formats suggests there is a mismatch between the expectations of the general audience and the features of available formats. On the other hand, the vast number of attempts to create an efficient universal JSON-like data serialization format suggests that there is demand for such a format.

CBOR offers an approach that is rather similar to the existing ones, which means it will be easy to understand and work with for professionals with background in other technologies, yet it also also delivers several key features that have not been combined together yet.

This combination could be useful, especially in the context of ubiquitous Internet-enabled mobile devices, where efficiency and versatility of application-level communication protocols still remains one of the limiting factors due to both constrained network resources and limited power.

	Streaming	Lazy dec.	Bin. content	Format ext.	User ext.
JSON	Ad-hoc	No	No	No	No
XML	SAX	No	No	N/A	Yes
CBOR	Yes	Yes	Yes	Yes	Yes
BSON	No	Yes	Yes	No	No
MessagePack	Ad-hoc	Partial	Yes	No	No
UBJSON	No	No	Optional	No	No

Table 1.1: Feature comparison matrix

1.3.1.1 Streaming

With the widespread adoption of the WebSockets API for HTTP clients [29] and the inclusion of the SPDY protocol features [6] in HTTP/2.0, it is clear that streaming will play a crucial role in the technological foundation of the next-generation interactive and real-time web applications. Unlike the alternatives, CBOR provides a solid foundation for building streaming-based applications.

1.3.1.2 Lazy decoding

There are many performance optimizations lazy decoding can enable. For example, both JSON and MessagePack are often used for storing structured log data. Searching these data often involves decoding of large sequences. Counted format with lazy decoding allows us to deserialize only the relevant parts, presumably leading to substantial performance gains.

1.3.1.3 Binary content

Binary content support is an important feature since it enables binary resource embedding. This is especially useful when building JSON-based web APIs for environments where multiple network transmissions may be expensive (i.e. mobile devices). The gain in efficiency may noticeably decrease power consumption.

```
{
  "entry": [
    {
      "id": "15342888",
      "hash": "5eb6f7e271e6c6b46d08c9b86a2b7443",
      "requestHash": "5eb6f7e271e6c6b46d08c9b86a2b7443",
      "profileUrl": "http://gravatar.com/pavelkalvoda",
      "preferredUsername": "pavelkalvoda",
      "thumbnailUrl": "https://secure.gravatar.com/avatar/5eb..",
      "photos": [
        {
          "value": "https://secure.gravatar.com/avatar/5eb6...",
          "type": "thumbnail"
        }
      ],
      "name": [],
      "displayName": "pavelkalvoda",
      "urls": []
    }
  ]
}
```

Listing 1.1: A typical REST API response from the Gravatar service which provides users' profile images based on their emails. Most clients will make a second request to fetch the image from `thumbnailUrl`. Embedding the thumbnail content directly could save the unnecessary connection.

1.3.1.4 Extensibility

Unlike most of the alternatives mentioned in section 1.2, the CBOR standard defines future extension points and migration paths. Should it become widely used, this aspect will be of immense value when the specification is updated or extended.

1.3.2 Compactness & performance

Unlike BSON or UBJSON, CBOR empathizes efficient encoding of common values. This leads us to believe that it will be more space-efficient than the alternatives. This presumption is analyzed and verified in section 5.2.

1.3.3 Simplicity of integration

Based on the survey of CBOR's features and constructs conducted in section 1.1, it is clear that CBOR is – in the sense of semantics – a superset of JSON. This will enable the potential users to easily leverage their existing knowledge of JSON, as well as greatly simplify the integration of CBOR with any existing JSON-based tools and applications.

1.4 Goals

Having discussed the background and the motivation, we will now outline the goals of this thesis. While we have discussed the reasons why CBOR might have the potential to become widely adopted, it will not be until we have carefully examined it in more detail that we will be able to support our speculations by measurements and a comprehensive analysis. More precisely, this thesis should examine

- the space efficiency of CBOR,
- the encoding and decoding performance that can be achieved,
- the resources required for encoding and decoding, and
- the extent to which CBOR is suitable for various applications niches,

all while critically reflecting on the design decisions made in the standard with respect to the aforementioned aspects.

At the time of writing, there is no general purpose implementation for the C language. As Foreign Function Interfaces (FFIs) for C are available for virtually all languages, meaning that a C implementation would be remarkably easy to reuse, we will proceed to implement a generic CBOR library in C. The goals are

- full standard compliance,
- well-structured and easy to use API, including the streaming features,
- verifiable correctness,
- flexible and safe memory management,
- robustness and safety (since the inputs will often be produced by external sources), and
- performance that at least matches that of popular JSON implementations

Furthermore, we realize that the target application niches for CBOR often make use of productivity-oriented languages rather than systems languages such as C. We will therefore create a binding that will connect the aforementioned library with Ruby, a popular programming language often used for developing web applications. We will strive for

- an easy to use streaming API,
- an idiomatic design that will enable CBOR to be used as a drop-in replacement for e.g. JSON, and
- seamless interoperability with other libraries across all common runtime environments.

Chapter 2

Analysis

In this chapter, CBOR is analyzed from the implementation viewpoint. This includes a detailed survey of its data types, features, and semantics, as well as an exploration of the existing implementations and the techniques they employ. We also entertain the possibility of using state machine compilers or other similar tools that are commonly used for implementing new formats and protocols.

2.1 A closer look at CBOR

In order to better grasp the structure of CBOR, a brief overview of all its constructs is provided, along with examples, including several corner cases or degenerate inputs.

2.1.1 Data types

CBOR refers to a semantic unit of data as **data item**s. There are eight **major types** of data items [7, s. 2.1]. Data items of the same major type share the same **major type byte**, which serves as a header used for determining the major type. Items of the same type have largely similar structures[1].

The semantics of CBOR data items are defined in terms of bits and bytes, a byte being 8 bits. All well-formed items consist of 1 or more full bytes of data. The length of most data items can be determined from their headers.

To prevent any possible confusion, we will use the standard C-like notation

[1] With the exception of major type 7

for numbers throughout this text. Numerals starting in `0x` are to be interpreted as hexadecimal, those starting in `0b` as binary. The terms 'low-order bits' and 'high-order bits' is used with their standard meaning, low-order bit being the leftmost ones in the natural representation.

2.1.1.1 Diagnostic notation

Since CBOR data items represented by series of bytes are not easily read by humans, the CBOR standard introduces a diagnostic notation to express the data items. The notation follows the JSON convention, making it quite natural. Several examples are presented in listing 2.1; full description of the notation is available in the RFC [7, s. 6].

```
An indefinite array of numbers:
[_ 1, 2, 3]
A tagged string:
24("foo")
A map with an indefinite map as a key:
{{_}: "value" }
```

Listing 2.1: Diagnostic notation showcase

2.1.1.2 Encoding the major type

The first byte of a data item (the major type byte) consists of 3 low-order bits of information specifying the major type according to their value. The remaining 5 high-order bits (called **additional information**) are used to either store the item's value, or to specify the magnitude of size of the item.

Decimal	Hexadecimal	Binary	First 3 low-order bits
10	0xa	0b1010	0b101
255	0xff	0b1111	0b111
962	0x3c2	0b11_1100_0010	0b111

Table 2.1: Illustration of the numerical notation

To illustrate this concept, let us consider the data item with major type byte `0x9e`. This is equivalent to `0b1001_1110`. The 3 low-order bits are `100`, therefore the major type of this item is `4`. The remaining five bits carry an additional information value of `0x1e`. We might also write the aforementioned byte as

`0b100_11110` in order to more clearly separate the major type and the additional information.

An overview of all the major data types along with examples follows. Note that their numbers directly correspond with their respective major type headers.

2.1.1.3 Major type 0 - unsigned integers

Items of this type are used for unsigned integers in the range from 0 to $2^{64} - 1$. Depending on the actual value, the 5-bit additional information has different meanings as defined in the following table.

Value range	Additional information	Comments
$[0, 23]$	`0x00` .. `0x17`	Corresponding to the value
$[24, 2^8 - 1]$	`0x18`	`uint_8t` follows
$[2^8, 2^{16} - 1]$	`0x19`	`uint_16t` follows
$[2^{16}, 2^{32} - 1]$	`0x1a`	`uint_32t` follows
$[2^{32}, 2^{64} - 1]$	`0x1b`	`uint_64t` follows

Table 2.2: Additional information values for Major type 0

Small unsigned integers are encoded in the additional 5 bits, which allows very compact encoding of small numbers. This principle is used in the following data types as well.

Another important property that should not go unnoticed is the fact that the same logical value can be encoded in multiple ways. For instance, 42 can be correctly represented in four different ways: `0x18_2a`, `0x19_002a`, `0x1a_0000002a`, or `0x1b_00000000_0000002a`. The implications of this property are discussed in subsection 2.2.3.

Examples

```
02 # unsigned(2)
```

Listing 2.2: An embedded positive `uint_8t`

```
18 2a # unsigned(42)
```

Listing 2.3: A full-length positive `uint_8t`

```
19 0101 # unsigned(257)
```

Listing 2.4: A positive `uint_16t`

```
1a 00067932 # unsigned(424242)
```

Listing 2.5: A positive `uint_32t`

```
1b 0000007f4b73be72 # unsigned(546726723186)
```

Listing 2.6: A positive `uint_64t`

2.1.1.4 Major type 1 - negative integers

Items of this type are used for negative integers in the range from -2^{64} to -1, in much the same way as unsigned integers.

It is important to notice that these integers are strictly negative. Somewhat counter-intuitively, CBOR does not have a 'signed' integral type. As this model does not fit with the typical signed-unsigned approach, mapping these values to the types of the host language might be challenging.

One should also observe that since a major type byte of a negative integer lies within the $[1 << 5, 1 << 5 + 27]$ range (equivalent to `0x20..0x3b`), the major type bytes between `0x1c` and `0x1f` inclusive remain unused. This is intended to allow future extensions to the type [7, s. 5.1].

Value range	Additional information	Comments
$[-1, -24]$	`0x00 .. 0x17`	Corresponding to the value
$[-25, -2^8]$	`0x18`	`uint_8t` follows
$[-2^8, -2^{16}]$	`0x19`	`uint_16t` follows
$[-2^{16}, -2^{32}]$	`0x1a`	`uint_32t` follows
$[-2^{32}, -2^{64}]$	`0x1b`	`uint_64t` follows

Table 2.3: Additional information values for Major type 1

Examples

```
21 # negative(1)
```

Listing 2.7: An embedded negative `uint_8t`

```
38 29 # negative(41)
```

Listing 2.8: A full-length negative `uint_8t`

```
39 0100 # negative(256)
```

Listing 2.9: A negative `uint_16t`

```
3a 00067931 # negative(424241)
```

Listing 2.10: A negative `uint_32t`

```
3b 0000007f4b73be71 # negative(546726723185)
```

Listing 2.11: A negative `uint_64t`

2.1.1.5 Major type 2 - byte strings

Items of this type are used for storing sequences of zero or more bytes (octets). This feature enables clients to include arbitrary binary data in CBOR items. The data are not escaped or altered in any way.

There are two fundamental kinds of byte strings: **definite** and **indefinite**. Whereas definite byte strings have a fixed length specified in their headers, indefinite byte strings consist of zero or more definite byte strings – so called **chunks** – followed by a **break** (a `0xff` byte indicating end of an indefinite item).

Additional information is used in a manner similar to that of previous types, with an extra value to denote the indefinite variant.

Additional information	Meaning
`0x0` .. `0x17`	0 – 23 bytes (corresponding to the value) follow
`0x18`	`uint_8t` and correspondingly many bytes follow
`0x19`	`uint_16t` and correspondingly many bytes follow
`0x1a`	`uint_32t` and correspondingly many bytes follow
`0x1b`	`uint_64t` and correspondingly many bytes follow
`0x1f`	indefinite byte string start

Table 2.4: Additional information values for Major type 2

Examples

```
40  # bytes(0)
    # ""
```

Listing 2.12: An empty byte string

```
58 1b                                           # bytes(27)
   416e79206279746573202d206576656e206e756c6c733a2 [...]
```

Listing 2.13: A byte string with `uint_8t` length

```
59 0102                                         # bytes(258)
   a25bb77465531b202fb3c938e0f4f7a95ec880364cad492 [...]
```

Listing 2.14: A byte string with `uint_16t` length

```
5f                                              # bytes(*)
   40                                           # bytes(0)
                                                # ""
   58 21                                        # bytes(33)
      c8da6848fa6a2defe45bc1b6b0499571dbb6af7fc743 [...]
   58 48                                        # bytes(72)
      14bb167a22786723f48426daa9e235e171928b9508e9 [...]
   ff                                           # primitive(*)
```

Listing 2.15: An indefinite byte string with several chunks

2.1.1.6 Major type 3 - text strings

Items of this type are used for storing sequences of zero or more (well-formed) characters (or more precisely, code points) of UTF-8 [63] encoded text.

As with byte strings, the characters are never escaped. For example the newline character (`\n`, U+000A) is encoded as `0x0a`, not as `0x5c6e` (`'\n'`), nor as `0x5c7530303061` (`'\u000a'`) [7, s. 2.1].

Text strings use the additional information in exactly the same way as byte strings do.

It should be noted that the definition implies that splitting a multi-byte character between two chunks of an indefinite text string is illegal, as the chunk would not be a string of *characters*.

One should also notice that the specified length of a text string or its chunk is given in bytes, not characters.

Additional information	Meaning
`0x0 .. 0x17`	0 – 23 bytes (corresponding to the value) follow
`0x18`	`uint_8t` and correspondingly many bytes follow
`0x19`	`uint_16t` and correspondingly many bytes follow
`0x1a`	`uint_32t` and correspondingly many bytes follow
`0x1b`	`uint_64t` and correspondingly many bytes follow
`0x1f`	indefinite byte string start

Table 2.5: Additional information values for Major type 3

Examples

```
60  # text(0)
    # ""
```

Listing 2.16: An empty text string

```
6c                          # text(12)
   48656c6c6f20776f726c6421 # "Hello world!"
```

Listing 2.17: A text string with embedded `uint_8t` length

```
79 01cd                                  # text(461)
   48656c6c6f20776f726c64212041206c6f74206d6f72652 [...]
```

Listing 2.18: A text string with `uint_16t` length

```
7f                           # text(*)
   6c                        # text(12)
      48656c6c6f20776f726c6421 # "Hello world!"
   60                        # text(0)
                             # ""
   69                        # text(9)
      53747265616d696e67     # "Streaming"
   ff                        # primitive(*)
```

Listing 2.19: An indefinite text string with several chunks

2.1.1.7 Major type 4 - arrays

Items of this type are used for storing sequences of zero or more (well-formed) data items. Arrays' length is encoded in much the same way as that of byte and text strings.

Nesting multiple arrays arbitrarily is legal, regardless of whether they are definite or indefinite. This creates possibly complex semantics, which are analyzed in section 2.2.4.

Indefinite arrays are terminated by the break code, just like strings.

Additional information	Meaning
`0x0 .. 0x17`	0 – 23 items (corresponding to the value) follow
`0x18`	`uint_8t` and correspondingly many items follow
`0x19`	`uint_16t` and correspondingly many items follow
`0x1a`	`uint_32t` and correspondingly many items follow
`0x1b`	`uint_64t` and correspondingly many items follow
`0x1f`	indefinite array start

Table 2.6: Additional information values for Major type 4

Examples

```
80 # array(0)
```

Listing 2.20: An empty array

```
9f    # array(*)
   ff # primitive(*)
```

Listing 2.21: An empty indefinite array

```
84           # array(4)
   01        # unsigned(1)
   02        # unsigned(2)
   03        # unsigned(3)
   63        # text(3)
      666f6f # "foo"
```

Listing 2.22: An array with several members

```
# [1, [0, [_ [[]]]], 2]
83             # array(3)
   01          # unsigned(1)
   82          # array(2)
      00       # unsigned(0)
      9f       # array(*)
         81    # array(1)
            80 # array(0)
         ff    # primitive(*)
   02          # unsigned(2)
```

Listing 2.23: A deeply nested array

2.1.1.8 Major type 5 - maps

Items of this type are used for storing associative maps (hash maps, analogous to JSON objects [12, s. 4]), consisting of keys and values. They are represented by a series of zero or more pairs of items. Pairs have no explicit delimiter, nor do their members. Length of maps is encoded using the same technique as for arrays.

Maps can also be indefinite, with the *break* code at the end. What is more, there is no limitation pertaining the keys and values. For example, indefinite keys and indefinite values are legal. Implications of this design decision are discussed in subsection 2.2.4.

Additional information	Meaning
`0x0 .. 0x17`	0 – 23 pairs (corresponding to the value) follow
`0x18`	`uint_8t` and correspondingly many pairs follow
`0x19`	`uint_16t` and correspondingly many pairs follow
`0x1a`	`uint_32t` and correspondingly many pairs follow
`0x1b`	`uint_64t` and correspondingly many pairs follow
`0x1f`	indefinite map start

Table 2.7: Additional information values for Major type 5

Examples

```
a0 # map(0)
```

Listing 2.24: An empty map

```
bf    # map(*)
   ff # primitive(*)
```

Listing 2.25: An empty indefinite map

```
# {"foo": 42, "bar": []}
a2           # map(2)
   63        # text(3)
      666f6f # "foo"
   18 2a     # unsigned(42)
   63        # text(3)
      626172 # "bar"
   80        # array(0)
```

Listing 2.26: A simple map with text keys

```
# {{_"foo": [1,2]}: 42, {_}: []}
a2              # map(2)
   bf           # map(*)
      63        # text(3)
         666f6f # "foo"
      82        # array(2)
         01     # unsigned(1)
         02     # unsigned(2)
      ff        # primitive(*)
   18 2a        # unsigned(42)
   bf           # map(*)
      ff        # primitive(*)
   80           # array(0)
```

Listing 2.27: A map with indefinite map keys

2.1.1.9 Major type 6 - semantic tags

Items of this type are not data per se. Instead, they appear before other data items and specify additional semantics or type conversions. The meaning of a particular tag is specified by a central authority [11]. Depending on the nature of a tag, it can apply to one or more types of data items.

For example, tag 1 specifies that the value of the immediately following item should be interpreted a as an epoch time stamp [28, s. 4.15]; it can be applied to both positive and negative integers, as well as to floats.

Tag application is right-associative. Tags A and B followed by a data item C should be interpreted as `A(B(C))`. The only limit for tag usage is that defined by the tag itself.

Additional information	Meaning
0x0 .. 0x17	Tags 0 – 23
0x18	uint_8t specifying the value follows
0x19	uint_16t specifying the value follows
0x1a	uint_32t specifying the value follows
0x1b	uint_64t specifying the value follows

Table 2.8: Additional information values for Major type 6

Examples

```
# 0("1970-01-01T00:01Z")
c0                                          # tag(0)
   71                                       # text(17)
      313937302d30312d30315430303a30315a # "1970-0 [...]
```

Listing 2.28: A string tagged as a timestamp

```
# 24("8")
d8 18    # tag(24)
   61    # text(1)
      38 # "8"
```

Listing 2.29: A byte string tagged as a nested CBOR (with value 8)

```
# 129(54873(24("8")))
d8 81          # tag(129)
   d9 d659     # tag(54873)
      d8 18    # tag(24)
         61    # text(1)
            38 # "8"
```

Listing 2.30: Nested tags

```
# 546234566543(42)
db 0000007f2e1e078f # tag(546234566543)
   18 2a            # unsigned(42)
```

Listing 2.31: A uint64_t tag

2.1.1.10 Major type 7 - floating-point numbers and simple values

Finally, items of this type are used for storing floating-point numbers, as well as several **simple values** and the *break* code.

The encoding defined by the IEEE 754 [32] standard is used for the numbers, but only half-, single-, and double-precision floats are currently available. There are no limitations regarding float normalization or denormals [26].

Simple values refer to `true, false, null,` and `undefined value`. As the standard contains no information about mapping of these values to the types of common programming languages, these values – especially `undefined value` – require extra attention when used.

Additional information	Meaning
`0x0 .. 0x17`	Simple value 0 – 23 (corresponding)
`0x18`	`uint_8t` specifying a simple value follows
`0x19`	half-precision float (16) follows
`0x1a`	single-precision float (32b) follows
`0x1b`	double-precision float (64b) follows
`0x1f`	*break* code

Table 2.9: Additional information values for Major type 7

Examples

```
# true
f5 # primitive(21)
# false
f4 # primitive(20)
# null
f6 # primitive(22)
# undefined
f7 # primitive(23)
```

Listing 2.32: `true, false, null, undefined`

```
# 0.5
f9 3800 # primitive(14336)
```

Listing 2.33: A half-precision float

```
# 100000.0
fa 47c35000 # primitive(1203982336)
```

Listing 2.34: A single-precision float

```
# 3.14159265358979323846264338327
fb 400921fb54442d18 # primitive(4614256656552045848)
```

Listing 2.35: A double-precision float

```
# Infinity
f9 7c00 # primitive(31744)
# NaN
f9 7e00 # primitive(32256)
# -Infinity
f9 fc00 # primitive(64512)
```

Listing 2.36: `Infinity, NaN, -Infinity`

2.1.2 Functionality

In the strict sense, CBOR is a data serialization format, and therefore it does not provide any 'functionality'[2] as such. There are, however, some major aspects which could be considered 'protocol features', or perhaps even functionality by a high-level client. These are streaming, lazy decoding, and custom semantics.

They are one of the major differentiators that set CBOR apart from similar formats, and thus deserve a thorough discussion. These properties are contrasted with the similar concepts found in other formats introduced in section 1.2.

Another reason to pay close attention to these concepts is the fact that a significant part of complexity in generic encoders and decoders arise from them, making them a major factor in overall performance (see section 5.1).

2.1.2.1 Streaming

The presence of indefinitely-sized data items goes hand in hand with the concept of streaming. Arrays and strings are particularly well-suited for streaming operations, as they are often used to transport data of a stream nature, for example lists (lazily evaluated, as opposed to fixed-length arrays) and IO streams.

This idea is commonly found in other similar formats, the most well-known example being SAX [13] for XML. Similar approaches were adopted for streaming JSON data. The existing implementations are somewhat fragmented, with no single specification, whether format or conventional. Some implementors prefer

[2] As in 'the range of operations one can perform using the system'

extending the format itself [57], whereas others choose to only allow concatenating multiple top-level objects [61].

2.1.2.2 Lazy decoding

Unlike both text-based and binary delimiter-based formats, most common CBOR structures have fixed length encoded in their headers. This enables one kind of lazy decoding, where a part of input is skipped and may be decoded later, when actually needed.

To illustrate this idea, consider the nested array

```
[1, 2, "abc<many more characters>", 3]
```

When decoding, only the header of the text string needs to be decoded. If appropriate, the decoder may skip the string with a simple jump or seek, as the number of bytes the string will occupy is known beforehand.

The other kind of lazy decoding uses byte strings that contain CBOR data items. Given the nested array

```
[1, [2, 3]]
```

we can encode it either as is, resulting in or use a more sophisticated approach

```
82       # array(2)
   01    # unsigned(1)
   82    # array(2)
      02 # unsigned(2)
      03 # unsigned(3)
```

and encode

```
[2, 3]
```

fragment as a byte string, resulting in

```
82    # array(2)
   02 # unsigned(2)
   03 # unsigned(3)
```

We then apply the tag 24, which is used for encoded CBOR items, [7, s. 2.4.4.1] and include it in the top-level array, resulting in

```
82              # array(2)
   01           # unsigned(1)
   d8 18        # tag(24)
      43        # bytes(3)
         820203 # "\x82\x02\x03"
```

The decoder can then decide freely when to invest its computational resources into decoding the inner array.

2.1.2.3 Custom semantics

Owing to the tagging system and byte string data type, arbitrary data formats and conventions can be used and supported systematically in CBOR-based applications. This is a significant improvement over earlier formats, as it allows to clearly and consistently separate schema validation from data validation. Hopefully, it will usher in the creation of more robust applications protocols.

Furthermore, as new tags and conventions become generally accepted, they can be added to the official tag repository [11]. This should enable smooth formalization of new conventions without the need for updating the protocol or generic codecs.

2.2 Key properties of the CBOR format

2.2.1 The notions of validity and well-formedness

There is an important distinction to be made between **well-formed** and **valid** CBOR data items. The standard defines a *well-formed* data item as

> A data item that follows the syntactic structure of CBOR. A well-formed data item uses the initial bytes and the byte strings and/or data items that are implied by their values as defined in CBOR and is not followed by extraneous data. [7, s. 1.2]

In order for a data item to be consider *valid*, it has to be well-formed and "follow the semantic restrictions that apply to CBOR data items." [7, s. 1.2]

One might think of these concepts as being equivalent to syntax and semantics. To illustrate the difference between these two concepts, consider a hash map with duplicate keys. Such an item is *well-formed*, but not *valid* [7, s. 2.1].

```
# {"key": "first value", "key": "second_value"}
a1                              # map(1)
   63                           # text(3)
      6b6579                    # "key"
   6c                           # text(12)
      7365636f6e645f76616c7565 # "second_value"
```

Listing 2.37: A well-formed, invalid map

This distinction fits in well with the model of a generic decoder that only concerns itself with syntax. In streaming application, for instance, not only is it impractical for the decoder to verify validity, it may in fact be outright impossible for long enough streams.

Invalid Unicode encoding of a text string is another real-world example that is worth mentioning. It presents us with an interesting corner-case, as the distinction between well-formedness and validity is not clearly defined by the standard in this case. The standard states that a text string data item consists of UTF-8 encoded text, which implies that only valid UTF-8 strings are valid CBOR data items [7, s. 2.1]. This in turn obliges the decoder to verify the semantics of UTF-8 encoding. We can clearly see that this is not consistent with the 'syntax only' understanding of well-formedness, and what is more, it contradicts the instructions for error handling [7, s. 3.4].

```
64          # text(4)
   74007374 # "t?st"
```

Listing 2.38: An invalid UTF-8 string

Even though this behavior pushes more complexity into the decoder and comes at a performance cost, we put forward the view that it is necessary to verify text string encoding in the decoder. The main reason for this claim is the fact that many systems today still rely on NULL-terminated strings. The possibility of introducing unexpected[3] NULL bytes, especially from external sources, is a well-known source of potential vulnerabilities [17][47].

[3]Valid UTF-8 encoded string do not contain NULL bytes.

2.2.2 Nested data items

CBOR arrays (2.1.1.7), hash maps (2.1.1.8), and tags (2.1.1.9) allow nesting of arbitrary depth. Both streaming and standard codecs will most likely utilize a stack mechanism to keep track of the data structure. This, together with the fact that we cannot easily allocate memory beforehand, means that the implementation will need to take precautions to prevent resource exhaustion attacks by means of excessively nested structures.

Another aspect to consider is memory management and correctness. This is outstandingly important in C, as the language provides no abstractions to work with. Given the object-like structure of CBOR, a graph of 'fake objects' seems to be a natural fit. In order to allow for correctly handling operations such as 'append a map to an array', a memory ownership and lifetime model will be needed.

2.2.3 Hash map keys

Unlike all the other protocols that support hash maps, CBOR allows using arbitrary data items as map keys [7, s. 3.7].

Another somewhat underspecified aspect of the format is the equality relation between different representations of two (possibly) semantically equivalent data items. For example, the same array can be represented as either fixed-length or indefinite item:

```
# [1, 2, 3]
83    # array(3)
   01 # unsigned(1)
   02 # unsigned(2)
   03 # unsigned(3)
# [_ 1, 2, 3]
9f    # array(*)
   01 # unsigned(1)
   02 # unsigned(2)
   03 # unsigned(3)
   ff # primitive(*)
```

Listing 2.39: Definite and indefinite versions of the same array

Given these settings, we need to ask ourselves how to define equality of hash

map keys. This is a rather convoluted issue, as the equality relation may be application specific. Moreover, it is very difficult to provide efficient associative data structures for keys that lack canonical representation. An even more detailed discussion of this issue can be found in subsection 3.3.2.

The *cbor-ruby* gem discussed in subsection 2.3.3 is a real-world example of a decoder that hides and disregards whether decoded objects are definite or indefinite.

As most real-world applications only use a restricted set of fairly uniform keys, we came to the conclusion that a generic codec should not extend its scope to include optimized associative data storage structures.

2.2.4 Indefinite data items

When dealing with items whose length is not known beforehand, we must carefully consider how to handle memory allocation. Although effective memory allocation strategies for buffers of unknown length exist (namely the technique of exponential growth and shrinking), they are not looked upon favorably in network-facing code. What is more, relatively frequent reallocations might not only be costly, but also amplify fragmentation issues, as shown by Chang et al. [14, pp. 3].

One might consider either operating in a fixed, pre-allocated memory pool, or use a more elaborate memory management strategy, perhaps one that can incorporate application-specific hints to its decision strategy (see also Jula et al. [37]).

Unfortunately, both of the possible approaches require careful planning and are very much dependent on the knowledge of the specific application and operational properties. For this reason, we propose using the classical approach. When implemented carefully, it will provide an acceptable performance without impeding the generality of the implementation. The option to use a custom allocator might be a good way to allow for fine-tuning without introducing high incidental complexity.

2.3 Existing implementations

As mentioned in section 1.3, there is no general-purpose C implementation available at the time of writing. There are, however, some simplified or niche-specific implementations already available. As for Ruby libraries, a fairly mature fork of the official MessagePack implementation adapted to work with CBOR has been published by C. Bormann [9].

2.3.1 RIOT OS

An experimental C implementation can be found in the RIOT operating system [51] (a lightweight OS for embedded computing). While it does support encoding and decoding all the major types, it cannot be meaningfully integrated into any application as it doesn't provide any data model – all the decoded data are just pretty-printed to the standard output.

Streaming support is completely omitted as well. Although the code appears to be functional, it really is much more of a proof of concept rather than a full-fledged implementation.

2.3.2 cn-cbor

cn-cbor is a very lightweight C implementation for constrained nodes [10]. The author claims the compiled code size to be smaller than 1 KB.

The API covers most of the standard, excluding some specific functionality like UTF-8 checking, and works with indefinite items, although not in a streaming manner. The code itself is rather terse, as one would expect.

Memory management employs a simple hierarchical structure: container items are responsible for deallocation of the items nested inside them. One should notice that this is a simple instance of our approach described in subsection 3.3.3.

2.3.3 cbor-ruby

cbor-ruby [9] is a fairly mature and complete Ruby implementation that is based on msgpack-ruby [24]. It claims to be 'high-performance', although no bench-

marks are available. This claim is based on the fact that the core is implement in C rather than Ruby. On the one hand, the C backend is likely to boost performance, but on the other hand, it renders the library incompatible with platforms other than MRI and Rubinius, as it uses features specific to MRI[4].

A major problem with *cbor-ruby* lies in the outdated API and its documentation – only basic features have been updated to use CBOR, other more advanced features (such as streaming and evented socket handlers) remain dysfunctional.

The code, including the memory model, is tied to the internals of MRI. Native memory management is interwoven with Ruby's garbage collection. Memory management is, therefore, performed through the Ruby runtime, which handles the actual resources. The library only creates the appropriate GC marks.

2.4 Other related work

2.4.1 Noteworthy implementations of other formats

2.4.1.1 YAJL

YAJL [30] is a JSON parser with an interesting approach to memory representation: instead of building new data structures in the memory, YAJL is a purely streaming parser. It only emits events, meaning it almost never allocates new objects. Although it is not necessarily faster than traditional approaches [4], this approach has a significant potential due to the fact that memory operations are often the limiting factor in both encoding and decoding applications.

2.4.1.2 Jansson

Jansson [42] is a JSON library that is on the other end of the spectrum. Offering a rich API and a complete memory representation and manipulation model for documents, it is often touted as the easiest JSON library for C to use. Another interesting property is its reference counting memory management. While it does suffer from the problems commonly associated with reference counting, such as the inability to detect circular references, it can be beneficial in two ways.

[4]Rubinius maintains MRI API compatibility

Firstly, it enables *structural sharing* of fragments. One can easily imagine how this can be useful for reusing common pieces of data, e.g. headers, cached objects, or metadata.

Secondly, it can be used to effortlessly establish a simple ownership model. Functions can either increase, decrease, or not modify the reference counter. Borrowing and ownership of input and output arguments is then easily determined from functions interaction with the reference counter.

The approach used in the implementation presented in this thesis builds upon this idea. A similar, albeit more sophisticated, model is introduced in section 3.3.3.

2.4.2 Generated parsers

A fairly popular approach to building implementations of new protocols and data formats is to generate the actual decoder from its specification using a combination of a lexer generator and a parser generator.

This approach has several advantages, most importantly it minimizes the effort required, ensures the product will comply to the specification, and usually delivers optimized parsers.

Unfortunately, there are no known tools that could generate lexers operating at a granularity lower than one byte or octet. Boost.Spirit [35] probably comes the closest, but it still doesn't offer a way to handle constructs such as major type byte embedded values.

2.5 Summary

Several CBOR features that will have a cogent impact on any generic implementation have been discussed. Most problems seem to center around memory management and data manipulation.

Exploring the landscape, we have investigated the approaches, advantages, and shortcomings of three existing CBOR implementations, as well as two innovative JSON implementations. They are utilizing techniques such purely streaming decoding and memory reference counting that might be useful for implementing

CBOR as well.

Finally, the possibility of using a parser generator or a similar technology has been considered. Due to the tight binary coding of CBOR, such an approach is not viable.

Chapter 3

Implementation

In this chapter, our C implementation of the CBOR format, *libcbor*, and its Ruby derivative, *libcbor-rb*, are introduced. We present and justify the design decisions through which the goals laid out in section 1.4 have been achieved.

The tools, methods, and design paradigms that were utilized during the development of *libcbor* are also briefly mentioned.

Finally, we conclude the chapter with an overview of reliability and correctness features and processes that were employed.

3.1 Design overview

As already stated in the introduction, the implementation consists of two main parts:

- *libcbor*, the C library,
- *libcbor-rb*, the Ruby binding and extension for *libcbor*.

libcbor is completely oblivious of *libcbor-rb* in terms of API design. This is intended to stipulate that a clear design will not be cluttered by any Ruby specifics. Furthermore, it increases the potential of re-using *libcbor* in other contexts as well. One can easily imagine use cases such as bindings similar to *libcbor-rb* for other languages, incorporation into specific applications, or perhaps even building custom implementations of CBOR-based protocols.

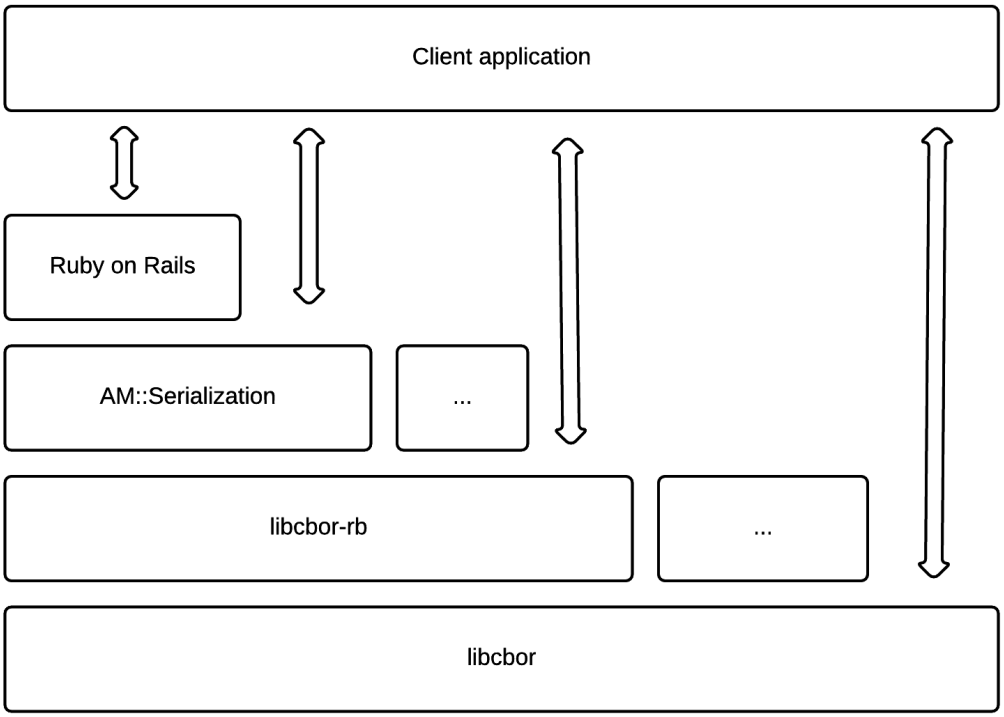


Figure 3.1: Possible layers of abstraction on which the client might interact with CBOR. In this particular example, we show how *ActiveModel::Serializers* [1], a library that abstracts different serialization formats, could possibly integrate *libcbor-rb*. Since the *Ruby on Rails* [53] web framework relies on *ActiveModel::Serializers*, the functionality will be propagated higher up the abstraction hierarchy.

The aforementioned approach has many advantages. Structuring abstraction in such a way is generally considered to be a 'best practice' [40, pp. 146]; besides, there are three more tangible benefits to it for our implementation.

Firstly, the client can choose the **optimal control and performance to effort ratio** according to his own needs. For example, a web developer wishing to use CBOR in a WebSockets [29] application will be happy to use the highly abstract interface. A message queue engineer, on the other hand, might want to take the opportunity and configure *libcbor* with the parsing strategy and data structures that will provide the required performance.

Secondly, it enables **faster iteration and release cycle**. While it is common to update your dependencies (i.e. *libcbor-rb*) on a monthly basis in the Ruby ecosystem, system libraries (i.e. *libcbor*) get updated in much longer cycles. Decoupling the components will allow users to take advantage of new updates

in *libcbor-rb* without having to update *libcbor*.

Finally, having possibly many instances of CBOR-related software that all rely on a single low-level library is the best-case scenario in terms of integration and compatibility issues. It is quite common that different implementations of standardized protocols slowly deviate from each other, ushering in incompatibilities. The presence of one implementation that **sets the standard** is likely to mitigate most of the problems.

Overall, the design philosophy follows five simple paradigms (roughly in order of importance):

- Adhere to the standard
- Focus on robustness and security
- Prefer simplicity over unnecessary abstraction
- Handle failure predictably and gracefully
- Strive for high performance

Notwithstanding the effort that was put forward to achieve the goals while following these principles, there were situations where we found some of them in conflict. A detailed discussion of the features that were shown to be somewhat problematic can be found in section 6.1.

3.2 Technology and supported platforms

Broadly speaking, there are only two technologies that *libcbor* and *libcbor-rb* depend on by intentional choice. These are the C99 revision [34] of the C language, and the FFI gem[1][52]. The reasoning behind these choices is presented below. For tools and other supporting software, please refer to section 3.5.

3.2.1 C99

We have chosen C99 instead of the older revisions because CBOR is an emerging technology, therefore it is rather unlikely that users of legacy systems will seek

[1] A packaged Ruby library. See the glossary.

to utilize it. Having said that, we also take advantage of several new features that cannot be found or emulated in the older version, e.g. the `restrict` specifier. C11, on the other hand, is still lacking support on many platforms and while it does have several useful features (`_Generic` macros would be especially convenient), these features do not justify the loss of compatibility.

The reason for not choosing C++, whose OO nature could simplify modeling the object-like nature of CBOR items, is the fact that C is much more widely used for FFI embedding, predominantly due to its much simpler calling convention.

We could provide a C interface for a C++ library, but we maintain that this would be detrimental to the design quality across all the components. Due to C++'s exception mechanism, overloading, and constructor mechanisms, wrapping idiomatic C++ API into a C one is a daunting task. To worsen the situation even further, C++ generics would be of no value as we would be forced to list and implement all template instantiations explicitly. Finally, C is, compared to C++, more commonly found on embedded platforms, which is yet another reason to prefer plain C, as embedded devices might be a good niche for CBOR.

3.2.2 Ruby FFI

Ruby FFI is a useful tool that allows developers to create FFI integrations for Ruby that will be independent of the concrete Ruby implementation. This is an extremely valuable asset that even many popular Ruby gems sorely lack.

To illustrate this, consider JRuby [46], the popular JVM-based Ruby implementation. In order to attach a native extension, one would typically have to do so through the JNI. The C code would then have to be specific to the JNI. Ruby FFI provides a unified interface that abstracts this capability across virtually all Ruby implementations.

3.3 *libcbor*

The C part is fairly simple in terms of conceptual complexity, as one would expect. The incidental complexity, however, manifests itself in three major aspects.

First of these aspects is memory management. Dealing with nested data

items with dynamic size requires a substantial amount of attention by both the implementors and the client. To alleviate this woe, a reference-counting ownership schema has been implemented (see section 3.3.3).

The second problem arises from the need to robustly handle failure. Since C lacks exceptions or a similar error-signaling mechanism, we had to resort to the typical approach employing either return values or output parameters.

Finally, CBOR structures map poorly to the procedural constructs of C. We use a combination of two well known techniques: opaque structures with functions emulating dynamic dispatch where necessary, and multi-way branching based on enumerations elsewhere.

3.3.1 Design overview

The library has three main layers, which are approximately divided by their level of abstraction.

3.3.1.1 Internal – encoders and loaders

This layer consists of functions located in the `src/cbor/internal` directory and provides primitive data operations. These include reading and writing atomic data blocks as well as endianess transformations. This API layer is not intended to be used by the clients. Header files that belong to this category are

- `src/cbor/internal/loaders.h` – provides function such as `_cbor_load_uint16` that read buffers and perform endianess conversion. They also handle IEEE 754 half precision floats [32, s. 4.3] decoding, as C has no standard facility to work with them.

- `src/cbor/internal/encoders.c` – provides functions such as `_cbor_encode_uint8` that write into buffers in a checked manner and also perform endianess conversion. Many of them use compiler specific built-ins in order to achieve higher performance, as shown in listing 3.1.

- `src/cbor/internal/unicode.h` – provides a fast UTF-8 validation routine and a code point counting routine.

```
#ifdef HAVE_ENDIAN_H
        *(uint16_t *) &buffer[1] = htobe16(value);
#else
        #ifdef IS_BIG_ENDIAN
                *(uint16_t *) &buffer[1] = value;
        #else
                buffer[1] = value >> 8;
                buffer[2] = value;
        #endif
#endif
```

Listing 3.1: Built-in endianess conversion routines (`src/cbor/internal/encoders.c`).

3.3.1.2 Streaming API

The streaming API provides the ability to incrementally decode or encode data in an event-driven manner.

When decoding, *libcbor* provides a stateless decoder that invokes a set of callback functions as it progresses. These callbacks are free to handle the input in any way they deem suitable, including ignoring it.

Since the parsing process does not involve memory allocation or complex manipulation, this API level is a perfect fit for creating application specific high-performance decoders that work with the client's data structures.

The encoding API provides functions that operate upon buffers, encoding simple values with the appropriate CBOR major type byte. Just like the decoding API, they do not have any notion of state or non-local syntax; the client takes the responsibility for ensuring the well-formedness of the resulting CBOR.

The streaming API is defined in the following header files:

- `src/cbor/encoding.h` – provides the aforementioned encoding functions, such as `cbor_encode_string_start`
- `src/cbor/callbacks.h` – defines the data types for decoder interaction. Listing 3.2 exemplifies the structure of the 'callback bundle'.
- `src/cbor/streaming.h` – contains the streaming decoder itself

```
17 /** Callback prototype */
18 typedef void(*cbor_int8_callback)(void *, uint8_t);
19
20 /** Callback prototype */
21 typedef void(*cbor_int16_callback)(void *, uint16_t);
```

```
48 struct cbor_callbacks {
49         /** Unsigned int */
50         cbor_int8_callback uint8;
51         /** Unsigned int */
52         cbor_int16_callback uint16;
```

Listing 3.2: Two parts of the callback passing structure (`src/cbor/callbacks.h`).

3.3.1.3 Data items API

This layer provides the highest level of abstraction and convenience, at the cost of performance and flexibility. It encodes and decodes whole items that are stored in the memory and provides some 200 routines to manipulate them.

The data items are reference-counted to alleviate the burden of manual memory management. The ownership model is described in more detail in subsection 3.3.3.

The key headers for this layer are:

- `src/cbor.h` – top level header, includes all the APIs
- `src/cbor/serialization.h` – data items serialization
- `src/cbor/common.h` – utilities, custom allocators support, and reference counting manipulation
- Type specific routines in
  - `src/cbor/ints.h`
  - `src/cbor/bytestrings.h`
  - `src/cbor/strings.h`
  - `src/cbor/arrays.h`
  - `src/cbor/maps.h`
  - `src/cbor/tags.h`

– `src/cbor/floats_ctrls.h`

When decoding complete data items, *libcbor* performs syntactical and semantic verification to ensure that the incoming data are well-formed and valid. Parsing is implemented using a simple stack-based mechanism using the algorithm described in listing 3.3.

3.3.2 Data structures

The lower two layers of the API do not use any noteworthy data structures by conscious decision – they were designed to be as simple and transparent as reasonably possible.

The data items API is centered around `cbor_item_t` (shown in listing 3.4). This structure mimics the OO nature of CBOR data items, effectively acting as a discriminated union with the discriminator `type`.

The `metadata` field is composed of metadata structures specific to each respective major type. From this it follows that the interpretation of `data` depends upon the `type` and the `metadata`.

We were faced with a considerable challenge when choosing the data structures for the two non-trivial nested types: maps and arrays. As to arrays, the primary premise is that clients will expect them to work as arrays, i.e. contiguous blocks of memory. This leaves us with no other viable choice than to use an array-like structure.

Our implementation attempts to maximize the performance by

- using a *libstc++*'s [41] `std::vector`-like preallocation strategy parametrized with an optimal value (configurable during build) based on experimental measurements,
- growing the array exponentially (likewise, the growth factor is configurable), and
- giving the client the ability to provide his own allocator.

In spite of the acknowledged lack of sophistication, the measurements presented in section 5.1 show that this approach, largely by virtue of careful implemen-

```
Data: List of primitive CBOR tokens L
Result: A well-formed CBOR item
initStack(S);
if empty(L) then
  | fail('No data')
while notEmpty(L) do
  | switch takeHead(L) do
  |   | case ARRAY, INDEFARRAY
  |   |   | push(S, new_array)
  |   | end
  |   | case MAP, INDEFMAP
  |   |   | push(S, new_map)
  |   | end
  |   | case INDEFSTRING
  |   |   | push(S, new_string)
  |   | end
  |   | case INDEFBSTRING
  |   |   | push(S, new_bstring)
  |   | end
  |   | case TAG
  |   |   | push(S, new_tag_head)
  |   | end
  |   | case BREAK
  |   |   | if indefinite_item(top(S)) then
  |   |   |   | append(pop(S))
  |   |   | else
  |   |   |   | fail('Malformed item')
  |   |   | end
  |   | end
  |   | otherwise
  |   |   | append() the item
  |   | end
  | endsw
end
if size(S) > 1 then
  | fail('Not enough data')
else
  | return pop(S)
end
```

Listing 3.3: CBOR parsing algorithm. The append routine is defined in listing 3.7

tation, can outperform seemingly more advanced ones. Moreover, the ability to use the array with standard library functions such as `qsort` is a significant factor in terms of practicality (illustrated in `examples/sort.c`).

In the case of maps, the issue is even more convoluted. Not only are the keys

```
135 };
136
137 /** Union of metadata across all possible types - discriminated in #cbor_item
138 union cbor_item_metadata {
139         struct _cbor_int_metadata        int_metadata;
140         struct _cbor_bytestring_metadata bytestring_metadata;
141         struct _cbor_string_metadata     string_metadata;
142         struct _cbor_array_metadata      array_metadata;
143         struct _cbor_map_metadata        map_metadata;
144         struct _cbor_tag_metadata        tag_metadata;
145         struct _cbor_float_ctrl_metadata float_ctrl_metadata;
```

Listing 3.4: The generic item handle (`src/cbor/data.h`).

potentially ambiguous, as described in subsection 2.2.3, but they may also be extremely long.

While one could allow the client to define application specific key comparators that could then be used in tree-based structures, a crucial thing to remember here is the aforementioned unbounded length of keys for general-case input. Furthermore, any reasonable comparator would most likely have linear complexity in term of the item's serialized length. Consequently, any deterministic comparison-based structure would have a propensity for pathological behavior for certain inputs (i.e. traversal or lookup paths). This is a major security concern that could potentially result in both denial-of-service and timing vulnerabilities.

Tries may, in theory, serve our purpose. The fact that we would have to construct them byte-wise, however, renders them unusable. Memory overhead of one pointer per byte is unacceptable by all standards, and we would have to keep a working copy of the key regardless to keep e.g. arrays contiguous. Even if implemented as a compressed trie, dealing with updates would still be prohibitively expensive.

Finally, hashing is generally unsuitable due to its memory overhead and the susceptibility to complexity-based attacks through pathological inputs, as demonstrated by Crosby and Wallach [18].

Referring to the goals defined in section 1.4, we have decided to use a simple array instead of introducing the complexity that would inevitably come with a more elaborate solution. Based on the discussion above, there is no known

elegant and compact solution. Therefore, given these circumstances, simplicity and separation of concerns take precedence.

In most applications, keys will be fairly simple, perhaps just simple text strings. Users can implement their own data structure using the streaming API with relatively little effort.

3.3.3 Ownership and reference counting

In order to correctly handle memory management in complex nested structures, *libcbor* implements a reference counting scheme that ensures correct allocation and deallocation, while also enabling structural sharing.

Every data item has a **reference count** stored in its handle (`cbor_item_t`). This value corresponds to the total number of references to the item held across the system. When the reference count reaches zero, the item is no longer accessible and should be deallocated.

Those parts of the system that hold a reference to a value, be it in a variable or an intermediate value, are said to **own** it. When an entity owns an item, it can do so in either **exclusive** or **shared** manner.

An entity has an exclusive ownership of an item if and only if it holds one or more references to the item and no other part of the system holds a reference to the same item. In other words, no other part of the system can know that this item exists. More specifically, when the reference count is one, any ownership relation is an exclusive ownership relations.

Conversely, shared ownership is the state when two or more different entities hold a reference to the same item.

We define the reference graph to be **consistent** at the given instant if and only if the reference count of all items corresponds to the actual number of references to each and every item in existence. Note that this does not imply that the respective owners are somehow 'aware' of their ownership. We could use a stronger definition requiring that every reference owner is aware of his ownership. The weaker requirement is, however, sufficient for ensuring correct memory management. If the reference graph is not consistent, we consider it to be **inconsistent**.

libcbor functions can either **create** new references, **borrow** them, or **take ownership** from the caller. Every function in *libcbor* is annotated with respect to their reference behavior.

Functions returning new reference can be thought of as constructors in OO environments. They take no data item on input, and return items with `refcount = 1`. This reference belongs to the caller and is exclusive. The pointers created by these functions are guaranteed to follow strict aliasing rules (as defined by the C99 `restrict` specifier [54, s. 6.7.3.1]) over their valid lifetime.

Functions that borrow a reference take temporary ownership of the item during their invocation. These are often functions that manipulate or transform items. Caller's reference ownership is transferred to them upon invocation and they return it to the caller once they yield control. They do not change the reference count.

Functions that take ownership become owners of the caller's reference when invoked. The caller gives up on the reference and acknowledges that it can no longer operate upon the item unless it owns another, different reference. The most notable example is `cbor_decref`, which should be called when exiting the scope, as illustrated in listing 3.5.

A reference owner may also either borrow or transfer his ownership to a thread or a similar execution unit. Furthermore, some function increase the reference count of some of its arguments – this is essentially a combination of creating a new reference and then passing it to a function that takes ownership.

```
1 FILE * f = fopen(argv[1], "rb");
2 if (f == NULL)
3   usage();
4 fseek(f, 0, SEEK_END);
5 size_t length = (size_t)ftell(f);
6 fseek(f, 0, SEEK_SET);
7 unsigned char * buffer = malloc(length);
```

Listing 3.5: Ownership and lifetime illustration. `cbor_load` returns a new reference, therefore releasing the reference with `cbor_decref` is necessary (`examples/readfile.c`).

This model is not only useful for correct memory management, but it also gives us several straightforward properties for concurrent manipulation with data items. First and foremost, exclusive ownership guarantees exclusive access, and the owner is therefore free to use the data item. When items and references are shared across threads or other concurrent execution units and the owner cannot prove his exclusive ownership, the client has to take the responsibility for access synchronization in his application.

In order to allow for the following two guarantees, library functions are extensively annotated with the `const` specifier and mutation descriptions to distinguish **read-only** routines from **read-write** ones, depending on whether they refrain from modifying their arguments or not.

Serial correctness guarantee If the reference graph was consistent throughout the entire execution in a non-concurrent environment, *libcbor* guarantees that all the memory reads and writes issued by its functions will be correct and all the items allocated through its constructors will be fully and correctly deallocated.

Concurrent correctness guarantee If the reference graph was consistent throughout the entire execution in a concurrent environment and all read-write functions were invoked upon items only by their respective exclusive owner, *libcbor* guarantees that all the memory reads and writes issued by its functions will be correct and all the items allocated through its constructors will be fully and correctly deallocated.

Note that the aforementioned guarantees are implications. It is possible for a program to enter an inconsistent state and still maintain correctness. This is usually achieved through additional semantics of the client application.

3.4 *libcbor-rb*

The purpose of *libcbor-rb*, a Ruby bindings for *libcbor*, is to be as simple to use as possible while still allowing clients to use all the features of CBOR. In order to achieve this goal, simply providing a way to call the *libcbor* routines from Ruby is not sufficient – the resulting API would not be convenient by any

means. Therefore, *libcbor-rb* is not merely a binding, it is more of a combination of binding, wrapper, and a collection of utilities.

3.4.1 Design overview

Much like *libcbor*, *libcbor-rb* uses a layered architecture in order to present the clients with a flexible mix of control and convenience. The layers, however, are not build on top of each other. Instead, the two high-level layers both directly use the *libcbor* binding. Figure 3.2 illustrates this idea and provides a comparison to *libcbor*.

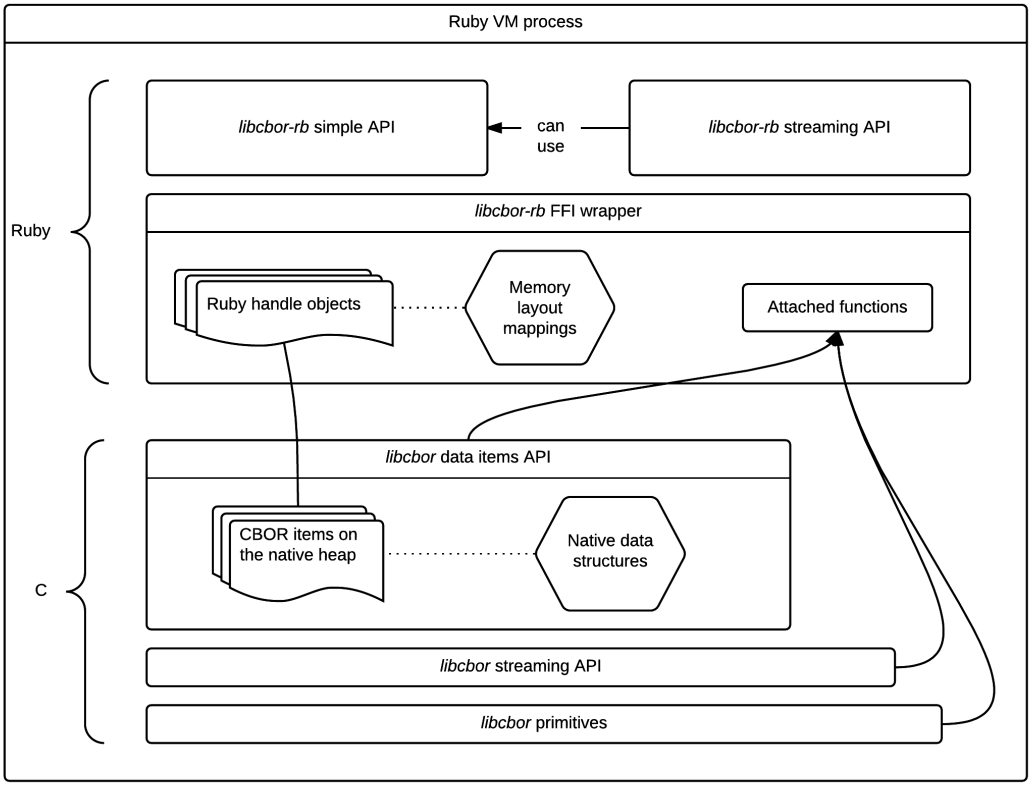


Figure 3.2: Illustration of the relation between *libcbor* and *libcbor-rb* in a Ruby process.

3.4.1.1 Native wrapper

At the heart of *libcbor-rb* lies the lowest-level wrapper to the native library. It provides very little to no abstraction; in simple terms, it allows calling C routines from Ruby and vice versa. The complexity that is being dealt with in

this component is the complexity of managing such interactions.

This process involves creating a platform-independent description of the *libcbor* API and the memory layout of its data structures. Upon execution, this blueprint is used to create a concrete ABI-like layer through which the shared library is attached to the Ruby interpreter.

```ruby
def __libcbor_to_cbor
  @@item ||= LibCBOR.cbor_new_definite_string
  string = FFI::MemoryPointer.from_string(self)
  out_bfr = FFI::MemoryPointer.new :pointer
  out_bfr_len = FFI::MemoryPointer.new :size_t
  LibCBOR.cbor_string_set_handle(@@item, string, bytes.length)
  res_len = LibCBOR.cbor_serialize_alloc(@@item, out_bfr, out_bfr_len)
  out_bfr.read_pointer.get_bytes(0, res_len).tap do
    LibC.free(out_bfr.read_pointer)
  end
end
```

Listing 3.6: Converting a Ruby string into its CBOR representation (`lib/libcbor/helpers.rb`).

The second important aspect of interfacing with C code is memory management. The FFI mechanism provides facilities to allocate both heap and stack memory directly from Ruby and manage it through a handle object. It should be noted that this memory is off limits for the Ruby runtime and whereas the handle object will be collected by the GC, the lifetime process of such buffers must be managed manually. Listing 3.6 illustrates the process of invoking a *libcbor* function from Ruby.

Files belonging to this layer are found in the `lib/libcbor/inner` directory. Apart from *libcbor-rb* files, the directory also contains the `lib/libcbor/inner/lib_c.rb` utility module, which wraps C memory allocation functions for use in Ruby.

3.4.1.2 Streaming API

This layer provides a simple API for incrementally encoding and decoding CBOR, namely the `CBOR::Streaming::BufferedDecoder` and `CBOR::Streaming::Encoder` classes, along with several utility classes.

Regardless of its simplicity, it is very easy to use, as shown in Appendix C.

3.4.1.3 Simple API

Unlike C, the standard library of Ruby provides suitable data structures for all the data types, with the exception of tags. *libcbor-rb* therefore relies on the built-in data structures for representation. Conversion between the *libcbor* representation and the Ruby object graph is provided by the `CBOR::CBORItem` class.

Encoding is simple as well; users can even define their own mappings for user-defined classes. We achieve this by dynamically extending the core classes with a suitable collection of modules (horizontal re-use units of Ruby). Due to the fact that this extension is performed after *libcbor-rb* has been loaded for the first time, it can be customized so as to prevent name clashes and give the user more flexibility.

3.4.2 Memory management

Where possible, *libcbor-rb* relies on handle objects representing native memory objects. When the handle object is deallocated by the garbage collector, it either frees the memory (in case it represents a buffer), or decreases the reference count in the *libcbor* reference counting mechanism. This approach is in line with the model described in subsection 3.3.3.

3.5 Tools, processes, and quality assurance

We have gone to great lengths to ensure that *libcbor* will be as practical and usable as is realistically possible. This includes adhering to standard processes and providing easy to use utilities for installation, management, and documentation. The importance of reliability and robustness has already been stressed; this section also describes the approach that has been taken to fulfill this goal.

3.5.1 Building, packaging, and installation

libcbor uses CMake, a widely used free build software [43] that can generate build scripts for common back ends such as GNU `make`, Microsoft's Visual C tool chain, or OS X's XCode tools. This, paired with the standard-compliant

C99 implementation, ensures effortless portability across all major platforms, including ARM-based devices.

Users can either build the library themselves from the source (as described in the documentation), or use pre-built `rpm`, `deb`, and plain `tgz` packages that are provided with each release.

As to *libcbor-rb*, it is distributed through the central gem repository. The package manager will resolve it together with its dependencies automatically. Alternatively, one can build the gem from source locally. Both of these approaches are documented in the user manual.

3.5.2 Versioning

Both *libcbor* and *libcbor-rb* follow the semantic versioning convention [49]. The source code is managed using *git*, with a stable master branch and version tags. This approach will ensure smooth cooperation with dependency managers and effortless updates.

3.5.3 Correctness

Both *libcbor* and *libcbor-rb* are covered by extensive test suites. The *git* repository is integrated with a CI service that runs all the checks automatically for every commit across many possible configuration and build setups, ensuring that broken versions are fixed in time. This checking also includes code coverage analysis, which forms an important part of the test suite's credibility.

In the case of *libcbor*, the test suite is roughly divided by the API functionality levels and comprises both unit and integration tests based on the CMocka framework [56]. All parts of the library are tested by more than 1100 manually written assertions with coverage nearing 99 %.

libcbor-rb is tested using an executable specification that provides not only behavior verification, but a human readable description of the desired contract as well. The test coverage is 100 %.

A fuzz test where many random sequences of bytes are repeatedly passed to the decoder is also a part of the test suite. We require that the decoder

either succeeds or fails with a meaningful error while leaking no memory. This measure is crucial to ensuring that both *libcbor* and *libcbor-rb* can be safely used in network-facing code.

Yet another measure to provide the best possible quality is the use of static analysis tools during the CI process, *cppcheck* in particular, to prevent common mistakes and vulnerabilities.

3.5.4 Memory correctness verification

One of the most common pitfalls of developing in the C language is incorrect memory management, ultimately resulting in crashes, undefined behavior, and memory leaks. *libcbor* benefits greatly from its use of automated memory correctness checking. We use Valgrind [45], which, through dynamic instrumentation of memory allocation and access, can detect most of the common mistakes.

During the CI process, the whole test suite is run with Valgrind instrumentation and the results are reported. This has enabled us to identify tens of potential memory errors quickly.

3.6 Summary

Our implementations of the CBOR format, *libcbor* and *libcbor-rb*, have been introduced. They rely on commonly used technologies, provide a well-structured API design, and have a formalized notion of ownership that allows reliable resource management without hindering performance. An important finding is that generic CBOR does not lend itself to usage with sophisticated data structures. The key techniques that were used to ensure correctness and robustness have been described as well.

```
Data: Parser stack S, item to append
if empty(S) then
    if simple_value(item) then
        push(S, item)
    else
        fail('Malformed item')
    end
else
    switch type(top(S)) do
        case ARRAY
            array_push(top(S), item);
            if definite_and_complete(top(S)) then
                append(pop(S))
            end
        end
        case MAP
            map_push(top(S), item);
            if definite_and_complete(top(S)) then
                append(pop(S))
            end
        end
        case INDEFSTRING
            if type(item) = STRING then
                append_chunk(top(S), item)
            else
                fail('Malformed item')
            end
        end
        case INDEFBSTRING
            if type(item) = BSTRING then
                append_chunk(top(S), item)
            else
                fail('Malformed item')
            end
        end
        case TAG
            tag_set_value(top(S), item);
            append(pop(S))
        end
        otherwise
            fail('Malformed item')
        end
    endsw
end
```

Listing 3.7: CBOR parsing algorithm: the append routine

Chapter 4

Using the implementation

In this chapter, we will briefly showcase how a client application might use *libcbor* and *libcbor-rb*. Please note that the examples aim to provide a conceptual overview rather than a comprehensive manual. Detailed reference and complete API documentation can be found in the attachments section.

4.1 Using the C component

4.1.1 Data items API

Decoding serialized CBOR data is easy and straightforward, as seen in listing 4.1. The snippet showcases the core of a program that reads CBOR data from a buffer and pretty-prints the result.

```
25    usage();
26  FILE * f = fopen(argv[1], "rb");
27  if (f == NULL)
```

Listing 4.1: Reading serialized CBOR (`examples/readfile.c`).

Likewise, building and serializing data items is just as simple (illustrated by listing 4.2). Notice how `cbor_move` enables fluent manipulation with intermediate values.

```c
/* Preallocate the map structure */
cbor_item_t * root = cbor_new_definite_map(2);
/* Add the content */
cbor_map_add(root, (struct cbor_pair) {
  .key = cbor_move(cbor_build_string("Is CBOR awesome?")),
  .value = cbor_move(cbor_build_bool(true))
});
cbor_map_add(root, (struct cbor_pair) {
  .key = cbor_move(cbor_build_uint8(42)),
  .value = cbor_move(cbor_build_string("Is the answer"))
});
/* Output: `length` bytes of data in the `buffer` */
unsigned char * buffer;
size_t buffer_size,
  length = cbor_serialize_alloc(root, &buffer, &buffer_size);
```

Listing 4.2: Creating and serializing CBOR data items (`examples/create_items.c`).

4.1.2 Streaming API

The streaming API is quite easy to work with as well, although the resulting code is usually more verbose. When decoding, one just has to specify the desired callbacks and repeatedly invoke `cbor_stream_decode` as needed. This technique is illustrated in listings C.2 and 4.4.

To encode data, the user can use methods such as `cbor_encode_double` that take the value to encode along with a reference to the output buffer, writing the correct headers and data to the buffer. One potential pitfall to be aware of is the use of maps and arrays, especially indefinite ones, where it is the users responsibility to correctly serialize the inner items and provide the break code. Failing to do so may result in invalid CBOR data output.

4.2 Using the Ruby component

Setting up and using *libcbor-rb* is easy and intuitive. The lowest level interface is available, but will not likely be of interest to users. The remaining two layers are, especially contrasted to *libcbor*, more flexible and can be mixed-and-matched in the same application as needed.

4.2.1 Traditional API

The user only needs to concern himself with two simple methods, `CBOR.decode` for decoding (usage is illustrated in listing 4.3)

```
8  require 'lib/libcbor'
9  require 'pp'
10
11 ARGV.each { |_| PP.pp CBOR.decode(IO.read(_)) }
```

Listing 4.3: Loading files from the command line arguments and printing the decoded Ruby structures (`examples/load_file.rb`).

4.2.2 Streaming API

The streaming API layer is very akin to that of *libcbor*, the only difference being that it provides buffering and can interact directly with sockets and IO streams.

When decoding, the user attaches a set of callbacks to `CBOR::Streaming::BufferedDecoder`. These will then be invoked when the matching item is read from the input data.

One can also use `cbor_stream_decode` directly, supplying it with Ruby callbacks. The FFI wrapper will then generate C stubs that will pass the data to the actual Ruby procedures. In fact, `CBOR::Streaming::BufferedDecoder` is built using this very mechanism.

The `CBOR::Streaming::Encoder` class provides functionality similar to the streaming encoder of *libcbor*. The key difference is that one can also encode complex CBOR structures. This layer is built on top of the *libcbor* encoders and provides the same level of control. Listing C.1 exemplifies how to use the streaming facilities for network programming.

4.3 Building custom encoders and decoders

We have discussed the reasons for choosing the layered architecture over other alternatives in section 3.1. Indeed, building custom decoders is extremely simple. One just has to provide a correct set of callback function and invoke the stateless decoder (`cbor_stream_decode`) as needed. The default decoder is built using

the very same mechanism (shown in listing 4.4), which serves as evidence of how clear-cut and transparent this design is.

```
67 if (source_size > result->read) { /* Check for overflows */
68   decode_result = cbor_stream_decode(
69     source + result->read,
70     source_size - result->read,
71     &callbacks,
72     &context);
73 } else {
74   result->error = (struct cbor_error) {
75     .code = CBOR_ERR_NOTENOUGHDATA,
76     .position = result->read
77   };
78   goto error;
79 }
```

Listing 4.4: `cbor_load`'s usage of `cbor_stream_decode` (`src/cbor.c`).

Implementing custom decoders based on *libcbor* or *libcbor-rb* is rather straightforward as well, but we put forward the view that this option will be useful only in very rare cases. Most applications will only require streaming or special control to some level, and will not be concerned with encoding their 'atomic' data items.

Finally, we should also mention the possibility of integrating *libcbor-rb* into reactor-style event loops [55]. These constructs are rather common in Ruby code that deals with networking, and *libcbor-rb* is a natural fit for providing a quintessential event source and consumer. Bearing this in mind, *libcbor* is designed to integrate well with popular abstractions, such as the widely used EventMachine [15] library. We expect this to be a popular usage pattern; a sample implementation is provided in *libcbor-rb* (see listing C.1).

4.3.1 Schema-based codecs

libcbor is suitable for implementing custom schema-based decoders, be they based on CDDL [60], or any other mechanism. This can be achieved through a variety of methods, the most common of which will probably be specializations of the CBOR parsing algorithm (shown in listing 3.3).

The complexity associated with this approach, however, will rarely be justified. If an application requires a custom codec for messages whose structure is

defined beforehand, solutions design specifically for this approach (such as Google Protocol Buffers [59]) will most likely be a better fit.

4.4 Existing applications

libcbor-rb follows idiomatic Ruby design principles and should therefore be very friendly to users seeking to use it either as a replacement or alongside existing serialization solutions, presumably predominantly JSON. In most cases, substituting `CBOR.load` for `JSON.load` and `#to_cbor` for `#to_json` will suffice.

As to *libcbor*, the integration process will not be as effortless as with *libcbor-rb*, but it will not be any more difficult than integrating other data serialization libraries. The source codes of benchmarks presented in chapter 5 provide a side-to-side comparison of programs that accomplish the same task using different formats and implementations.

4.5 Linking to other languages

A significant advantage of the design approach described in section 3.1 is that *libcbor-rb* equivalents for other high-level languages are just as easy to design and implement as *libcbor-rb*.

The reference counting approach to memory management plays a significant part in this process, as most target languages will be garbage collected. Regardless of the GC approach, reference counting is easy to integrate with.

If an implementor of a *libcbor* binding wants to give up on the reference counting approach and manage data items by the host language's facilities, it suffices to provide a fake `free` implementation and accordingly register the objects with the host language.

4.6 Summary

We have shown how *libcbor* and *libcbor-rb* can be easily used and integrated in several different ways. For custom decoders, streaming API is a convenient basis

to build upon and clients are expected to take advantage of it, whereas custom encoders will likely be rather rare.

The possibility of codecs programmatically generated from schema definition has been discussed; we have arrived at the conclusion that this approach is rarely preferable and does not align with the properties of CBOR. Finally, we have shown that *libcbor* is suitable for creating embeddings and bindings in other environments.

Chapter 5

Evaluation

In this chapter, a series of experimental measurements that will help us to evaluate various aspects of *libcbor*'s and *libcbor-rb*'s performance and efficiency is conducted. The performance is then analyzed and compared to several popular alternatives.

The encoding efficiency of CBOR in comparison to that of other formats mentioned in section 1.2 will also be investigated. This should provide us with the data based on which CBOR's suitability for different niches will be evaluated.

5.1 Performance

5.1.1 Measurement methods

The performance experiments were conducted by measuring the execution time of the most common operations that should represent real-world use cases. We have used six input data files which should represent complete a sample of all practical inputs.

The timings were determined using the PAPI framework [62]. More specifically, the `PAPI_get_real_usec` facility was used. On our test machine, this mechanism affords sub-microsecond precision[1].

The testing setup involved a common customer-grade computer with an Intel Xeon E3-1230 CPU running Linux 3.13 in a standard configuration. Some of

[1]Determined using the `papi_clockres` utility. The resolution is hardware- and software-dependent.

the memory intensive tests may also be sensitive to the memory bandwidth and latency – the setup was equipped with 32 GB of DDR3 memory running at 1333 MHz. All the libraries were compiled from the latest stable source (specific versions are noted in listing 5.1), following their authors' instructions exactly. The compiler used was GCC 4.8.2.

In some cases, especially when we discuss throughput, we present the results as 'normalized to' some particular format. In such cases, the result was scaled by the ratio of the data size in the reference format to the data size in the normalized format. This allows us to give more relevant metrics since values scaled in such a way represent comparable amount of useful information. Not taking this precaution would skew the comparison in favor of formats with less efficient encoding.

One should keep in mind that many of the comparisons we are about to present are not completely consistent or significant due to the fact that the implementations we compare differ in terms of functionality and design trade-offs. We put forward the view that the comparisons, accompanied by explanation of these differences, remains valuable regardless.

All the measurements are based on a statistically significant number of samples and include a warm-up round to eliminate the possible effects of lazy symbol resolution in shared libraries.

We have also excluded XML from these comparisons, as its real-world use cases do not align with those of other formats and the mapping between different data constructs such as arrays and booleans is largely a matter of specific implementation decision, hence any synthetic benchmarking would be largely irrelevant. For JSON, we use the minimal equivalent input (i.e. one without any unnecessary whitespace).

UBJSON is not included either, as there are hardly any resources available to work with. At the time of writing, there is no documented C library publicly available, nor are sample data or conversion tools. This fact is rather regrettable since UBJSON is, just like CBOR, a new format that is likely to see some development.

- *libbson* 1.1.5
- *libcbor* 0.3.1
- Jansson 2.6
- *msgpack-c* 1.1.0
- Yajl 2.1.0

Listing 5.1: Versions of the benchmarked implementations

5.1.2 Testing inputs

A range of input covering the whole range of available constructs is used. Please refer to appendix A for samples and a more detailed description.

- `citylots`, 181 MB: Real city planning department information; array of nested map with text, integers, decimals and booleans. Represents balanced, mixed inputs.
- `numbers`, 4.4 MB: A matrix of integers. Represents number-heavy inputs.
- `cards`, 43 MB: Collectors' card game information. Combination of maps and strings between 3 and 270 characters.
- `glossary`, 552 B: A simple map 'object' with string keys. Typical for web applications data exchange.
- `instruments`, 216 KB: A software instrumentation log. Nested map with numerical values. Common for web applications and configuration files.
- `blobs`, 65 MB: An array with many binary objects.

5.1.3 Decoding to memory

This benchmark measures the time it takes to decode a document from a memory buffer into a representation that can be used for manipulation. As mentioned in the introduction, these measurements are not directly comparable as the flexibility and complexity of the memory representations vary significantly.

For example, *libbson* uses a flat representation, where decoding is performed just by loading the data into a new buffer and performing several basic validations.

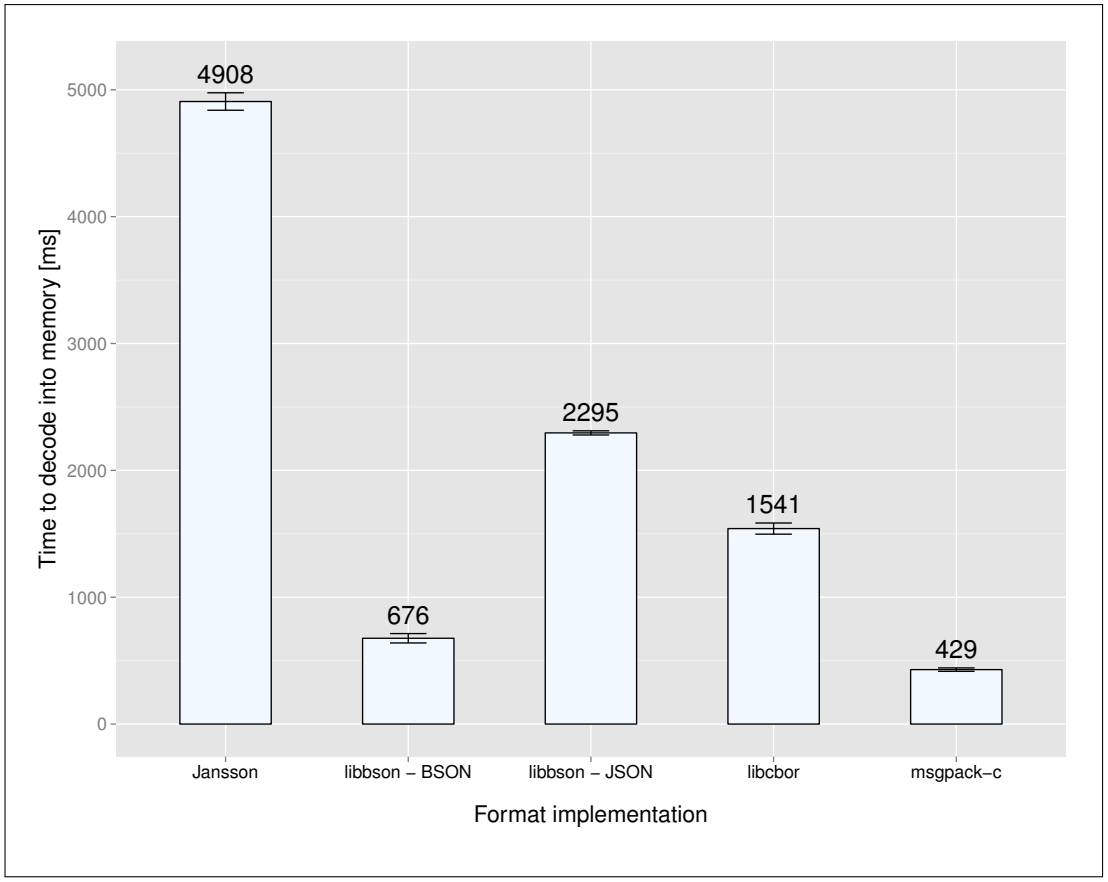


Figure 5.1: Parsing times for the `citylots` data file. Notice the difference between BSON and BSON/JSON. The whiskers denote the standard deviations of the measurements.

libcbor and Jansson, on the other hand, use a complex and flexible memory layout where each sub-item can be deallocated, replaced, and generally manipulated with independently, which inevitably comes at a performance cost.

Likewise, *msgpack-c* uses a memory pool mechanism and performs decoding with very few additional allocations beyond the initial one. This approach results in significantly higher encoding and decoding performance at the cost of flexibility and data manipulation performance.

This fact is nicely illustrated by figure 5.1, where we can see how MsgPack and *libbson* are both considerably faster than *libcbor* and Jansson because of their contiguous storage approach. Interestingly, the BSON/JSON test where we load JSON using the *libbson* library shows how *libbson*'s performance is optimized only for the aforementioned simple loading case.

When dealing with data serialization, it is often more convenient to express the performance in terms of data throughput. Figure 5.2 offers a relevant comparison

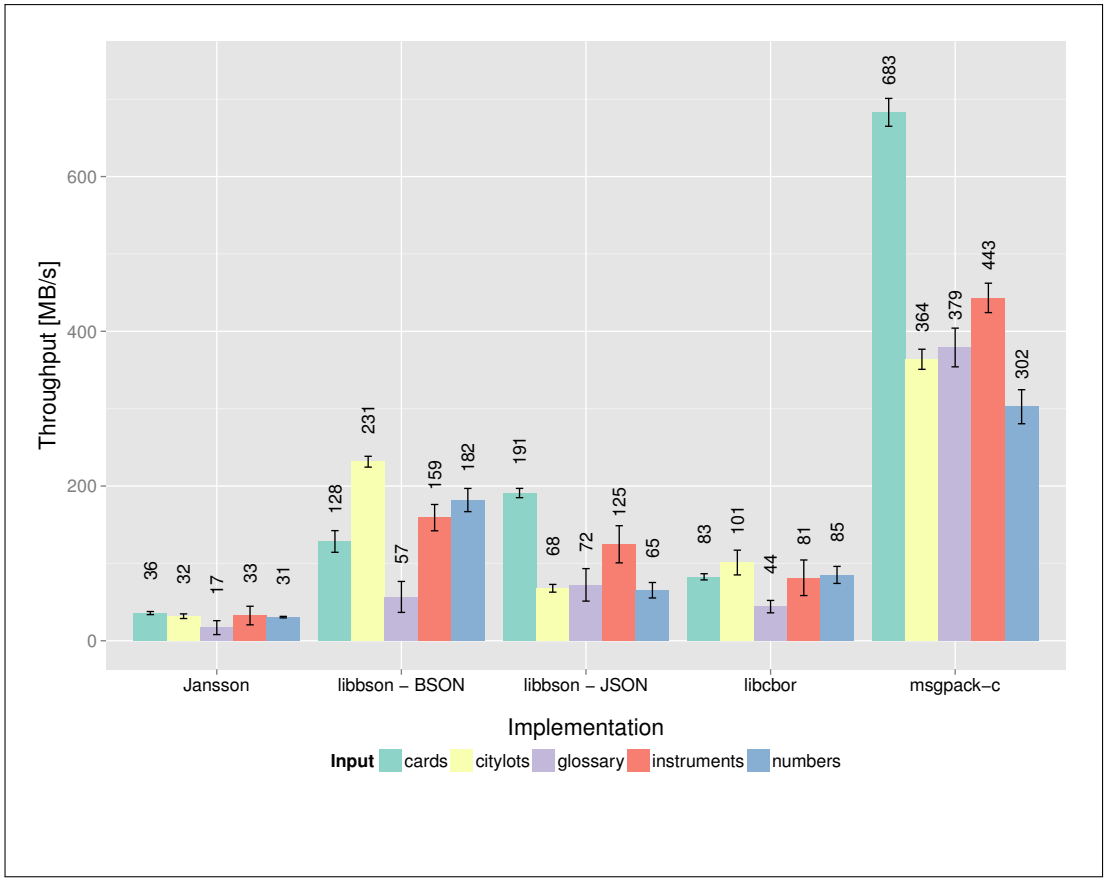


Figure 5.2: Decoding throughput by format and implementation. Normalized to minified JSON.

in terms of actual information value.

We can observe several interesting phenomena. Firstly, MsgPack's optimization allows it to decode at rates well over 200 MB/s, significantly faster than all the other implementations, which upholds the general perception of *msgpack-c* as the leading solution in terms of speed. These results suggests that a more detailed survey of the techniques used in *msgpack-c*'s implementation might be worthwhile.

Secondly, we see that the decoding performance can differ significantly based on the input data for *msgpack-c* and *libbson*. *libcbor* and Jansson, on the other hand, deliver a consistent performance regardless of the input.

Finally, we can observe that *libcbor* consistently outperforms Jansson by a factor of two or more despite the fact that they use largely similar memory representation and that Jansson is much more mature. This is likely to be a combination of the properties of CBOR and the quality of the implementations.

libcbor is also surprisingly fast in comparison to *libbson*'s JSON mode (with the exception of the string-heavy inputs), which hints at *libbson* not being optimized for anything but the 'flat memory load' approach described in the introduction.

5.1.4 Encoding

This benchmark measures the time it takes to encode a document from the memory representation to the serialized equivalent. As with the previous benchmark, *libcbor* can be compared directly to Jansson and *libbson* is directly comparable to *msgpack-c*; comparisons between members of the two pairs are only illustrative. Once again, this is due to the different flexibility-performance trade-offs in their memory representation design.

Figure 5.3 captures several important findings. Unsurprisingly, *libbson* is by far the fastest due to the fact that its memory representation is identical to the encoded data representation, hence 'encoding' is performed as a simple copying of heap memory blocks. This characteristic arises from its role in the MongoDB database, which is, in many ways, write-oriented.

Contrarily, MsgPack's performance does not meet the expectations set by the previous benchmark and is left behind by *libcbor* across all the inputs. The same goes for *libbson*'s JSON mode and Jansson, which were outperformed by up to two orders of magnitude.

This leads us to believe that the encoding performance of all the implementation except for *libbson* and *libcbor* is secondary to their decoding performance. From the practical point of view, however, encoding throughput in the order of hundreds of megabytes per second is likely to be sufficient for the vast majority of applications, particularly networking ones.

5.1.5 Manipulation

Both Jansson and *libcbor* provide the user with data manipulation routines. In this benchmark, we measure the time it takes to build a fairly big array with booleans, strings, and numbers. The array is then traversed and the values are updated.

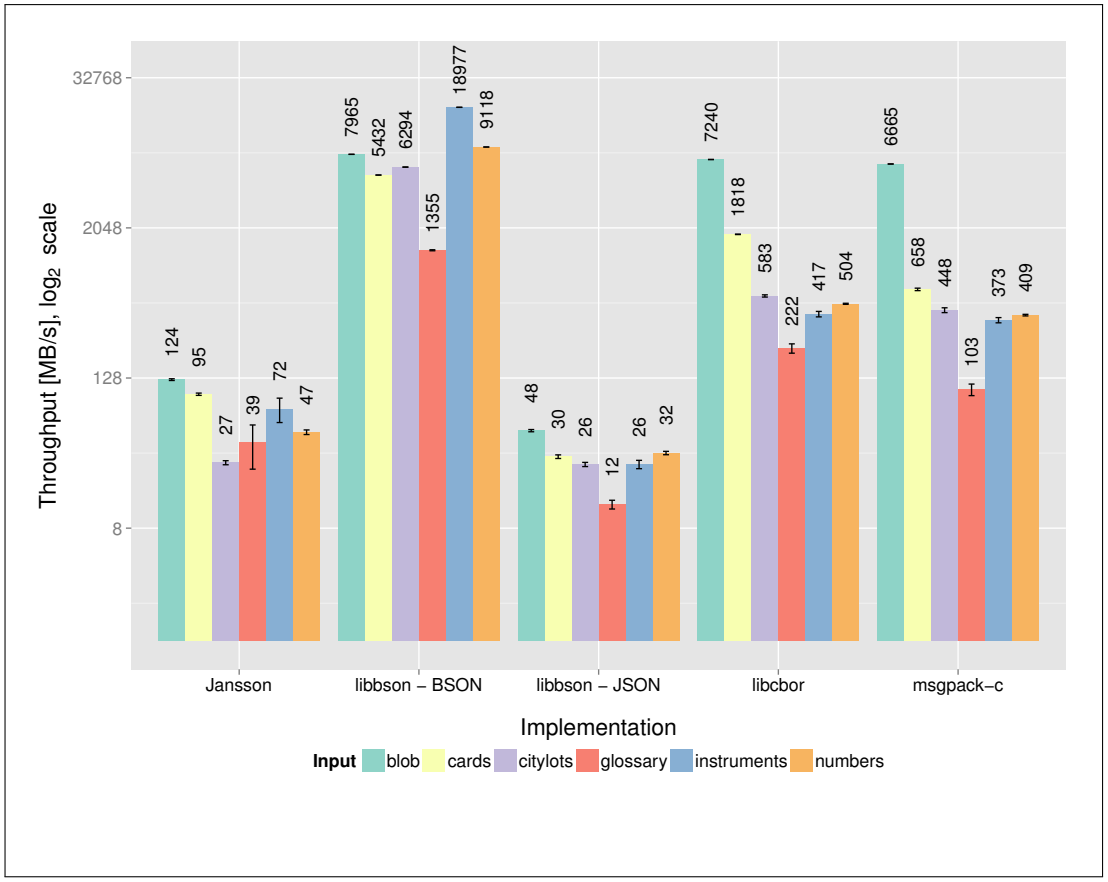


Figure 5.3: Encoding throughput by format and implementation. Normalized to minified JSON. Notice the logarithmic scale.

Figure 5.1.6 shows that *libcbor* and Jansson are closely tied, with Jansson being about 5 % faster. This is likely down to the simplicity of JSON (only one integer width, no distinctions between definite and indefinite items) and Jansson's maturity.

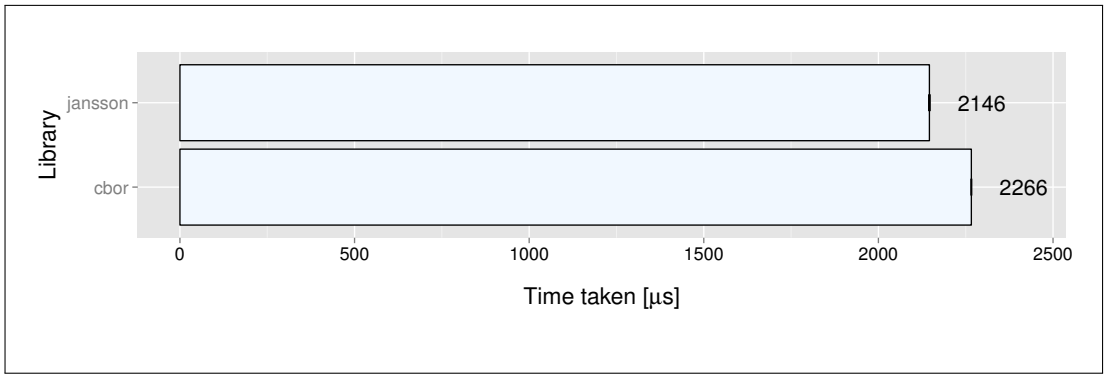


Figure 5.4: Manipulation benchmark.

5.1.6 Event emitting

This test is focused on the speed of emitting parser events. As this technique is only supported by *libcbor* and *msgpack-c*, Yajl, the de facto standard for parsing JSON in a streaming manner, will also be included in this benchmark in order to provide a standard for reference.

Figure 5.5 shows how *libcbor* outperforms both *msgpack-c* and Yajl, achieving 761 MB/s geometric mean throughput, compared to 433 MB/s delivered by *msgpack-c* and 233 MB/s achieved by Yajl.

Another observation is that the event emitting performance varies between different inputs even more significantly than the encoding and decoding performance. The outstandingly low performance of *msgpack-c* for the `glossary` input shows the cost of its sophisticated approach that manifests itself through a significant initialization overhead.

CBOR, on the other hand, performs rather well on the `glossary` input, which is in line with one of its intended use cases in web applications.

Finally, Yajl is consistently slower than both *libcbor* and *msgpack-c*, leading us to the conclusion that both MsgPack and CBOR are more suitable for high-volume streaming applications not only due to their encoding efficiency, but also because of the superior decoding performance.

5.2 Encoding efficiency

In this section, the encoding efficiency of the formats whose implementations were benchmarked in section 5.1 is examined. We use the same set of inputs as well, since they accurately represent the actual and anticipated use cases.

The first result to notice in figure 5.6 is the efficiency (or lack of thereof) of BSON. Overall, it is the least efficient format of the four. The liberal encoding strategy for numbers and simple values contributes to the relatively high performance demonstrated in section 5.1, but it also renders it even less efficient than JSON.

As with the encoding strategy, this property stems from its original purpose of a database serialization format. While it does still offer a binary data type that

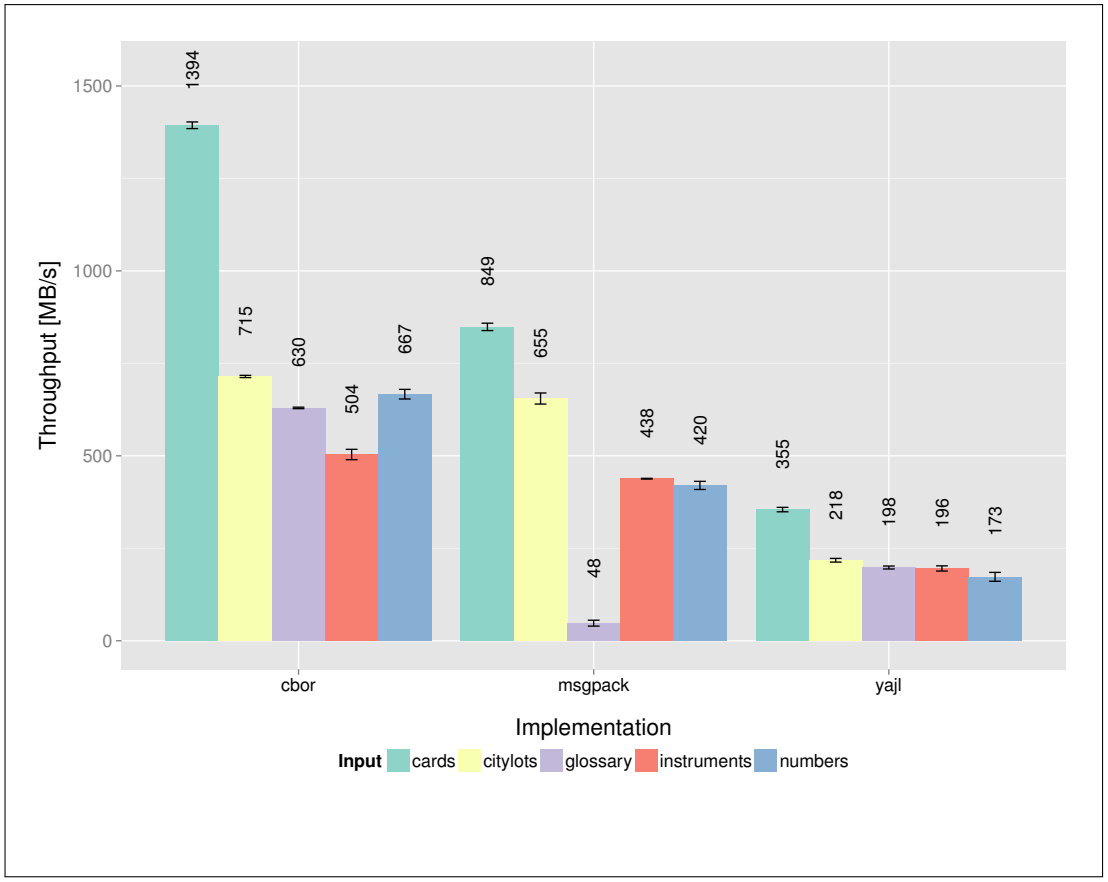


Figure 5.5: Event emitting throughput by format and implementation. Normalized to minified JSON.

can be used to embed resources and therefore save multiple network requests, BSON's inefficiency prevents it from being considered a relevant alternative to the other formats, at least in domains involving data transmission.

JSON, on the other hand, compares surprisingly well to the binary alternatives. While it produces files between 13 % to 48 % larger compared to the CBOR/MsgPack tandem, its string encoding is almost as efficient as that of CBOR, resulting in solid performance on string-heavy inputs. It falls short on inputs containing long numbers and special values, as expected.

The final conclusion to be drawn from these data is that CBOR and MsgPack are very closely tied in terms of encoding efficiency. The largest difference measured is 5.8 % in favor of MsgPack for the `citylots` input, whereas CBOR performs better for both `numbers` and `cards`.

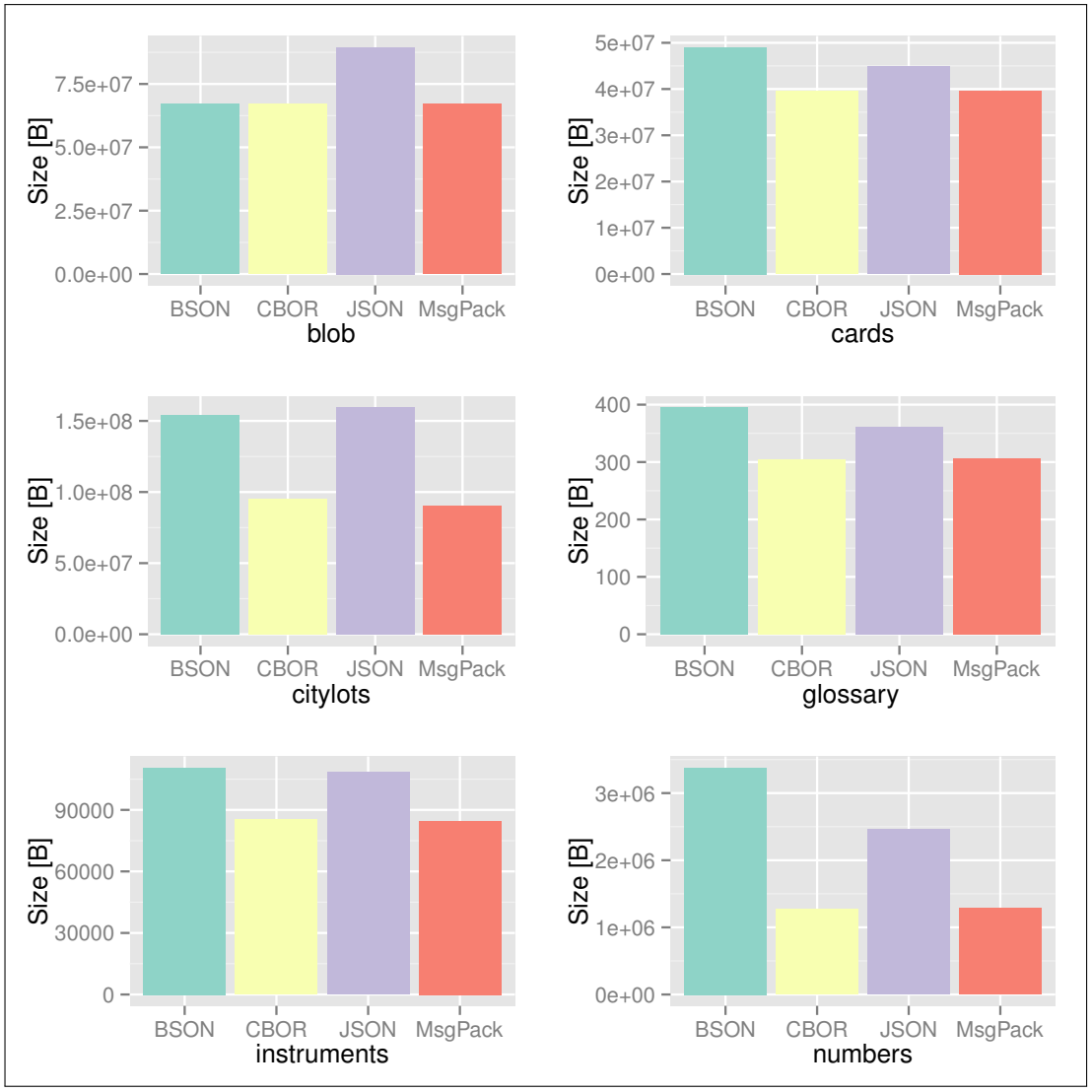


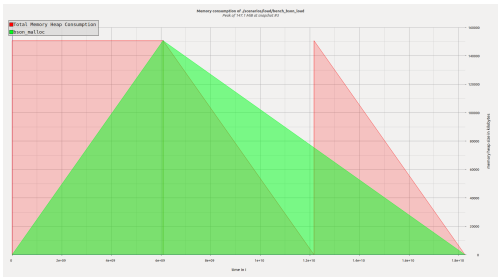
Figure 5.6: Sizes of the testing data serialized in different formats.

5.3 Memory usage

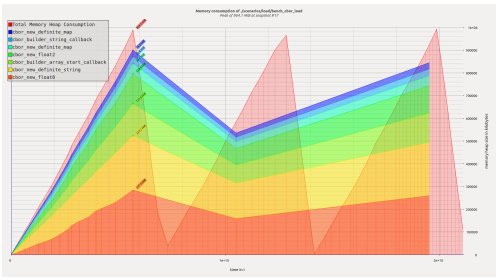
Figure 5.7 represents the heap profiles of three iterations of the decoding benchmark from subsection 5.1.3 taken using Valgrind's `massif` [45] tool and the `citylots` input. By instrumenting the allocation mechanism and sampling the heap at regular intervals, it provides a graph of memory allocated over time including information about which functions the allocations originated from.

One can observe that the libraries have fairly dissimilar memory profiles, as well as the total memory usage. Unsurprisingly, *libbson* allocates the least memory, while *libcbor* and Jansson top the chart due to the high overhead of their approach.

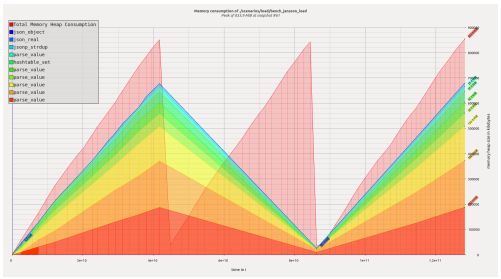
We can also see how *libbson* allocates all the memory in advance when it starts



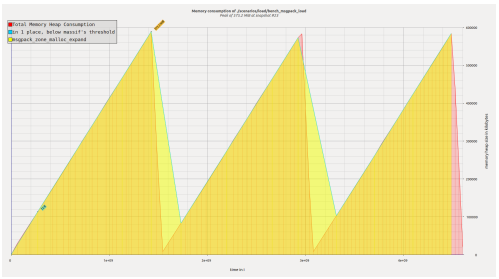
(a) *libbson*



(b) *libcbor*



(c) Jansson



(d) *msgpack-c*

Figure 5.7: Memory usage profiles for loading of the `citylots` input. Please refer to appendix B for the full-scale version.

the decoding process, whereas the remaining libraries allocate space gradually as needed. This is another reason why *libbson* tops the chart.

5.4 Complexity

A final part of the evaluation will be a brief survey of the complexity of the aforementioned libraries. It is important for several reasons, ranging from the effort required for maintaining the code base to the expected size of the compiled binary, which still remains a concern for embedded devices and the Internet of things.

Since measures of software complexity are a fairly convoluted topic that lies outside the scope of this text, only two simple, universally understood measures are used, namely LoC count (excluding tests and configuration files) and the size of the resulting shared library without any debugging symbols.

	libbson	*libcbor*	Jansson	*msgpack-c*	Yajl
`.so` size (KB)	177	64	45	163	35
LoC	10649	3363	4012	32534	2429

Table 5.1: Size and complexity facts

Based on table 5.1, we can see that *libbson* and *msgpack-c* have significantly larger code bases compared to the remaining libraries. This complexity is also reflected in their shared object size, which could be a disadvantage in somewhat memory restricted environments. Yajl lies at the other hand of the spectrum, with just 2.5K lines of code.

One reason for the relatively large size of *msgpack-c*'s code base is the extensive use of templates and their specializations. This technique can eliminate many virtually resolved calls, and plays a significant role in the high performance achieved in the benchmarks in section 5.1.

The combination of these characteristics predetermine *msgpack-c* towards usage in high-volume applications where throughput is critical, as opposed to mobile computing and restricted nodes. The key thing to keep in mind, however, is that this pertains only to the particular implementation, not to the protocol itself.

5.5 Summary

This chapter presented an evaluation of *libcbor* in terms of performance and complexity, as well as a comparison of JSON, BSON, CBOR, and MsgPack in terms of encoding efficiency.

Despite the fluctuations depending on the input data and the nature of benchmark, the data show that *libcbor* is consistently faster than two highly regarded and commonly used JSON implementations, regardless of the fact that it combines both of their specialized approaches. *msgpack-c* excels in decoding data to memory, but is left behind by *libcbor* in event driven decoding. What is more, its performance comes at the cost of a very high complexity.

The efficiency comparison has shown that BSON is by far the least efficient of the four. JSON, on the other hand, comes surprisingly close to CBOR and MsgPack, especially for string-heavy inputs. CBOR and MsgPack provide an almost identical encoding efficiency.

Chapter 6

Discussion

This chapter provides a discussion of the findings from chapter 5. We recapitulate the key challenges that our implementation was faced with and critically reflect on the CBOR standard design from the standpoint of practical implementation.

Furthermore, several improvements to remedy the most pressing issues are proposed. Some of them are compatible with the existing standard, others would require breaking changes in the specification.

The appropriate use cases for CBOR are discussed; this list is of an illustrative nature since CBOR is very flexible and can be used in many unexpected ways.

Finally, the alternative approaches to implementing CBOR are discussed in terms of their expected performance, use cases, and general viability.

6.1 CBOR semantics

There are seven main areas where, based on the previous chapters, CBOR either does not meet its original design objectives or forces implementors to resort to excessively complex approaches. This section will describe these areas, whereas section 6.2 lists the proposed solutions.

6.1.1 Unclear distinction between well-formedness and validity

As discussed in subsection 2.2.1, the distinction between well-formedness and validity remains unspecified for text strings.

6.1.2 Too complex map keys

The arbitrarily complex structure of map keys is problematic for both the decoder and the memory representation, as shown in subsection 2.2.3, hence it directly violates the second design goal stated in the RFC [7, s. 1.1].

6.1.3 No signed numeral type

Although the lack of signed types (see subsection 2.1.1.4) contributes to the efficiency of encoding, it maps extremely poorly to fixed-width integers in C and many other languages. The logical value of $n - 1$ is utterly counter-intuitive and is likely to be a source of programming errors.

6.1.4 Combination of `null` and `undefined value`

As with the previous point, these constructs map poorly to the constructs of the vast majority of programming languages and will likely create unnecessary confusion.

6.1.5 Lack of *no-op*-like construct

A rather useful feature of UBJSON (introduced in 1.2.7.5) is the *no-op*, an auxiliary data items that is used to prevent network timeouts in protocols such as HTTP. The lack of this feature in CBOR renders all the sophisticated streaming features significantly less valuable.

6.1.6 Unspecified relationship between definite and indefinite items

Whether or not definite and indefinite maps and arrays with the same inner value are equivalent is crucial for designing high-level implementations that should arguably provide functionality such as equality comparison. It would also be useful for implementing more efficient hash map data structures (see subsection 2.2.3).

6.1.7 Implementation complexity

The standard requires that "the code for an encoder or decoder must be able to be compact" [7, s. 1.1]. This can only be the case for simple implementations that do not need to concern themselves with memory representation or robust error handling. With the two requirements present, the complexity of CBOR implementations surpasses that of JSON implementations due to the richer set of available constructs and a roughly similar complexity in other areas.

6.2 Proposed improvements

6.2.1 Unclear distinction between well-formedness and validity

Simply clarifying all the corner cases in the next revision of the standard should remedy this deficiency. More specifically, as argued in subsection 2.2.1, the validity UTF-8 text strings should be dealt with on the well-formedness level.

6.2.2 Too complex map keys

Only very few applications will take advantage of the possibilities arising from this feature. Restricting map keys to definite length strings and integers would likely cover most real-world use cases while significantly reducing the complexity of generic implementations.

6.2.3 No signed numeral type

This point cannot be argued for by means of objective evidence, but we put forward the view that replacing negative integers (2.1.1.4) with plain signed integers would greatly simplify type mapping in virtually all application environments, whereas the drop in efficiency caused by the value range overlap between signed and unsigned numerals would likely be, for most practical payloads, marginal to insignificant.

6.2.4 Combination of `null` and `undefined value`

Dropping one of the values from the protocol would suffice.

6.2.5 Lack of *no-op*-like construct

A *no-op* with the same semantics as the one found in UBJSON should be introduced as one of the simple values. Since there are unassigned one byte simple values still available, this can be done in a backwards-compatible manner in one of the future revisions.

6.2.6 Unspecified relationship between definite and indefinite items

The protocol specification should include a normative recommendation on this issue. From the application point of view, definite and indefinite items are most likely equivalent, hence generic codecs should, at least by default, handle them in the corresponding manner.

6.2.7 Implementation complexity

The protocol specification should address error handling in more detail. More specifically, defining several levels of error handling would allow very simple approaches for constrained nodes while allowing for consistent error detection and handling in a more complicated setting.

Alternatively, this could be integrated into the CDDL [60] proposal for universal schemata definition. The question of whether or not it is appropriate to handle these errors at a higher level of abstraction is outside the scope of this text; it should be noted that even a CDDL-based approach will require a certain level of cooperation on the codecs side.

6.3 Appropriate use cases

Since CBOR is a relatively recent technology that has not yet had much public exposure, exploring potential use cases is a worthwhile undertaking, as already argued in section 1.3. Based on the findings from the previous chapters, CBOR is suitable for a variety of applications, although the cost of introducing it into existing applications might not always be justified.

6.3.1 Web applications

CBOR is certainly a viable alternative to JSON, although the efficiency gain (shown to be between between 13 % and 48 % in section 5.2) is unlikely to be the key factor. The ability to embed binary resources and thus save multiple requests, especially in situations where new requests require establishing a new connection, is presumably of greater value in terms of overall efficiency and performance. Due to their dependence on a number of variables and application specifics, these benefits are hard to quantify or measure.

Usage with the WebSockets API appears to be very feasible, especially for multimedia-heavy applications that can take advantage of binary embedding. It should be noted, however, that such applications would most likely rely on WS API for streaming, hence the streaming functionality of CBOR will likely remain unused in this context.

6.3.2 REST-style APIs

The extensible semantics and tagging system could work well with REST-like [21] APIs, where version negotiation and specific semantics often come into play, but a generic, schema-free format is still preferred. The feasibility of this usage

will largely depend on the popularity and completeness of the tag repository (as discussed in subsection 2.1.2.3).

6.3.3 Internet of things

CBOR is suitable for communication between low-powered devices due to its relatively compact encoding. Suitability for embedded devices was one of the original design goals of CBOR. In spite of the reservations and criticisms outlined in section 6.1, the measurements suggest that it has been met, at least to some extent.

6.3.4 Piggybacking on other protocols

One unexpected application of CBOR that has already been reported by a *libcbor* user is piggybacking on other protocols and transport mechanisms. The aforementioned application uses CBOR to pack data payloads into the ZeroMQ [31] messaging queue messages. The fact that CBOR is schema-less enables zero-downtime upgrades of producers and consumers in the messaging system. Despite being quite surprising, this usage is perfectly reasonable and demonstrates that CBOR can be used in many other contexts we had not consider.

6.3.5 Creating new protocols

The possibility of building custom CBOR-based protocols is discussed in the standard [7, s. 3], and indeed, the observations made throughout the previous chapters support this vision. *libcbor* can feasibly be used as a basis for such protocols, especially the streaming API layer.

6.4 Alternative implementation designs

Chapter 5 explains, among other things, how the different approaches to implementing CBOR and similar protocols affect properties such as performance, complexity, or memory consumption. Although *libcbor* compares to the alternatives

fairly favorably, there might be situations where a different CBOR implementation would be more suitable.

6.4.1 Flat memory representation

libbson's approach described in subsection 5.1.3 could work just as well for CBOR. This strategy is suitable when data items will not be modified, or at least not frequently. The expected encoding and decoding performance is likely to surpass that of *libcbor*, at the cost of flexibility. This approach is likely to be of a lower complexity than that of *libcbor*.

6.4.2 Flat memory representation with parse trees

A more sophisticated version of the previous technique is to use a hybrid approach where the data structure is not subdivided into atomic items and is represented in a flat manner and a (contiguously stored) parse tree is stored alongside with it, enabling fast queries and meta data extraction (e.g. array members count, aggregated depth of structure).

It is suitable for read-oriented applications with the need for more advanced manipulation; modifying data items remains expensive. The encoding and decoding performance is expected to trounce *libcbor*; the complexity of implementation will likely be higher due to the need for a sophisticated parse tree construction approach that will be required so as to allow it to use just one memory block.

6.4.3 Purely event-driven parser

Some lightweight applications may, for example, only extract particular values (as shown in example C.2), or perhaps just read the data for monitoring or similar purpose. If that is the case, a stateless decoder (e.g. *libcbor*'s streaming API layer) will likely suffice. Both speed and complexity will likely be very favorable, as shown in subsection 5.1.6.

6.4.4 Advanced memory control

The final alternative is inspired by *msgpack-c*, which uses a sophisticated lazy parsing strategy with memory management based on memory pools (zones in figure 6.1), preallocation, and reference counting. The substantial complexity and sophistication can enable impressive performance, especially when decoding (see 5.1), but is not necessarily superior for all usage patterns.

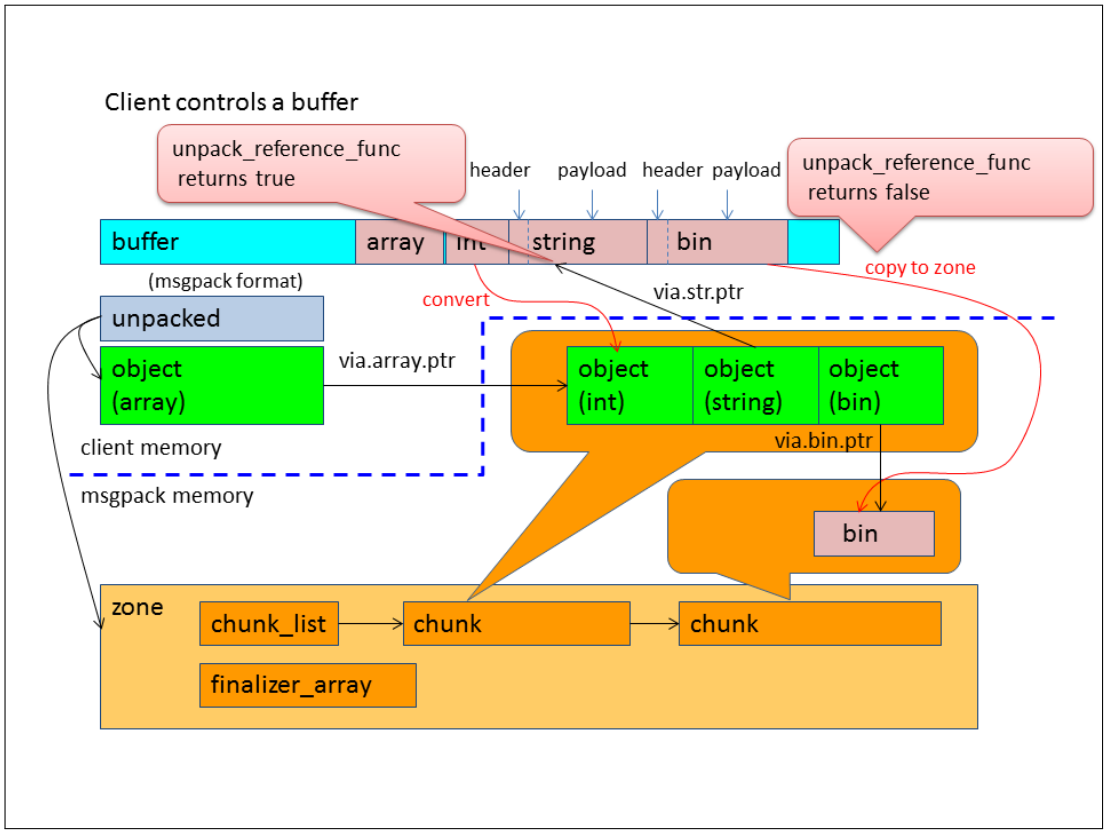


Figure 6.1: Diagram of *msgpack-c*'s memory management mechanism. Adapted from the official website [16].

Manipulation with data items still comes at a nontrivial cost, but offers significantly better performance than all of the previous alternatives. This implementation design is likely to be suitable for complex, extensively optimized applications dealing with high volumes of data.

6.5 Summary

The problematic aspects of CBOR and the proposed solutions were discussed. Several of them would require changing the existing standard in one way or

another. In spite of these problems, CBOR meets most of its design objectives and has good prospects to become one of the commonly used formats alongside JSON and MsgPack.

As we list the viable niches, it becomes apparent that despite the declared difference in goals [7, Appendix E], CBOR's and MsgPack's domains have a significant overlap. Although *libcbor* already has several known users, the adoption trend remains largely unpredictable for the near future.

Finally, alternative approaches to implementing generic CBOR codecs and decoders were discussed and compared in terms of their expected performance, design, and complexity properties. The approaches predominantly derive from techniques found in existing implementations of other protocols, and will likely be beneficial in specific contexts.

Chapter 7

Conclusions and future work

This chapter will discuss the extent to which the goals laid out in section 1.4 have been met. Several possible directions for future work and development will be mentioned as well.

7.1 C implementation

In spite of the challenges described in chapter 3, we have addressed and met all the goals and objectives. *libcbor* is fully standard compliant, the layered API is easy to use and offers both convenience and tight control, and the implementation is extraordinarily well tested and robust.

The goals also required that the performance would match that of popular JSON implementations. *libcbor* has exceeded the expectations in all the areas, including encoding, decoding, manipulation, and streaming decoding. Based on measurements from section 5.1, *libcbor* outperforms Yajl, Jansson, and *libbson*'s JSON mode in nearly all the tests, sometimes by up to two orders of magnitude.

In many cases, *libcbor*'s performance even matches or surpasses that of *msgpack-c*, which is a significant achievement and a testimonial to the quality of *libcbor*'s design, considering *msgpack-c*'s maturity and extensive optimizations.

Overall, *libcbor* is a production quality implementation that offers a solid foundation for working and experimenting with CBOR. Its performance is sufficient for almost any domain; nevertheless, there is potential for improvement and optimization. If required, matching the performance profile of *msgpack-c* is

certainly possible by applying the same techniques.

Perhaps due to factors such as practical design and development practices, extensive documentation, the informative web site (`http://libcbor.org`), and the permissive open source licensing, *libcbor* has been receiving the attention of the community from its earliest days and already has self-reporting users. We are therefore faced with an obligation to maintain and improve *libcbor* for the months and years to come.

The aforementioned improvements could be in several areas: performance, tutorials and more high-level documentation, and the packaging and distribution process. While *libcbor* already adheres to a high standard in all of these fields, there is definitely room for improvement. One major effort that is underway is the inclusion of *libcbor* in the standard package repository of Ubuntu and OS X's *homebrew* packaging systems.

7.2 Ruby implementation

libcbor-rb has fulfilled the goals as well. The *libcbor* binding is elegant and concise, whereas the rich selection of Ruby additions allow it to integrate effortlessly with most existing Ruby code, rendering it remarkably easy to use.

Just like *libcbor*, *libcbor-rb* presents its users with a high standard of software engineering in all key areas. The descriptive executable specification makes it extremely easy to modify and improve with confidence, which is an important trait for a quickly developing software.

Unsurprisingly, *libcbor-rb* already has a handful of users as well. Nevertheless, the possible improvements for *libcbor* outlined in the previous section apply to it just as much, with the addition of more work on integration with common web-related technologies. Such integration should enable even smoother adoption for users wishing to evaluate CBOR for use in their applications.

7.3 Evaluation

All the measurements and experiments outlined in the introduction were conducted and analyzed. The value of these measurements has been somewhat undermined by the fact that many of the alternatives were found to be mutually incomparable, or at least not directly comparable.

Based on the evaluation, we have concluded that CBOR is closely tied to MsgPack in most areas. Although their design goals differ, MsgPack and CBOR are likely to provide almost identical performance, efficiency, and feature set in many domains. This leads us to believe that more detailed analyses of CBOR designed with respect to particular niches should be conducted.

Considering all the aforementioned arguments, the future adoption of CBOR remains largely unpredictable. Although a number of CBOR's properties, use cases, and potential deficiencies have been investigated, the conclusions might differ depending on the viewpoint. Chapter 5 has concluded that, while a viable alternative to JSON or MsgPack, CBOR's unique combination of features does not give it a significant competitive edge over these formats in their respective domains. Instead, it might have a bigger impact in emerging fields such as the Internet of things, with the reservations pointed out in section 6.1.

In summary, the evaluation has provided the first comprehensive inquiry into the mechanics and characteristics of the CBOR protocol, arriving at surprising and relevant findings in areas such as performance, efficiency, or the semantics and features. The conclusions will be of value to anyone who is either considering using CBOR, looking to compare different data serialization formats, or creating a new, similar protocol.

Bibliography

[1] *ActiveModel::Serializers*. URL: https://github.com/rails-api/active_model_serializers (visited on 04/25/2015).

[2] Gavin Andersen. *Bitcoin JSON-RPC documentation*. URL: https://en.bitcoin.it/wiki/Proper_Money_Handling_(JSON-RPC) (visited on 04/11/2015).

[3] J Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: the definitive guide*. O'Reilly Media, Inc., 2010. ISBN: 8184049471.

[4] Chad Austin. *JSON Parser Benchmarking*. URL: http://chadaustin.me/2013/01/json-parser-benchmarking/ (visited on 04/22/2015).

[5] Kyle Banker. *MongoDB in action*. Manning Publications Co., 2011. ISBN: 1935182870.

[6] Mike Belshe and Roberto Peon. *SPDY protocol*. IETF Network Working Group, 2012.

[7] C. Bormann and P. Hoffman. *Concise Binary Object Representation (CBOR)*. RFC 7049 (Proposed Standard). Internet Engineering Task Force, Oct. 2013. URL: http://www.ietf.org/rfc/rfc7049.txt.

[8] Carsten Bormann. *CBOR official webpage*. URL: http://cbor.io/impls.html (visited on 04/18/2015).

[9] Carsten Bormann. *cbor-ruby*. URL: https://github.com/cabo/cbor-ruby (visited on 04/22/2015).

[10] Carsten Bormann. *cn-cbor: A constrained node implementation of CBOR in C*. URL: https://github.com/cabo/cn-cbor (visited on 04/22/2015).

[11] Carsten Bormann et al. *Concise Binary Object Representation (CBOR) Tags*. 2013. URL: http://www.iana.org/assignments/cbor-tags/cbor-tags.xhtml.

[12] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 7159 (Proposed Standard). Internet Engineering Task Force, Mar. 2014. URL: http://www.ietf.org/rfc/rfc7159.txt.

[13] D. Brownell. *SAX2*. O'Reilly Series. O'Reilly, 2002. ISBN: 9780596002374. URL: http://books.google.cz/books?id=WrZQAAAAMAAJ.

[14] J Morris Chang, Yusuf Hasan, and Woo H Lee. "A high-performance memory allocator for memory intensive applications". In: *High Performance Computing in the Asia-Pacific Region, 2000. Proceedings. The Fourth International Conference/Exhibition on*. Vol. 1. IEEE. 2000, pp. 6–12.

[15] Francis Cianfrocca. *EventMachine*. URL: https://github.com/eventmachine/eventmachine (visited on 04/25/2015).

[16] msgpack-c contributors. *msgpack-c implementation website*. URL: https://github.com/msgpack/msgpack-c (visited on 04/25/2015).

[17] Crispin Cowan et al. "Buffer overflows: Attacks and defenses for the vulnerability of the decade". In: *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*. Vol. 2. IEEE. 2000, pp. 119–129.

[18] Scott A Crosby and Dan S Wallach. "Denial of Service via Algorithmic Complexity Attacks." In: *Usenix Security*. Vol. 2. 2003.

[19] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach, Revised*. Course Technology, 1998. ISBN: 0534954251.

[20] Jon Ferraiolo. *Scalable Vector Graphics (SVG) 1.0 Specification*. W3C Recommendation. http://www.w3.org/TR/2001/REC-SVG-20010904. W3C, Sept. 2001.

[21] Roy Fielding. "Representational state transfer". In: *Architectural Styles and the Design of Netowork-based Software Architecture* (2000), pp. 76–85.

[22] Sadayuki Furuhashi. *MessagePack specification*. 2014. URL: https://github.com/msgpack/msgpack/blob/master/spec.md (visited on 03/19/2015).

[23] Sadayuki Furuhashi. *MsgPack official website*. URL: http://msgpack.org/ (visited on 04/18/2015).

[24] Sadayuki Furuhashi. *msgpack-ruby: MessagePack implementation for Ruby*. URL: https://github.com/msgpack/msgpack-ruby (visited on 04/22/2015).

[25] Sadayuki Furuhashi. *Streaming Serialization/Deserialization with MessagePack*. URL: https://msgpack.wordpress.com/2010/05/18/streaming-serializationdeserialization-with-messagepack/ (visited on 04/18/2015).

[26] David Goldberg. "What every computer scientist should know about floating-point arithmetic". In: *ACM Computing Surveys (CSUR)* 23.1 (1991), pp. 5–48.

[27] JSON-RPC Working Group. *Json-rpc 2.0 specification*. 2012.

[28] The Open Group. *The Open Group Base Specifications*. 2013. URL: http://pubs.opengroup.org/onlinepubs/9699919799/.

[29] Ian Hickson. *The WebSocket API*. Candidate Recommendation. W3C, Sept. 2012. URL: http://www.w3.org/TR/2012/CR-websockets-20120920/.

[30] Lloyd Hilaiel. *YAJL*. URL: https://github.com/lloyd/yajl (visited on 04/22/2015).

[31] Pieter Hintjens. *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, Inc., 2013. ISBN: 1449334067.

[32] *IEEE Standard for Floating-Point Arithmetic*. Tech. rep. 3 Park Avenue, New York, NY 10016-5997, USA: Microprocessor Standards Committee of the IEEE Computer Society, Aug. 29, 2008, pp. 1–70. URL: http://dx.doi.org/10.1109/ieeestd.2008.4610935.

[33] International Organization for Standardization. *ISO 8879:1986: Information processing — Text and office systems — Standard Generalized Markup Language (SGML)*. Geneva, Switzerland: International Organization for Standardization, Aug. 17, 1986, p. 155. URL: http://www.iso.ch/cate/d16387.html.

[34] ISO. *ISO C Standard 1999*. Tech. rep. ISO/IEC 9899:1999 draft. 1999. URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf.

[35] Hartmut Kaiser Joel de Guzman. *Boost.Spirit*. URL: http://www.boost.org/doc/libs/1_48_0/libs/spirit/doc/html/index.html (visited on 04/22/2015).

[36] *JSON official homepage*. URL: http://www.json.org/ (visited on 04/11/2015).

[37] Alin Jula and Lawrence Rauchwerger. "Two memory allocators that use hints to improve locality". In: *Proceedings of the 2009 international symposium on Memory management*. ACM. 2009, pp. 109–118.

[38] Takuki Kamiya and John Schneider. *Efficient XML Interchange (EXI) Format 1.0*. W3C Recommendation. http://www.w3.org/TR/2011/REC-exi-20110310/. W3C, Mar. 2011.

[39] Ben Klemens. *21st Century C: C Tips from the New School*. O'Reilly Media, 2012. ISBN: 1449327141.

[40] Charles W Krueger. "Software reuse". In: *ACM Computing Surveys (CSUR)* 24.2 (1992), pp. 131–183.

[41] Chris Lattner. "LLVM and Clang: Next generation compiler technology". In: *The BSD Conference*. 2008, pp. 1–2.

[42] Petri Lehtinen. *Jansson*. URL: http://www.digip.org/jansson/ (visited on 04/22/2015).

[43] Ken Martin and Bill Hoffman. *Mastering CMake*. Kitware, 2010. ISBN: 1930934319.

[44] MongoDB team. *BSON specification*. URL: http://bsonspec.org/ (visited on 04/18/2015).

[45] Nicholas Nethercote and Julian Seward. "Valgrind: a framework for heavy-weight dynamic binary instrumentation". In: *ACM Sigplan notices*. Vol. 42. 6. ACM. 2007, pp. 89–100.

[46] Charles Nutter. *JRuby Ruby implementation*. URL: http://jruby.org/ (visited on 04/25/2015).

[47] Aleph One. "Smashing the stack for fun and profit". In: *Phrack magazine* 7.49 (1996), pp. 14–16.

[48] J Paoli et al. "ECMA-376 office open XML file formats". In: *URL http://www.ecmainternational. org/publications/standards/Ecma-376. htm* (2006).

[49] Tom Preston-Werner. *Semantic Versioning 2.0.0*. URL: http://semver.org/ (visited on 04/25/2015).

[50] *RethinkDB*. URL: http://rethinkdb.com/ (visited on 04/11/2015).

[51] RIOT OS Team. *RIOT OS*. URL: http://www.riot-os.org/ (visited on 04/22/2015).

[52] Ruby FFI team. *Ruby FFI*. URL: https://github.com/ffi/ffi (visited on 04/25/2015).

[53] *Ruby on Rails web framework*. URL: http://rubyonrails.org/ (visited on 04/25/2015).

[54] *SC22/WG14. ISO/IEC 9899: 2011*. URL: http://www.iso.org/iso/iso_catalogue/catalogue_%20tc/catalogue_detail.htm.

[55] Douglas C Schmidt. *Reactor: An object behavioral pattern for concurrent event demultiplexing and dispatching*. 1995.

[56] Andreas Schneider. *CMocka testing tool*. URL: https://cmocka.org/ (visited on 04/25/2015).

[57] Yonik Seeley. *Noggit JSON streaming parser*. URL: http://yonik.com/noggit-json-parser/ (visited on 03/24/2015).

[BZ] The Buzz Media, LLC. *Universal Binary JSON Specification*. URL: http://ubjson.org/ (visited on 04/18/2015).

[58] Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby: The Pragmatic Programmers' Guide, Second Edition*. Pragmatic Bookshelf, 2004. ISBN: 0974514055.

[59] Kenton Varda. *Protocol buffers: Google's data interchange format*. 2008.

[60] Christoph Vigano, Henk Birkholz, and Ruinan Sun. *CBOR data definition language: a notational convention to express CBOR data structures.* Internet-Draft. IETF Secretariat, Mar. 2015. URL: http://www.ietf.org/internet-drafts/draft-greevenbosch-appsawg-cbor-cddl-05.txt.

[61] Ian Ward. *Documentation for the JSON Lines text file format*. URL: http://jsonlines.org/ (visited on 03/24/2015).

[62] Vincent M Weaver et al. "PAPI 5: Measuring power, energy, and the cloud". In: *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*. IEEE. 2013, pp. 124–125.

[63] F. Yergeau. *UTF-8, a transformation format of ISO 10646*. RFC 3629 (INTERNET STANDARD). Internet Engineering Task Force, Nov. 2003. URL: http://www.ietf.org/rfc/rfc3629.txt.

[64] Qu Zhan and Li Chan. "Application of JSON in Ajax data exchange". In: *Journal of Xi'an Shiyou University (Natural Science Edition)* (2011), p. 024.

List of Tables

List of listings

List of Figures

Glossary

ABI Application Binary Interface. 55

CI Continous integration. 57, 58

codec a software that can both encode and decode data. 8, 12, 82

FFI Foreign Function Interface. 18, 44, 62

GC Garbage collection. 64

gem is a commonly used term for a packaged Ruby library or component. 36, 43, 57

JNI a native FFI interface of the JVM. 44

LoC Line(s) of code. 76

major type fundamental data type of CBOR. 19

major type byte the leading byte of a CBOR item containing the 3 bit major type description. 19, 20, 46

OO object oriented. 44, 48, 52

PAPI Performance Application Programming Interface. 66

REST Representational State Transfer, a design paradigm for building HTTP-based applications. 16, 82, 95

Attachments

Attached to this printout you will find a CD containing the following attachments:

- The text of this thesis
- Source code of *libcbor*, including the following resources
  - Detailed build and installation instructions
  - Test suite
  - Examples
- Source code of *libcbor-rb*, including the following resources
  - Detailed build and installation instructions
  - Executable specification
  - Examples
  - A brief tutorial
- API documentation for *libcbor*
- API documentation for *libcbor-rb*
- User documentation for *libcbor*
- Source code of benchmarks, including sample data
- A copy of the RFC 7049 specification

Appendices

Appendix A

Benchmark data showcase and description

A.1 `citylots`

Size 181 MB

Content type Array of nested map with text, integers, decimals and booleans.

Representative use case Balanced, mixed inputs. Larger documents, configuration files, object graph dumps.

Source City of San Francisco government data (`https://data.sfgov.org/Geographic-Locations-and-Boundaries/City-Lots-Zipped-Shapefile-Format-/3vyz-qy9p?`)

```
"features": [
  {
    "geometry": {
      "coordinates": [
        [
          [-122.42200352825247, 37.80848009696725, 0],
          [-122.42207601332528, 37.808835019815085, 0]
        ]
      ],
    },
    "properties": {
      "BLKLOT": "0001001",
...
```

A.2 `numbers`

Size 4.4 MB

Content type A matrix of integers.

Representative use case Numbers-heavy inputs, graphics, 3D models, statistical data.

Source Generated.

```
[
  [
    37355,
    35787,
    37505,
...
```

A.3 `cards`

Size 43 MB

Content type Combination of maps and strings between 3 and 270 characters.

Representative use case Strings-heavy web applications, user input serialization.

Source MTG JSON (`http://mtgjson.com/`)

```
"cards": [
  {
    "artist": "Rebecca Guay",
    "colors": [
      "Green"
    ],
    "foreignNames": [
      {
        "language": "Chinese Traditional",
        "name": "\u8c50\u8863\u8db3\u98df"
...
```

A.4 glossary

Size 552 B

Content type A simple map 'object' with string keys.

Representative use case Web and interactive applications server communication. Messaging, lightweight synchronization.

Source The JSON specification (`http://json.org/`)

```
"ID": "SGML",
...
  "GlossDef": {
    "para": "A meta-markup language, ...",
    "GlossSeeAlso": ["GML", "XML"]
  },
  "GlossSee": "markup"
}
```

A.5 instruments

Size 216 KB

Content type Map with many simple values and small integers.

Representative use case Configuration files, machine generated data, database dumps, web applications, UI state.

Source Chad Austin's web benchmarks (`https://github.com/chadaustin/Web-Benchmarks`)

```
...
"default_filter_cutoff" : 0,
"default_filter_cutoff_enabled" : false,
"default_filter_mode" : 255,
"default_filter_resonance" : 0,
"default_filter_resonance_enabled" : false,
...
```

A.6 `blobs`

Size 65 MB

Content type Array of binary objects.

Representative use case Embedding resources, structured file transfer.

Source Generated. For JSON, the binary data are Base64-encoded

Appendix B

Memory profiles

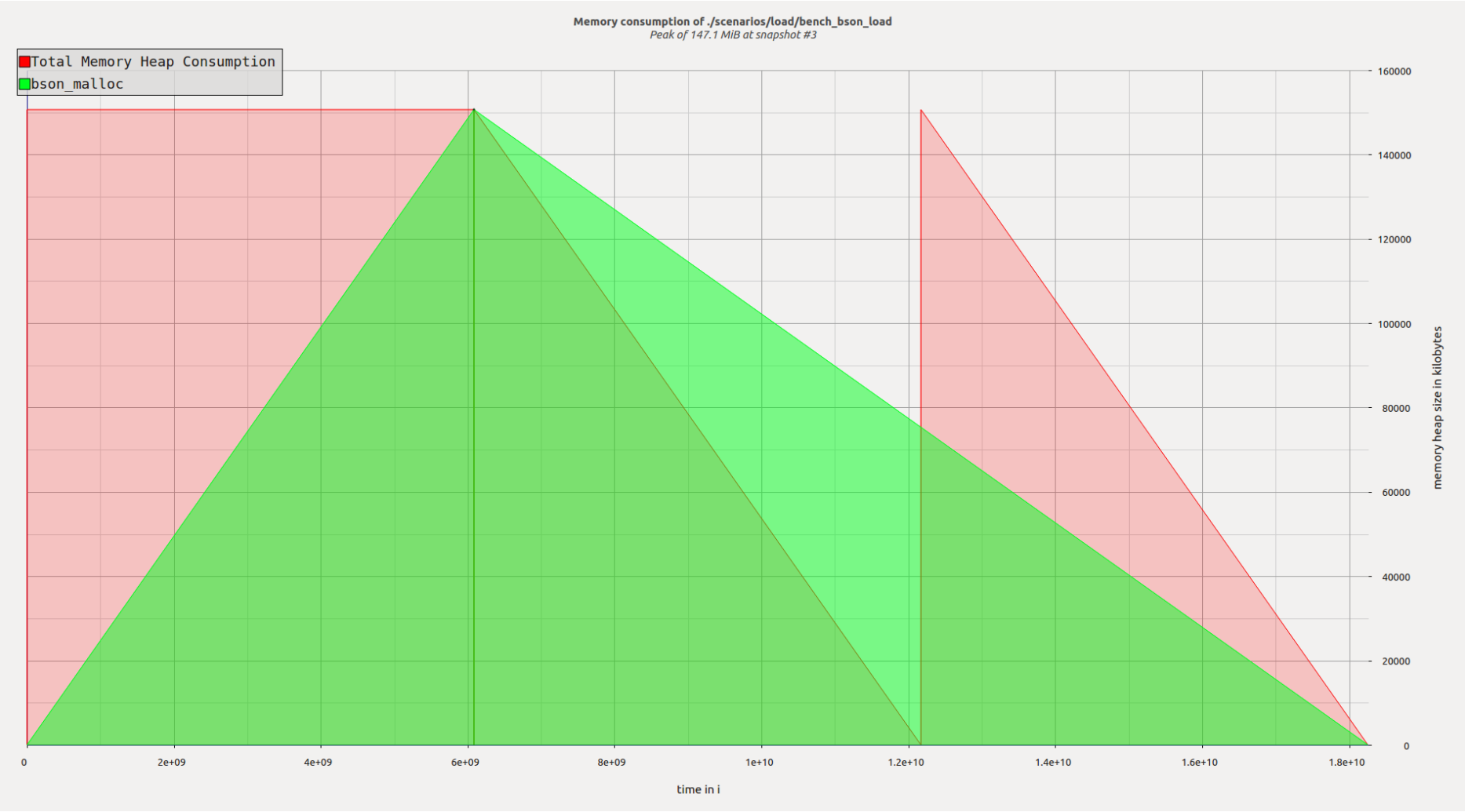


Figure B.1: *libbson*

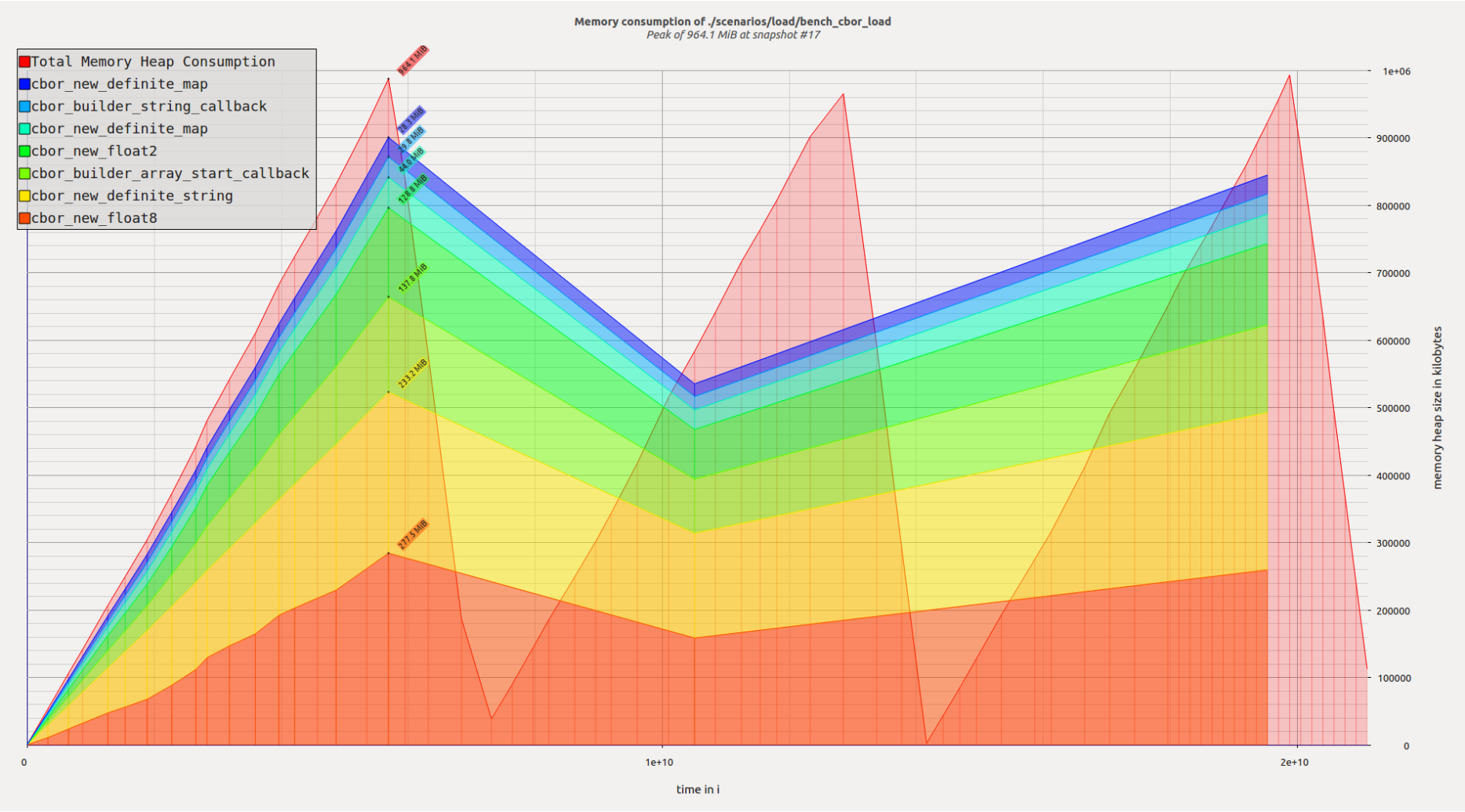


Figure B.2: *libcbor*

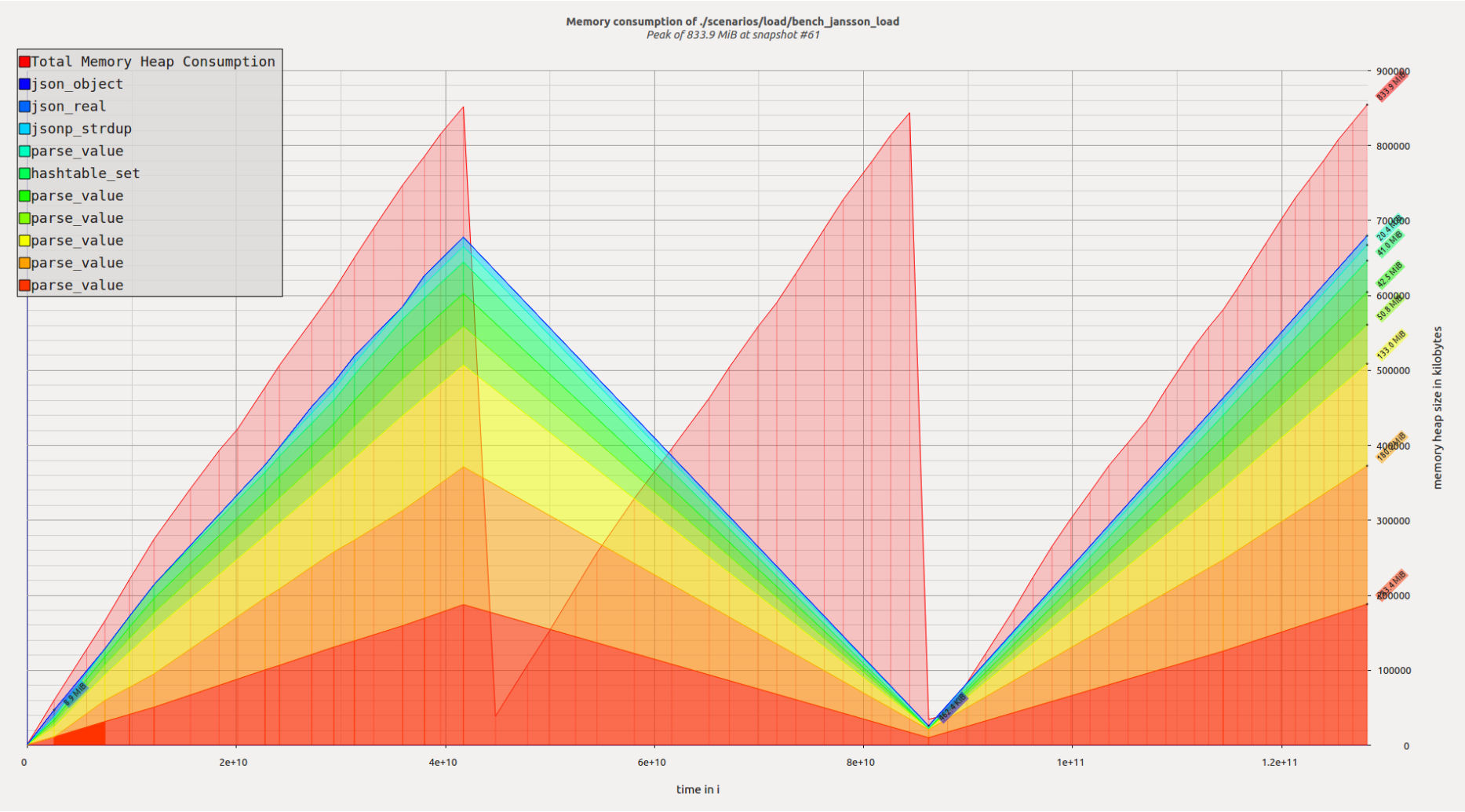


Figure B.3: Jansson

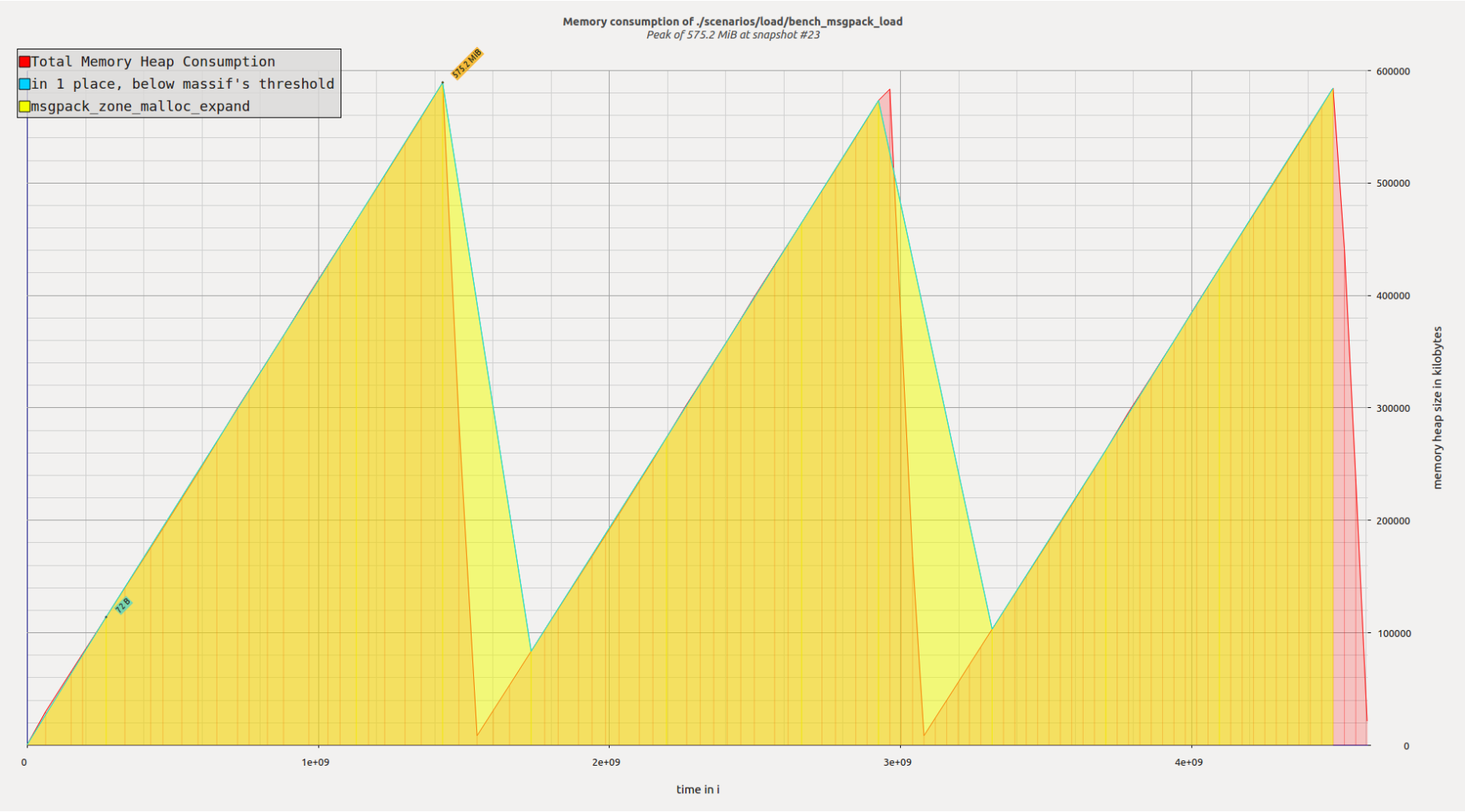


Figure B.4: *msgpack-c*

Appendix C

libcbor usage examples

```ruby
# Listens for connections, asynchronously receives data, replies with
# pretty-printed arrays (works for indefinite arrays with integers only for
# the sake of simplicity)
#
# Make sure to install EventMachine first (`$ gem install eventmachine`)
#
# Start with
# $ ./examples/network_streaming.rb
#
# Then send data from the example file using netcat or a similar tool:
# $ netcat localhost 9000 < examples/data/indef_array.cbor
#
# The file from the example contains the CBOR representation of
#   [_ [_ 1, 2], 3, [_ 4, [_ 5]]]
# Terminate with ^c

$LOAD_PATH.unshift(File.join(File.dirname(__FILE__), '..'))
require 'lib/libcbor'
require 'rubygems'
require 'eventmachine'

class CBORPrinter < EM::Connection
  def print(what)
    send_data('  ' * @nesting + what.to_s + "\n")
  end

  def initialize
    @nesting = 0
    @reader = CBOR::Streaming::BufferedDecoder.new(
      array_start: ->() { print '['; @nesting += 1 },
      integer: ->(val) { print val },
      break: ->() {
        @nesting -= 1; print ']'
        close_connection_after_writing if @nesting == 0
      }
    )
  end

  def receive_data(data)
    @reader << data
  end
end

EventMachine.run do
  Signal.trap('INT')  { EventMachine.stop }
  EventMachine.start_server('0.0.0.0', 9000, CBORPrinter)
end
```

Listing C.1: Using the streaming API in conjunction with network communication

```c
void usage()
{
  printf("Usage: streaming_parser [input file]\n");
  exit(1);
}

/*
 * Illustrates how one might skim through a map (which is assumed to have
 * string keys and values only), looking for the value of a specific key
 *
 * Use the examples/data/map.cbor input to test this.
 */

const char * key = "a secret key";
bool key_found = false;

void find_string(void * _ctx, cbor_data buffer, size_t len)
{
  if (key_found) {
    printf("Found the value: %*s\n", (int) len, buffer);
    key_found = false;
  } else if (len == strlen(key)) {
    key_found = (memcmp(key, buffer, len) == 0);
  }
}

int main(int argc, char * argv[])
{
  if (argc != 2)
    usage();
  FILE * f = fopen(argv[1], "rb");
  if (f == NULL)
    usage();
  fseek(f, 0, SEEK_END);
  size_t length = (size_t)ftell(f);
  fseek(f, 0, SEEK_SET);
  unsigned char * buffer = malloc(length);
  fread(buffer, length, 1, f);

  struct cbor_callbacks callbacks = cbor_empty_callbacks;
  struct cbor_decoder_result decode_result;
  size_t bytes_read = 0;
  callbacks.string = find_string;
```

Listing C.2: Using the streaming decoder to find just one value. No memory is allocated in the process.