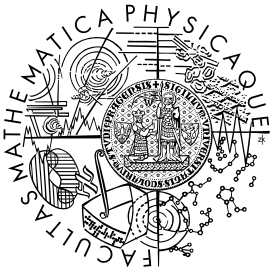

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

David Čepelík

Routing Information Exchange

Department of Applied Mathematics

Supervisor of the bachelor thesis:	Mgr. Martin Mareš, PhD.
Study programme:	Computer Science
Study branch:	IOI

Prague 2017

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 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

If I was to thank someone for helping me with this thesis, it would logically be my adviser, Mgr. Martin Mareš, PhD., who spent great deal of time and energy in the process. Thank you, Martin—as always, I very much appreciate your help.

But I feel I should thank other people, too, who—directly or indirectly—have been helping me for the long years it took me to finally arrive here:

- své rodině a přátelům za jejich lásku a neustálou podporu a za to, že se ke mně neotočili nikdy zády, zejména mým rodičům Evě a Pavlovi, Kláře a přátelům Ondrovi a Vojtěchovi,
- to Martin Mareš, for his constant support during my studies, for sharing his ideas and opinions on pretty much everything that crossed his mind and for letting me do the same, and of whom I think of as category (i) anyway,
- to the staff of the Faculty for their professional and friendly attitude to students, especially to doc. RNDr. Vít Jelínek, PhD., for his immense patience, support and great lectures on various topics in discrete mathematics,
- to the Student Affairs Department we may be really proud of, because they actually do care about us, which seems to be quite rare,

and I would like to end this non-exhaustive list by saying that

This thesis is dedicated
to good coffee and hard work,
neither of which is ever enough.

Title: Routing Information Exchange

Author: David Čepelík

Department: Department of Applied Mathematics

Supervisor: Mgr. Martin Mareš, PhD.,
Department of Applied Mathematics

Abstract: We present an efficient method for serialization of complex objects into a compact binary form based on the Concise Binary Object Representation (CBOR). We provide a high-performance, well-tested implementation of a data serialization library and implement it in the BIRD Internet Routing Daemon.

Keywords: BIRD, data serialization, CBOR

Contents

1 Introduction . . . 3
1.1 The BIRD Command Line Interface (CLI) . . . 3
1.2 The BIRD CLI Protocol . . . 4
1.3 The Encoding . . . 4
1.4 Evaluation of the Encoding . . . 5
1.4.1 Space Complexity . . . 5
1.4.2 Time Complexity . . . 6
1.4.3 Unstructured, ad-hoc format . . . 6
1.4.4 Development Effort . . . 6
1.4.5 Debugging Aid, Human-readable . . . 7
1.5 A Note About BIRD's I/O . . . 7
1.5.1 The `io_loop` . . . 7
1.5.2 BIRD control socket and `cli_connect` . . . 8
1.5.3 The event mechanism, `cli_event` and `cli_rx` hook . . . 8
1.5.4 `cli_get_command` and `cli_command` . . . 9
1.6 Performance Issues . . . 9
1.7 Summary . . . 10
2 CBOR . . . 11
2.1 Design Goals and Properties . . . 11
2.2 High-level Structure . . . 11
2.2.1 Major Types 0 through 6 . . . 11
2.2.2 Lower Bits . . . 12
2.3 Indefinite-length Encoding . . . 12
2.4 Items With Major Type 7 . . . 13
2.5 CBOR Diagnostic Notation . . . 13
2.6 Examples . . . 14
3 Problem Analysis . . . 16
3.1 Problem Statement . . . 16
3.2 Common Problems . . . 16
3.2.1 Version Negotiation . . . 16
3.2.2 Dependence On a Central Authority . . . 16
3.2.3 Dependence on Physical Topology of Data . . . 17
3.3 Requirements . . . 17
3.4 Possible Solutions . . . 18
3.4.1 Fixed Binary Scheme . . . 18
3.4.2 XML . . . 19
3.4.3 JSON . . . 20
3.4.4 Google Protocol Buffers . . . 20
3.4.5 Fixed CBOR Scheme . . . 21
3.4.6 NetBufs: a CBOR-based Protocol . . . 22
4 NetBufs . . . 23
4.1 Motivation . . . 23
4.2 Informal Description . . . 23
4.2.1 Basics of NetBufs Encoding . . . 23
4.2.2 Encoding With String Keys . . . 25

4.2.3 Introducing Groups . . . 26
4.2.4 Introducing Keys . . . 27
4.3 Formal Description . . . 28
4.3.1 CBOR Usage Rationale . . . 28
4.3.2 Full CBOR Compatibility . . . 28
4.3.3 Terminology . . . 29
4.3.4 Specification of the NetBufs Encoding . . . 29
4.3.5 Rationale And Comments . . . 29
5 Benchmarks . . . 31
5.1 Benchmarking Scheme . . . 31
5.2 Measurements . . . 31
5.3 Measurements Interpretation . . . 32
5.3.1 Fixed Binary/Fixed CBOR/NetBufs . . . 32
5.3.2 Bird/NetBufs . . . 33
5.3.3 NetBufs/Google Protocol Buffers . . . 33
5.3.4 BIRD/XML . . . 34
5.4 Conclusion . . . 34
6 Implementation . . . 35
6.1 Dependencies . . . 35
6.1.1 Building . . . 35
6.2 Interfaces . . . 35
6.2.1 Memory-management Interface . . . 35
6.2.2 I/O Interface . . . 35
6.3 CBOR Encoder and Decoder . . . 36
6.4 Serialization and Deserialization Routines . . . 37
6.5 Coroutines . . . 37
6.5.1 Coroutines and `<ucontext.h>` . . . 38
6.5.2 The I/O Problem . . . 38
6.5.3 Coroutines Solution . . . 39
6.5.4 Performance of `<ucontext.h>` . . . 39
6.6 BIRD Integration . . . 40
6.6.1 Limitations . . . 41
6.6.2 Protocol Replacement . . . 41
6.7 Benchmarking Code . . . 41
6.8 The `nbdiag` Diagnostic Utility . . . 42
6.8.1 Library Functions For Diagnostics . . . 43
6.9 Test Suite . . . 43
7.1 Limitations . . . 45
7.2 Configuration . . . 45
7.3 Sending/Receiving Data . . . 46
7.3.1 Sending API . . . 46
7.3.2 `while`/`switch` construct for receiving . . . 46
7.3.3 Receiving API . . . 47
7.4 Example . . . 47

Chapter 1
Introduction

The Bird Internet Routing Daemon (BIRD) was created as a school project at the Faculty of Mathematics and Physics, Charles University in Prague and first released in 2000. It implements several routing protocols, namely the Border Gateway Protocol (BGP), Routing Information Protocol (RIP), Open Shortest Path First (OSPF) protocol and the Babel protocol, besides others.

All paths in this thesis are relative to the `netbufs` root directory of the project as submitted with this work, unless specified otherwise. By `$BIRD_ROOT` we mean the root directory of a BIRD distribution.

BIRD does not carry out the actual routing of packets in computer networks. Instead, BIRD exchanges routing information with other routing daemons and configures the routing tables of the operating system's kernel, which handles the routing. Or it can be used as a route server, to centralize peerings between BGP speakers at Network Access Points (NAPs).

BIRD features powerful route filtering mechanism. Administrators may use filters to specify which routes from which peers should be accepted, refused or accepted but modified; which routes shall be exported to which peers, etc.

As a route server, it is used in several Internet exchanges, such as the London Internet Exchange (LINX), London Network Access Point (LONAP), Deutscher Commercial Internet Exchange (DE-CIX) and Moscow Internet Exchange (MSK-IX).

1.1 The BIRD Command Line Interface (CLI)

To command a running BIRD instance, administrators may use the BIRD CLI Client program which ships with BIRD. This utility establishes a communication channel with the BIRD daemon process through a Unix-domain control socket and allows the them to issue various commands to perform basic administrative tasks, such as:

- displaying the basic status of the daemon,
- printing BIRD's internal route database, applying filters as needed, printing route statistics and resource usage information,
- monitoring logs by requesting that logging output is copied into the CLI,
- requesting BIRD to reload the configuration file,
- terminating or restarting BIRD and much more.

Besides being used by the administrators, the CLI is used by various scripts and other automated tools which depend on it to obtain routing information from BIRD which they further process to do their job.

To provide the CLI service, BIRD needs to (i) define a communication protocol between the clients and the server, and (ii) define the encoding scheme which is used to serialize and deserialize data.

In this chapter, we will describe both the protocol and the encoding scheme and identify some problems they have.

1.2 The BIRD CLI Protocol

BIRD's CLI uses a very simple text-based protocol similar to SMTP of FTP.

In this protocol, requests are commands encoded as a single line of text, while replies are sequences of lines starting with a four-digit code followed by either a space to indicate 'end-of-reply', or with a minus sign to indicate that another line will follow. (Or with a plus sign to indicate asynchronous answers used when logging output is echoed into the CLI.) Reply codes starting with 0 stand for 'action successfully completed' messages, 1 means 'table entry', 8 means 'runtime error' and 9 means 'syntax error' [2]. If a reply line has the same code as the previous one and it's a continuation line, the whole prefix can be replaced by a single white space character. Figure 1.2.1 shows an example session created using the Netcat tool.

```
0001 BIRD 1.6.3 ready.
show status
1000-BIRD 1.6.3
1011-Router ID is 192.168.1.4
 Current server time is 2017-07-21 06:21:31
 Last reboot on 2017-07-21 06:21:25
 Last reconfiguration on 2017-07-21 06:21:25
0013 Daemon is up and running
show memory
1018-BIRD memory usage
 Routing tables:   9136  B
 Route attributes: 5456  B
 ROA tables:        192  B
 Protocols:         840  B
 Total:              39 kB
```

Figure 1.2.1 Example BIRD session.

The Protocol itself is not a source of any problems. It may even be useful to have a text-based protocol: this way, one may use common tools such as Netcat to talk to BIRD.

However, to not have to worry about the Protocol, many tools that talk to BIRD parse the output of the CLI client utility (which can be run non-interactively). This way, the tool processes only the data contained in the answer, the status information is stripped.

Despite its simplicity, the Protocol is a barrier for further development and it would be useful to have a library which the tools could use directly, instead of having to execute an external binary and to worry about escaping of arguments.

1.3 The Encoding

BIRD's response to the CLI client often contains complex data structures, such as routing tables. To send this data to the client, BIRD encodes them as plain-text strings.

To clarify terminology, in this work we refer to the process of translating in-memory structures into their text representation as 'encoding' or 'serialization', and to the inverse process of parsing the text back into in-memory data structures as 'decoding' or 'deserialization'.

If we omit the very protocol for a while and extract the data itself, we will see that routing table entries are encoded in the following fashion, which leads to

issues discussed later. Figure 1.3.1 below shows a single encoded entry of a BGP routing table as encoded by BIRD and sent to the client.

```
188.0.188.0/24     via 10.0.0.1 on eth0 [uplink 13:47:35 from 217.31.205.209] * ...
    Type: BGP unicast univ
    BGP.origin: IGP
    BGP.as_path: 25192 6939 12389 49724 49724 49724
    BGP.next_hop: 217.31.205.209
    BGP.local_pref: 100
    BGP.community: (25192,1111)
```

Figure 1.3.1 Excerpt from BGP table dump.

We see in Figure 1.3.1 that the encoding of a single route consists of a "header line" which describes basic properties of the route, like the target network (`188.0.188.0/24`), the gateway to be used to get to that network (`10.0.0.1`), the interface on which the gateway is reachable (`eth0`) and various status data (partially removed for brevity).

Following the header line, there are consecutive lines of "route attributes" which are indented by four spaces and which are specific to the BGP protocol in this case, such as where the route originated (`IGP` standing for Interior Gateway Protocol), which autonomous systems (ASs) this route has traveled to arrive here (the sequence starting `25192 6939`...), etc., and whose semantics is largely unimportant to the problem itself.

1.4 Evaluation of the Encoding

In Section 1.3, we have shown how BIRD encodes data. This approach has some advantages and some deficiencies, which will be discussed in this section. The problems will be addressed later when we define new data exchange format for BIRD in Chapter 3.

1.4.1 Space Complexity

As we have seen in Figure 1.3.1, BIRD uses only the first 128 symbols of the ASCII range to produce output. To write the number n in decimal, at least $\log_{10} n + 1$ of symbols `0`, `1`, ..., `9` is required, each encoded in a single byte. On the other hand, to encode n in binary, one would only need $(\log_2 n + 1)/8$ bytes. This is 5 bytes compared to 2 bytes to encode 21167.

To encode an IPv4 addresses, the scheme uses up to three bytes for each of the four components, with three dots in between. This way, an IP address is up to 15 bytes long, compared to 4 bytes which are always sufficient in binary.

BIRD also encodes the names of all attributes, such as `BGP.origin`, repeatedly. If attribute names were numbers instead, 256 attribute names would be single byte, and 65536 attribute names would fit into two bytes. Also, the dump contains "keywords" such as `uplink` and visual features such as `[` or `*` and whitespace which are only added for legibility.

These concerns are of significant practical importance. One of our test inputs is a routing table dump with approximately $6 \cdot 10^5$ records, whose size is 188 MiB.

1.4.2 Time Complexity

Time is wasted on encoding and decoding of data in this fashion. First, with more data, more time is spent transferring it—at the very least, it will take more time to copy the data to/from a socket's RX/TX buffer.

While decoding of decimal numbers is quite fast (using Horner's scheme, for example), encoding is slower. BIRD uses `printf`-like functions from the Standard C library to produce output.

For single-byte numbers, one could use a precompiled lookup table. This would make the encoding of IPv4 addresses much faster—their components are single-byte numbers. But for two-byte numbers, also frequently present in the encoding, this approach would be infeasible, as the tables would be too large to be practical. Therefore, for larger numbers, one would have to use more advanced algorithms.

Algorithms exist which encode numbers much faster than `printf`, but still much slower than most binary encodings, which rely on clever replacement of division by multiplication and modular arithmetic, but their description falls out of scope of this work. Our point is that, even though such clever solutions exist, for performance reasons, printing of numbers in decimal is better avoided at all.

Time is wasted on encoding of numbers and we will measure precisely how much in Chapter 5.

1.4.3 Unstructured, ad-hoc format

The format used by BIRD is described very informally. There is no documentation which would describe the format in detail. The 'shape' of data produced by BIRD is defined by the source code which happens to produce it.

This is a problem as serious—or in some cases, worse than—the space and time penalty imposed by the encoding. It forces developers of tools to depend on a fragile format, which changes slightly as BIRD undergoes development, and to reinvent parsers for the format again and again.

1.4.4 Development Effort

Little effort is needed to create a human-readable representation of a complex data structure, as can be seen in `rt_show_new` (file `$BIRD_ROOT/nest/rt-show.c`). The serialization code is straightforward and self-descriptive. (It is only complicated by route filtering and by complex internal representation of routes in BIRD.)

It is not very difficult to parse data encoded in this format, but the lack of formal description makes it rather tedious. We have, for illustration and for the purposes of benchmarking, developed a parser of routing table dumps, which can be found in `benchmark/src/parser.c`.

Also, to encode data in this format, one does not need any external dependencies except the Standard C Library, which simplifies deployment.

1.4.5 Debugging Aid, Human-readable

The biggest benefit of encoding data in plain-text is that it is human-readable and if it is to be processed by a human, the CLI Client utility does not need to do any other formatting—the data can be displayed as-is.

Furthermore, one usually wants to have a way of "dumping" a program's data structures into their human-readable representation for the purposes of debugging. Using human-readable text as serialization output circumvents the need for such specialized code; one can often use the serialized data itself.

On the other hand, debugging output often has to contain additional data which reflect internal state of the program, which would be of no interest to CLI users, and a specialized code for debugging often needs to be written anyway. An example of this is the `dump` function in `$BIRD_ROOT/nest/rt-fib.c`.

1.5 A Note About BIRD's I/O

BIRD was created in the early 2000's, as we said earlier. At the time of its creation, multi-core and multi-processor systems were rather rare, and BIRD was designed to achieve high single-thread performance and throughput.

This has affected BIRD's design significantly. The most important outcome of this decision for our work is that BIRD does not process clients' requests' in parallel, but sequentially. Also, when talking to a client, BIRD is incapable of serving other daemons and clients.

This means that I/O actions which take a lot of time to complete must be divided into smaller work units, each of which takes only a portion of the total processing time to complete. Otherwise, BIRD would block too long, causing hang-ups in other sessions, which could time out.

In the rest of this section, we shall explain how BIRD handles I/O internally; especially how CLI client's commands are handled. Much later in Section 6.6, we will alter this mechanism to use the new protocol and encoding.

1.5.1 The `io_loop`

For historical and performance reasons, and in some cases out of necessity, some of BIRD's code is system-dependent. The system-dependent functions reside in a module called `sysdep`, which provides functions to the rest of the code through a clean interface. `sysdep` provides implementation for Linux (`sysdep/linux`) and BSD (`sysdep/bsd`), with common code for all Unix-like systems (`sysdep/unix`) [2]. It could be, however, implemented on other operating systems if needed.

The file `sysdep/unix/main.c` contains BIRD's entry point, which—among other things—runs BIRD's `io_loop` from `sysdep/unix/io.c`. This is the top-level loop which executes the whole time a BIRD instance is running. The process is depicted in Figure 1.5.1.

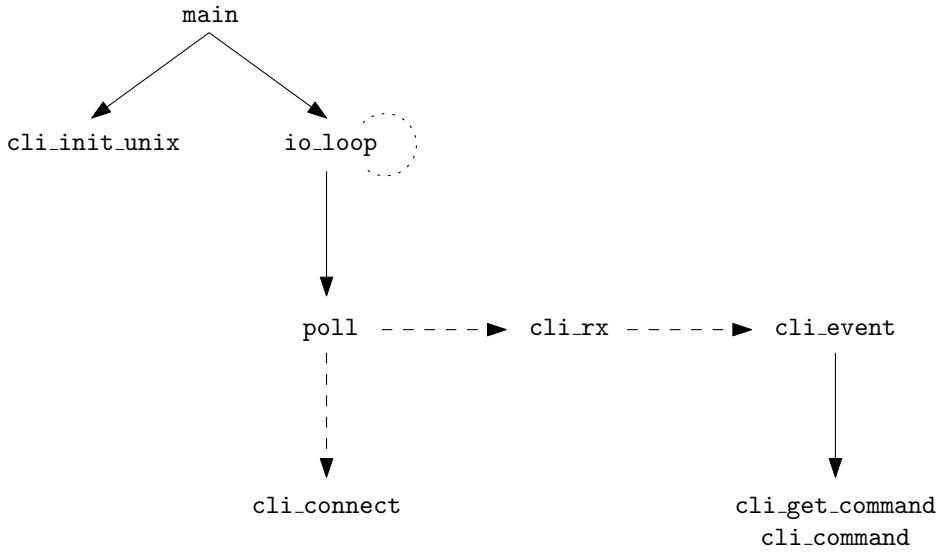

Figure 1.5.1 CLI communication and the `io_loop`.

The `io_loop` uses the `poll` function on all socket file descriptors that BIRD uses. BIRD provides a thin wrapper around BSD-style sockets, which allows one to define the function pointers `rx_hook` and `tx_hook` for "data was received" and "data may be transmitted" socket events, respectively. The `io_loop` calls these handlers upon notification from `poll`.

1.5.2 BIRD control socket and `cli_connect`

One of the sockets which are managed by `poll` in the `io_loop` is the Unix-domain BIRD control socket used by the CLI client. This socket is wrapped in a `struct birdsock`, too, whose `rx_hook` is set to `cli_connect`.

The new client connection is assigned a new socket (by calling `accept` on the control socket), which is passed to the `cli_connect` hook. This function configures the socket's `rx_hook` and `tx_hook` to `cli_rx` and `cli_tx`, respectively, and establishes a new context for the new CLI connection, `struct cli`.

It is this `rx_hook` and `tx_hook` that handle the incoming client requests and take care of sending the responses back.

1.5.3 The event mechanism, `cli_event` and `cli_rx` hook

Besides `poll`-ing of the socket file descriptors to initiate processing of clients' and other daemons' requests, BIRD uses events to schedule work. When BIRD needs to perform a time-consuming action which would block the `io_loop` for too long, it will perform only a part of the action and schedule the rest as an event to occur in the future.

To illustrate this, imagine that you issue the `show route` command on the CLI, which will result in BIRD having to iterate over and transfer all the routes in its database. Instead of doing that directly, BIRD will transfer a small block of routes upon receiving your request and will schedule an event in the future, whose handling will consist of transmitting of the rest. It will then continue in this fashion, sending small portions of data successively, until there's nothing left to be sent. This is a weaker form of the concept of "continuations".

The `cli_rx` hook does not directly process the client's request. Instead, BIRD will delay the processing and schedule an event.[1] When this event occurs, the `cli_event` handler gets called which will either continue sending response back to the client if the CLI's TX buffer has data ready, or it will continue the processing of the previous request (through the `cont` CLI hook) if it wasn't finished yet, or it will initiate the parsing and processing of the current request.

To illustrate some problems which are not apparent immediately: by the time the `cli_event` handler got called, the command's text may not have yet been fully received. So this invocation of `cli_event` will merely copy a piece of a command from the socket's `rbuf` to the CLI's `rx_buf`, returning control to the `io_loop` and depending on the next invocation of `cli_event` to (possibly) finish the processing of the request.

1.5.4 `cli_get_command` and `cli_command`

The `cli_get_command`, which is called by `cli_event`, is responsible for copying of the command's text from the socket's `rbuf` to CLI's `rx_buf`. It copies the text character by character until it hits a line feed or until the limit on command's length is reached. If multiple commands are received on the socket separated by a line feed character, the `rx_aux` pointer of `struct cli` is set so that next command's processing starts at the correct offset in the `rx_buf`.

`cli_event` then calls `cli_command`, which finally triggers the actual processing of the command. The individual commands' handlers are set in the Bison config files, along with the commands' syntax rules, and are defined in relevant places of the code.

That means that the printing of routing table attributes to the CLI, for example, is located in `$BIRD_ROOT/nest/rt-attr.c` along with other functions. Handlers of commands which do not fit into any module logically are located in `nest/cmds.c`.

To send response to the client, the command handlers call `cli_printf` function from `$BIRD_ROOT/lib/cli.c` which applies `printf`-like formatting to the message and prefixes output lines with status information. (Or the `cli_msg` macro to reply to current CLI session and which expands into `cli_printf` call.)

1.6 Performance Issues

In large setups with tens of concurrent BGP sessions, BIRD's single-threaded design is a limiting factor. This will likely result in the authors' rewriting of portions of BIRD to allow for multi-threading. But until then, we have to cope with the limitations BIRD's single threaded design brings. In the new implementation of CLI protocol and encoding in BIRD, therefore, we will have to use coroutines to avoid the problems created by BIRD's event-based approach to I/O processing. This will be explained in Chapter 6 thoroughly.

The encoding scheme which is time-consuming to decode also creates problems for various scripts that parse routing table dumps, and which take too long to complete as a result. This gets worse with periodically running scripts.

[1] It could start processing the request straight on, but scheduling an event instead guarantees that all request processing is invoked from the event handler—the initial processing as well as continuations.

1.7 Summary

In this chapter, we have described the BIRD protocol and encoding, which are employed in the CLI. We have described some of the problems the encoding scheme suffers. We have pointed out that these problems are important practically and limit interoperability of BIRD with external tools.

The I/O mechanism of BIRD was explained at a level necessary to understand how it was modified in Section 6.6.

Chapter 2
CBOR

Our quest for a new encoding scheme for BIRD will begin by describing a scheme that was highly influential to our work. Familiarity with CBOR will be beneficial to the reader in later chapters, which describe an encoding based on CBOR.

CBOR data format is described in full detail in RFC 7049 [1]. Here, only those features of the format which are essential for further discussion are presented.

2.1 Design Goals and Properties

CBOR was designed to allow for extremely small code size, fairly small message size and extensibility without the need for version negotiation.

The 'code size' above is really the amount of code one has to write to encode and decode CBOR. We have noted during development that the protocol is really programmer-friendly.

We have observed that the format is well-designed, easy to both encode and to decode. High-performance implementation of an encoder and a decoder is relatively easy to write, provided one has experience with advanced memory management and buffering. We have come to favor CBOR for its simplicity and elegance, too.

In the following text, "byte" means an octet, or eight bits. Where the order of bytes matters, CBOR always encodes data in network (big-endian) byte order, that is, writing the most significant byte first.

2.2 High-level Structure

The smallest encodable units are called "data items" in CBOR. Every data item has a defined "major type".

Every data item consists of at least one byte, which we will call the 'header' of the item. Three upper bits of the header carry information about the major type, the remaining five bits are interpreted depending on the major type.

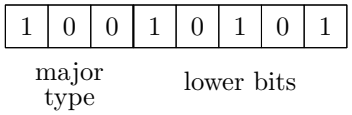

Figure 2.2.1 Header of a CBOR data item.

2.2.1 Major Types 0 through 6

There are 8 major types in CBOR: unsigned and negative integers, byte strings, text strings, arrays, maps, tags, and a special major type 7, which will be discussed later. (Below, numbers in parentheses are the actual major types.)

- (0) Unsigned integers lie in the range[1] $0, ..., 2^{64} - 1$.

[1] With the use of semantic tagging, however, CBOR defines encoding of "bignums".

- (1) Negative integers lie in the range $-2^{64}, \ldots, -1$.
- (2) Byte strings are strings of arbitrary bytes.
- (3) Text strings are byte strings, the difference is purely semantic.
- (4) Arrays are sequences of encoded items, possibly other arrays or maps.
- (5) Maps are sequences of encoded key-value pairs. Keys and values are arbitrary.
- (6) A tag conveys information about following data item, possibly simplifying the process of decoding and/or interpretation of encoded data.

Byte strings, text strings, arrays and maps may be empty.

2.2.2 Lower Bits

For major types 0 through 6, the 5 lower bits are interpreted as either a direct unsigned integer n (when $n \leq 23$), or they carry information of how many following bytes (1, 2, 4 or 8) should be decoded to obtain n. Figure 2.2.2 shows a header for unsigned integer (major type 0) with lower bits set to 25_{10} to indicate that two bytes should be decoded to obtain n. Then, the two bytes follow.

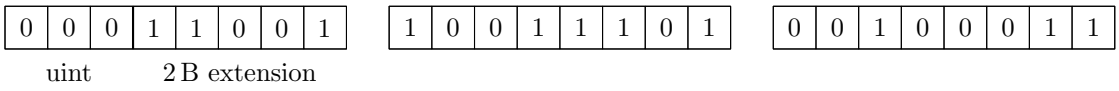

Figure 2.2.2 An unsigned integer encoded as a header and 2 B payload.

The interpretation of n is then as follows:

- For unsigned integers, n is the unsigned integer itself.
- For negative integers, $-(1+n)$ is the encoded negative integer[1].
- For byte and text strings, n is the length of the string in bytes.
- For arrays, n is the number of encoded items that follow.
- For maps, n is the number of encoded key-value pairs that follow.

To summarize, (i) the n may be encoded in the header of the data item, or in up to eight following bytes, and (ii) n can be the data itself, or it can be the length of a byte or text stream, or the number of items in an array, or the number of key-value pairs in a map. Any combination of (i) and (ii) is possible.

This way, 'small' integers in the range $-24, \ldots, 23$ fit into the header and require no further bytes to encode. Therefore, header of a string which is at most 23 characters, a byte stream which is at most 23 bytes, an array with at most 23 items or a map witch at most 23 key-value pairs fits into a single byte, too.

2.3 Indefinite-length Encoding

In 2.2, the meaning of the integer quantity n associated with a data item was explained. However, CBOR permits so called "indefinite-length" encoding for byte strings, text strings, maps and arrays, in which case the n can be omitted.

To signalize that an indefinite-length encoding is in use for a data item, the appropriate major type is expressed in the three most significant bits of the first byte of the encoding and the remaining five bits are set all high.

[1] This simple but clever formula encodes small negative numbers as small unsigned integers. That is useful when combined with variable encoding of integers to conserve space.

To signalize the end of the most recently encoded indefinite-length item, a "break" code is written. This purely syntactical major type 7 construct has no other meaning than terminating an indefinite-length item.

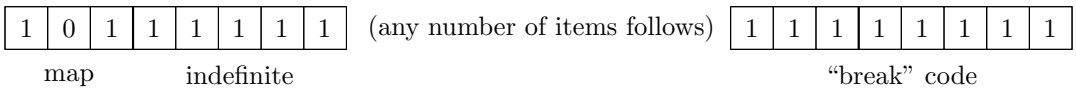

Figure 2.3.1 Indefinite-length map (header and "break" code).

All indefinite-length items may be empty. Indefinite length arrays and maps may contain other (possibly indefinite-length) data items, including indefinite-length arrays and maps. Indefinite-length byte and text strings are only allowed to contain fixed-length byte and text strings, respectively[1]; semantically, the value of an indefinite-length string is the concatenation of all the fixed-length strings it contains.

Indefinite-length items are useful when the amount of data to be serialized is not known at the time of the encoding and output buffering is to be avoided.

2.4 Items With Major Type 7

Items with major type 7 are used for so-called "simple values", floating-point numbers and the "break" code, the use of which was illustrated in Section 2.3.

The lower five bits encode an unsigned integer n, which is interpreted as follows:

- If n is in the range $0, \ldots, 23$, then n is a simple value n.
- If $n = 24$, the simple value is contained in a single byte which follows.
- If $n = 25$, then IEEE 754 half-precision float follows in two bytes.
- If $n = 26$, then IEEE 754 single-precision float follows in four bytes.
- If $n = 27$, then IEEE 754 double-precision float follows in eight bytes.
- If $n = 31$, then this item is the "break" code.

We shall not distract ourselves with floats, because we will not use them in this work. Values $28_{10}, 29_{10}$ and 30_{10} are unused.

The "simple values" are used for "values with no content" such as 'true', 'false', 'null' and 'undefined' constants. Simple values are registered by IANA, and the $24, \ldots, 31$ range of simple values is reserved for further use.

2.5 CBOR Diagnostic Notation

Throughout this work, we will often rely on CBOR Diagnostic Notation to provide a convenient way of expressing CBOR data items in human-readable form. The diagnostic notation is only described informally by the RFC; no formal description of the format is given, as it is not meant to be parsed—it is used solely for the purposes of debugging and documentation.

The notation borrows the JSON syntax for numbers (integer and floating point), True (`true`), False (`false`), Null (`null`), UTF-8 strings, arrays, and maps

[1] This is another clever restriction. If indefinite-length byte string could contain an indefinite-length byte string, no simple way would exist to check for the `0xFF` "break byte". And the length of byte strings stored in memory has to be known in advance, anyway.

(maps are called objects in JSON; the diagnostic notation extends JSON here by allowing any data item in the key position). Undefined is written `undefined` as in JavaScript.

```
[],
[[]],
[_
    [_
        [_
        ],
    ],
],
[
    1,
    [_
        2,
        3,
        [_
            "Hello World!",
            256,
            "BCDEFGHI",
        ],
    ],
    4
]
```

Figure 2.5.1 CBOR diagnostic notation.

Being informal, the diagnostic notation is best illustrated by example. Figure 2.5.1 shows the diagnostic notation of a CBOR stream contained in one of the CBOR test files, `tests/cbor/local/array/in`. The diagnostic notation shows:

- (Empty) array of size 0.
- Array of size 1 which contains an (empty) array of size 0.
- Indefinite-length array, which contains another indefinite-length array, which contains another indefinite-length array, which is empty.
- Array of size 3, which contains:
    - The integer 1.
    - An indefinite-length array, which contains:
        - The integers 2 and 3.
        - An indefinite-length array containing the string `Hello World`, the integer 256 and the string `BCDEFGHI`.
    - The integer 4.

This hopefully introduces CBOR diagnostic notation in enough detail to be useful to the reader.

2.6 Examples

In this short section, we will without further comments provide several examples of data and their CBOR equivalents to illustrate the concepts we have presented before. The examples will be given first in CBOR diagnostic notation and then in hex code.

CBOR diag notation	Bytes
`[]`	`0x80`
`11`	`0x0B`
`"Hello World!"`	`0x6C48656C6C...`
`{_1:{_2:{3:{_}}}}`	`0xBF01BF02A103BFFFFFFF`
23767	`0x197FFF`
-256	`0x38FF`
`true`	`0xF5`

Table 2.6.1 Examples of the CBOR encoding.

If the reader would like to try more complicated examples, which do not fit into the table, he can try to use the `nbdiag` tool described in Section 6.8 to dump the test vectors contained in `tests/cbor`:

```
xxd -p -r tests/cbor/.../in | ./nbdiag -123
```

Chapter 3

Problem Analysis

3.1 Problem Statement

Our goal in this work is to solve the problems presented in Chapter 1 by finding a new approach to encoding for BIRD CLI. The encoding may be invented, or an existing solution may be adopted. We will implement the solution in BIRD as a proof-of-concept. The choice of solution should be backed-up by benchmarks.

3.2 Common Problems

We have already described several problems that the text-based encoding suffers in Chapter 1. There are, however, other problems common to many encodings.

3.2.1 Version Negotiation

Version negotiation is actually a solution to the problem caused by two parties exchanging data, while one uses different version of the encoding format than the other. As software develops, message[1] formats develop, too—and different versions of the same software support (slightly) different message formats.

To be able to communicate, the parties start by "negotiating" about the version of the encoding format used, which means that they agree upon a version supported by both. Where version negotiation is used, the code which serializes and deserializes data has to behave conditionally, depending on negotiated version, and often looks like this:

```
if (version == 1) {
    ...
}
else {
    if (version > 2) {
        if (version != 3) {
            ...
        }

        ...
    }
}
```

This is a maintenance problem. So version negotiation is a problem after all.

3.2.2 Dependence On a Central Authority

When two parties exchange data, it is often required to prefix some items with a 'key', 'name' or 'ID' to give it a meaning. We have seen this already: in CBOR, simple values are numbers whose meaning is defined by the RFC.

But who is responsible for assignment of keys? Authorities such as IANA exist which handle this problem by centralizing the assignment of keys/names/IDs and

[1] Encoded objects are sometimes called "messages".

are used by most protocols in the IT industry, including CBOR (for simple values and tags), Address Resolution Protocol (ARP), BGP, etc.

Having an authority solves the problem, but brings bureaucratic overhead. IDs are assigned upon receiving a formal request, which slows development down. And someone has to run the authority, too.

3.2.3 Dependence on Physical Topology of Data

Some solutions assume that the internal representation of data is "isomorphic" the its representation on wire. This is often case with solutions which generate code and make assumptions about the structures which contain the data.

In many cases, especially in applications where performance is not critical, this can be acceptable or even beneficial. But in high-performance or memory-critical applications, the programmer has to be in charge of fine-tuning of the data structures of the program for desired properties. Not surprisingly, this is true for BIRD especially, which tries to maximize performance and minimize memory footprint.

3.3 Requirements

In this section, we would like to present some of the features the ideal encoding scheme would have. Our requirements naturally contradict each other; they are intuitive ideas rather than precise constraints.

Extensibility

BIRD is still in active development and the format of the messages will most likely change. The encoding should allow for development and facilitate change. Encoding produced by newer encoders should be decodable by older decoders and encoding produced by older encoders should be decodable by newer decoders under reasonable assumptions.

But we would also like to avoid version negotiation if possible, and we do not want to depend on a central authority to assign any identifiers.

Besides backward- and forward-compatibility, we would like the encoding to be "cross-compatible", meaning that two parties can directly decode data which they both know, and may decode the rest into generic data structures and examine or encode them later.

Expressive Power

The scheme has to be capable of transmitting complex objects such as routing tables, network topologies and configuration options without problems. We expect that the encoding and decoding functions will be provided by a library, and its API should handle these objects naturally.

Performance

We strive for as fast encoding and decoding as possible. Performance is one of the key properties that sets BIRD apart from its competitors and it shall not be affected negatively.

Compactness

The scheme needs to be compact. Our goal is to conserve storage and bandwidth and to facilitate end-user error reporting by allowing the users to enclose data with their bug reports easily. Compact scheme will also result in better cache behavior.

Smooth Integration and Compatibility

The code has to integrate well with existing BIRD code. New code shall be employed gradually as refactoring steps. This also means that we are obliged to use BIRD's internal memory management routines and other primitives rather than implementing our own. The text output of BIRD's CLI client utility must not be altered to ensure smooth operation of tools depending on it.

Generality

The use of the solution must not be limited to BIRD. It should be a good solution for many similar scenarios.

3.4 Possible Solutions

Many protocols and schemes have been devised by various authors to achieve similar goals: Extensible Markup Language (XML), JavaScript Object Notation (JSON), Concise Binary Object Representation (CBOR), External Data Representation (XDR) and Google Protocol Buffers to name a few.

We have decided we wanted to benchmark some of them on real data to obtain precise time and space measurements.[1] Also, we have chosen to design a protocol on our own and see how well it compares to established solutions.

Below, several schemes are presented briefly, including our own solution presented in section 3.4.6.

Most of the figures presented in the following sections were created by the `nbdiag` tool described in Section 6.8 and have been reasonably simplified and/or shortened.

3.4.1 Fixed Binary Scheme

In this scheme, data is encoded by simply appending each field's binary representation to a binary stream in a predetermined fixed order. The decoder and encoder have to agree on the order of the fields. The binary representation of individual fields may be chosen as close to common native representation as possible to be efficient.

If optional fields are supported, either each optional field has to be prefixed with a boolean value indicating whether it is present, or a bitmask has to be transferred which explicitly marks fields as (non-)present.

This scheme would be largely impractical to use in production as it is very fragile. Any change in the order of fields or even minor difference in their type

[1] Time and space taken by the encoding are the only parameters which can be measured precisely.

would result in incompatibilities across versions.[1] The scheme is hard to debug as Figure 3.4.1 implies. The protocol would definitely require version negotiation.

```
00000000: 0000 0000 0000 0005 312e 352e 3000 0000  ........1.5.0...
00000010: 1800 bc00 bc01 0000 0a00 0000 0000 0000  ................
00000020: 0465 7468 3000 0000 000d 0000 0000 2f00  .eth0........../.
00000030: 0000 0023 01d1 cd1f d901 0000 c23c 0000  ...#.........<..
00000040: 0000 0000 0000 0007 0000 0000 0000 0005  ................
```

Figure 3.4.1 Hex dump of a stream encoded using Fixed Binary Scheme (obtained by running the `xxd` tool on binary data).

We have decided to include this scheme in the experimental part—despite its many inherent flaws—because we assume it will be the fastest one both to encode and to decode, providing a lower bound on time complexity.

This scheme will likely waste some space on integers. Consider the BGP Autonomous System (AS) numbers. These 8-byte unsigned integers are present in BIRD's BGP routing tables and lie in the range $0 \ldots 2^{64} - 1$. In the Fixed Binary Scheme, all AS numbers are encoded as 8-byte integers, even though less than 4 bytes would often suffice to represent the value (cf. with CBOR in Chapter 2, which uses variable-length encoding of integers).

3.4.2 XML

In many people's opinion, the Extensible Markup Language (XML) is the industry's de-facto standard format for data serialization and exchange. It is somewhat human-readable and libraries exist for many programming languages and platforms. Figure 3.4.2 shows routing table encoded in XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<rt version="1.5.0">
  <rte netmask="24" netaddr="188.0.188.0" gwaddr="10.0.0.1" ifname="eth0" ...>
    <type_flags bgp="true" unicast="true" univ="true" static="false"/>
    <attrs>
      <bgp_origin>0</bgp_origin>
      <bgp_as_path>
        <as_no>25192</as_no>
        <as_no>6939</as_no>
        <as_no>12389</as_no>
        <as_no>49724</as_no>
      </bgp_as_path>
      <bgp_next_hop>217.31.205.209</bgp_next_hop>
      <bgp_local_pref>100</bgp_local_pref>
      <bgp_community>
        <cflag>
          <flag>25192</flag>
          <as_no>1111</as_no>
        </cflag>
      </bgp_community>
    </attrs>
  </rte>
  ...
</rt>
```

Figure 3.4.2 Routing information expressed using XML.

[1] This would mean that decoding an 8-byte integer where a 4-byte integer was encoded would lead to the corruption of rest of the stream and very likely to a crash at the end-of-file, if not sooner.

On the other hand, it is quite rare to find a well-written implementation of an XML parser, which would correctly decode the complicated standard.

By admitting this scheme to the experimental part, we would like to demonstrate that it is not a good fit for BIRD. We suspect that XML will be slow to decode and bulky.

3.4.3 JSON

JSON would be a good initial point if it was concise, faster to encode and decode and if it supported encoding of 64-bit integers reasonably.[1] JSON is described in RFC 7159 [3].

JSON evolved from a notation used for JavaScript objects in a rather complicated way. It is a text format which allows for rich structure. The first-class citizen in JSON is the "object", which is a dictionary made of key-value pairs. The keys are strings.

We have decided not to include this scheme in the benchmarks, as the results would be uninteresting due to JSON's similarity to BIRD's scheme in terms of size and efficiency.

3.4.4 Google Protocol Buffers

Google Protocol Buffers, or `protobuf` for short, were developed for Google's internal needs and open-sourced in version 2. They are a binary data interchange protocol, which is similar in spirit to CBOR internally.

To deploy Protocol Buffers, the developer first writes a `.proto` file specification in a domain-specific language which resembles C, which is shown in Figure 3.4.3. Then, the compiler `protoc` is used to generate code in one of the target languages from the `.proto` file, which uses `libprotobuf` to encode and decode data to and from the Protocol Buffers encoding.

```
message rte {
    required uint32 netaddr = 1;
    required uint32 prefix = 2;
    required uint32 gwaddr = 3;
    optional uint64 as_no = 4;
    optional string ifname = 7;
    required time uplink = 8;
    optional uint32 uplink_from = 9;
    required uint32 type = 10;
    optional rte_src src = 5;
    repeated rta attrs = 6;
}
```

Figure 3.4.3 Part of the `rt.proto` file used in benchmarks.

The generated code also defines the data structures ("containers") for data and convenient accessors.

Google Protocol Buffers suffer several problems. They do avoid version negotiation, but use numeric keys whose assignment would have to be centralized (this problem was described in Section 3.2.2). They make assumptions about the topology of data (creating problems described in Section 3.2.3).

[1] JSON does not define integer types, only floating point types, and 64-bit numbers are encoded as double-precision floats, which limits precision to the 52 bits of the mantissa.

The `protoc`-generated "containers" are semantically equivalent to what we need, but not "physically convenient" to store the data. We should also mention that the generated code is extremely hard if not impossible to follow or to debug.

Furthermore, the official implementation of Protocol Buffers we have benchmarked does not provide functions for streaming of messages and instead requires large messages to be pre-built in memory and then sent at once.

The on-line documentation of `protobuf` states that: "Protocol Buffers are not designed to handle large messages. As a general rule of thumb, if you are dealing in messages larger than a megabyte each, it may be time to consider an alternate strategy. That said, Protocol Buffers are great for handling individual messages within a large data set. Usually, large data sets are really just a collection of small pieces, where each small piece may be a structured piece of data. Even though Protocol Buffers cannot handle the entire set at once, using Protocol Buffers to encode each piece greatly simplifies your problem: now all you need is to handle a set of byte strings rather than a set of structures." [4]

The documentation basically says that large data sets are handled problematically and one basically has to invent a protocol built on top of Protocol Buffers to be able to handle large data sets—and BIRD would require just that. A surprising limitation of an otherwise great solution, which probably stems from the fact that Protocol Buffers were designed for and initially used for Remote Procedure Calls only. After all, it defines *messages*.

3.4.5 Fixed CBOR Scheme

Fixed CBOR Scheme is similar to the Fixed Binary Scheme: an individual item's meaning is deduced from its relative position in the stream. But with CBOR, great flexibility on the binary level is added, as the data type of each item in the stream is encoded with the data.

CBOR adds the "intelligence" which Fixed Binary Scheme lacks: it adds metadata to the stream without inflating its size too much. Also, it allows for formatting of "arrays" and "maps". We expect CBOR to be very compact, probably the most compact of all the schemes in the benchmark.

This scheme is easier to debug given the right tools. As each item's type is known, a stream decoder may be created that will pretty-print the items as they appear in the stream, making it easier to spot any differences in the order or type of items, as you may see in Figure 3.4.4. Without substantial modifications, this scheme would require version negotiation, too.

```
1823            uint(35)            35
F5              sval(21)            true
18D9            uint(217)           217
181F            uint(31)            31
18CD            uint(205)           205
18D1            uint(209)           209
F5              sval(21)            true
19C23C          uint(49724)         49724
00              uint(0)             0
07              uint(7)             7
9F              array(_)            [_
00              uint(0)             . . 0,
F4              sval(20)            . . false,
00              uint(0)             . . 0,
```

```
02              uint(2)             . . 2,
F4              sval(20)            . . false,
86              array(6)            . . [
196268          uint(25192)         . . . . 25192,
191B1B          uint(6939)          . . . . 6939,
193065          uint(12389)         . . . . 12389,
19C23C          uint(49724)         . . . . 49724,
19C23C          uint(49724)         . . . . 49724,
19C23C          uint(49724)         . . . . 49724,
                                    . . ]
```

Figure 3.4.4 A CBOR stream and its items.

We have chosen to benchmark this scheme (i) to measure the overhead of CBOR encoding as compared to Fixed Binary Scheme, and (ii) to measure the overhead of the CBOR-based protocol presented briefly in Section 3.4.6 and described in full detail later in Chapter 4.

3.4.6 NetBufs: a CBOR-based Protocol

We decided to design NetBufs as a thin protocol built on top of CBOR and see how it compares to established solutions. CBOR itself was designed to be extensible and usable as a binary layer for other protocols.

The key idea of NetBufs is that albeit CBOR adds great flexibility on the binary level, the meaning of items is still deduced solely from their position within the stream[1]. To avoid this shortcoming, we would like to prefix each item in the stream with a key describing the meaning of the item in a JSON-like fashion. While CBOR describes only syntactical meaning of data, NetBufs encoding describes its semantics, too.

This solves a lot of problems the Fixed CBOR Scheme has, but also introduces some problems in its own right; we shall address them in Chapter 4, where full description of NetBufs is given. For the brief description here, it is sufficient to contemplate Figure 3.4.5.

```
. . . . [                           . . . . [
. . . . . . {_
. . . . . . . . 0:
. . . . . . . . . . 2,              . . . . . . (bird.org/rte) {
. . . . . . . . 4:                  . . . . . . . . ./netaddr =
. . . . . . . . . . 12320956,       . . . . . . . . . . 12320956
. . . . . . . . 5:                  . . . . . . . . ./netmask =
. . . . . . . . . . 24,             . . . . . . . . . . 24
. . . . . . . . 2:                  . . . . . . . . ./gwaddr =
. . . . . . . . . . 16777226,       . . . . . . . . . . 16777226
. . . . . . . . 3:                  . . . . . . . . ./ifname =
. . . . . . . . . . "eth0",         . . . . . . . . . . "eth0"
```

Figure 3.4.5 NetBufs encoded stream (simplified `nbidag` output).

Figure 3.4.5 depicts a CBOR stream on the left-hand side and how its items were understood by netbufs on the right hand side. This is a preliminary illustration, detailed explanation will be given in Chapter 4.

[1] This is imprecise: the meaning of items is deduced from its position *relatively* to other items in the stream; i.e. if the meaning of a (homogeneous) array is known, for example, the meaning of its items is usually known, too.

Chapter 4
NetBufs

In this chapter, we will describe in full the encoding scheme we have designed.

The prefix "Net" in the name "NetBufs" does not come from "network" as everyone will surely guess, but was instead inspired by "net weight", suggesting there is only a light, thin packaging added to the data itself.

4.1 Motivation

Why design another data serialization protocol with so many out there? In Chapter 3, we have described six commonly used solutions: three text-based encoding schemes (BIRD, XML and JSON) which suffer performance issues, two very efficient encoding schemes (Fixed Binary and Fixed CBOR), which suffer flexibility issues and Google Protocol Buffers, which are quite flexible and very efficient, but suffer other problems.

This is not a criticism of the individual formats. There are applications of each of them certainly. But for use within BIRD, neither of them fits well.

Therefore, we have tried to design another encoding scheme—or protocol, if you like—to use for data exchange in BIRD, which is based on CBOR and will stick to the Requirements of Section 3.3 as close as possible. We hope to provide a high performance implementation which will produce reasonable size of the encoded data.

4.2 Informal Description

We shall start by first introducing the protocol informally, hoping that we will make it easier for the reader to understand the more technical parts to come. To make things even more illustrative, we will begin with a practical example.

4.2.1 Basics of NetBufs Encoding

For the purposes of benchmarking, we have defined several simple structures which mimic BIRD's internal structures that hold routing information, but we have left out all performance optimizations and technical glitches for brevity.

```
/*
 * Routing Table Entry
 */
struct rte
{
        ipv4_t netaddr;         /* target network address (IPv4) */
        uint32_t netmask;       /* target network prefix */
        ipv4_t gwaddr;          /* target gateway address (IPv4) */
        char *ifname;           /* interface where the gateway is reachable */
        struct tm uplink;       /* route uplink: how long the route is known  */
        bool uplink_from_valid; /* is uplink_from field valid? */
        ipv4_t uplink_from;     /* IPv4 address of the provider */
        struct rte_attr *attrs; /* route attributes */
        enum rte_type type;     /* route type */
        bool as_no_valid;       /* is as_no field valid? */
```

```
        uint32_t as_no;         /* number of the AS which provided the route */
        enum rte_src src;       /* route source */
};
```

Figure 4.2.1 `struct rte` as defined in `benchmark/src/benchmark.h`.

Without explaining[1] what each field in the structure means, which is unimportant for the example anyway, our goal will be to transfer this structure over a network.[2] Suppose we have decided to use NetBufs as the underlying protocol.

First of all, NetBufs-encoded streams are valid CBOR streams. Therefore, both the data required for the operation of the protocol and the payload itself are encoded in the form of CBOR data items.

In this chapter, we will often say "at the binary level" by which we will mean "at the level of individual CBOR data items". Similarly, "at the protocol level" means "at the level of NetBufs objects", which may be rephrased as "when data items have been decoded into pieces meaningful to NetBufs". We did not give precise definitions, but they should not be necessary.

To begin with, we would like to transfer a single `struct rte`. In BIRD's dump format, it may look like this:

```
188.0.188.0/24     via 10.0.0.1 on eth0 [uplink 13:47:35 from 217.31.205.209] * ...
    Type: BGP unicast univ
    /* route attributes omitted for brevity */
```

Figure 4.2.2 Example route to be encoded using NetBufs.

As NetBufs-encoded streams are just CBOR streams with additional information, we could start by encoding this data reasonably[3] in plain CBOR:

```
00BC00BC        uint(12320956)      12320956,
1818            uint(24)            24,
1A0100000A      uint(16777226)      16777226,
6465746830      text(4)             "eth0",
0D              uint(13)            13,
182F            uint(47)            47,
1823            uint(35)            35,
F5              sval(21)            true,
1AD1CD1FD9      uint(3519881177)    3519881177,
F5              sval(21)            true,
19C23C          uint(49724)         49724,
00              uint(0)             0,
07              uint(7)             7,
...             ...                 ...
```

Figure 4.2.3 The route from Figure 4.2.2 encoded in CBOR (shortened).

Actually, this is precisely what the `benchmark/src/serialize-cbor.c` module does and the scheme is precisely the Fixed CBOR Scheme we have described briefly in Section 3.4.5, where we have also identified some of its problems.

[1] But if you are interested, the meaning of individual fields in the `struct` maps directly onto the fields of Figure 1.3.1 commented in Section 1.3.

[2] Or to write it to a file, ...

[3] My initial misunderstanding of some of the poorly documented fields in BIRD's structures led to the representation not being very reasonable. The encoding of individual components of the time structure is pointless, as the time is relative and could very easily be stored in seconds. Correction would require change in all of the benchmarks. Also, what I call "netmask" in the examples is really the length of the network prefix, which I realized too late.

If you are interested how the individual attributes have mapped onto CBOR items, you may read the source file. The dump was obtained with `nbdiag` tool introduced in Section 6.8.

4.2.2 Encoding With String Keys

The first problem we are to solve is that the order in which the items were encoded matters. We want to drop this restriction and have the freedom of encoding and decoding items in any order. We shall therefore prefix each item with a string key. This would yield the following representation:

```
BF              map(_)              {_
676E657461      text(7)             . . "netaddr":
84              array(4)            . . . . [
18BC            uint(188)           . . . . . . 188,
00              uint(0)             . . . . . . 0,
18BC            uint(188)           . . . . . . 188,
00              uint(0)             . . . . . . 0,
                                    . . . . ],
676E65746D      text(7)             . . "netmask":
1818            uint(24)            . . . . 24,
6667776164      text(6)             . . "gwaddr":

/* omitted 6 lines (encoded IPv4 address) */

6669666E61      text(6)             . . "ifname":
6465746830      text(4)             . . . . "eth0",
6675706C69      text(6)             . . "uplink":
83              array(3)            . . . . [
0D              uint(13)            . . . . . . 13,
182F            uint(47)            . . . . . . 47,
1823            uint(35)            . . . . . . 35,
                                    . . . . ],
7175706C69      text(17)            . . "uplink_from_valid":
F5              sval(21)            . . . . true,
6B75706C69      text(11)            . . "uplink_from":

/* omitted 6 lines (encoded IPv4 address) */

6B61735F6E      text(11)            . . "as_no_valid":
F5              sval(21)            . . . . true,
6561735F6E      text(5)             . . "as_no":
19C23C          uint(49724)         . . . . 49724,
63737263        text(3)             . . "src":
00              uint(0)             . . . . 0,
6474797065      text(4)             . . "type":
07              uint(7)             . . . . 7,
FF              break               }
```

Figure 4.2.4 CBOR encoding with keys added (`nbdiag` output).

We have solved the aforementioned problem and created a new one. Now the keys are too plump, and that will make the encoding ineffective both in terms of time and space. We can solve that easily by replacing string keys with integers. But string keys have one undeniable advantage: they can be *structured*. How could that be of any use to anyone?

Suppose that we have successfully implemented NetBufs with integer keys within BIRD. Suppose that BIRD supported a plugin mechanism or similar feature. To end the assumption list, suppose we wanted BIRD's CLI to be able to retrieve plugin-specific information from a running BIRD instance upon request.

A plugin called `bpbe`[1] would therefore send information such as `bpbe/version` or even a map `bpbe/config` with all its configuration options. Now the advantage is hopefully obvious: we could easily avoid naming clashes by having structured (hierarchical) keys.

This could be made even better, and will be, by using keys which are URIs. Developers shall use only those keys with domain component of which they have authority over (for any agreed-upon definition of "having control over"). One could, for example, use the key `dcepelik.cz/bpbe/config` without having to worry about interfering with keys of others.

NetBufs therefore use string keys, but do it in a way which avoids vast majority of string encoding, decoding and comparisons. The mechanism is simple: prior to sending a key—say `dcepelik.cz/bpbe/version`—one has to write a NetBufs message that says: "the key `dcepelik.cz/bpbe/version` shall be referred to as 42 from now on". The number 42 is called a pID[2]. The NetBufs decoder has to keep a list of known keys, or precisely a mapping from pIDs to string keys.

More work will be needed if we do not want to mess with string keys as producers/consumers of a NetBufs stream. If things were kept this simple, one would have to call APIs such as `nb_send_i32(nb, "domain.org/key", 69);` which would need to compare strings internally. The consumer of the stream would do something similar, degrading performance terribly. Luckily the provided implementation managed to avoid almost all string comparisons—only those actually needed to establish key correspondence were kept.

4.2.3 Introducing Groups

For the moment let us ignore the details of how keys (or pIDs) are introduced into the stream. Instead, let us focus on *groups* and how they are useful to NetBufs.

The answer is not easy. Groups were first introduced as a direct intellectual descendant of C's `struct`[3]. The meaning of groups has changed several times. Groups now have the following functions in NetBufs:

- They package related attributes together. That is, a group has a meaning as a unit, as opposed to a collection of consecutive key-value pairs.
- When keys are URIs (or similarly structured), an attribute's name can be taken relative to a group's "base name", further conserving space and typing.
- Groups allow for crucial optimizations related to keys, which will be described in Chapter 6. (pIDs are "group-local", ergo small integers.)
- They allow for nicer decoder API. One can decode a group by switching the control flow on next attribute's ID in a loop until the end of group is reached. This is used by consumers to decode items out-of-order without buffering.
- They are a natural construct related to the way programmers seem to think and simplify debugging greatly.
- They make it possible to send a homogeneous CBOR array whose size is the actual number of groups in the array—that would not be possible otherwise.

Hopefully the above list convinced you that groups are useful, if not necessary.

[1] Best Plugin for BIRD, Ever.
[2] Protocol ID. There will be other IDs, too.
[3] At the very beginning, we have even considered a special type for `union`.

The last point needs additional commentary. The number of items in a definite-length CBOR array is the number of its direct descendants. For example, the size of the CBOR array `[ 1, 2, [ 3 ], [ 4, 5, 6 ], 6 ]` is 5. As each group is encoded in a single indefinite-length map, it counts as one item, regardless of the number of attributes it holds. If we had no groups, we would encode the attributes sequentially and the number of items in an array would no longer be a simple function of the number of encoded items, forcing us either to grow the array sequentially at the consumer side, or to transmit the actual size of the array in an extra item.

The encoding of the example route of Figure 4.2.2 in NetBufs with groups looks like this:

```
. . 0:
. . . . 3,                                  . . . . (bird.org/rte) {
. . 2:                                       . . . . . . ./netaddr =
. . . . 9044106,                             . . . . . . 9044106
. . 3:                                       . . . . . . ./netmask =
. . . . 23,                                  . . . . . . 23
. . 4:                                       . . . . . . ./gwaddr =
. . . . 16777226,                            . . . . . . 16777226
. . 5:                                       . . . . . . ./ifname =
. . . . "eth0",                              . . . . . . "eth0"
. . 6:                                       . . . . . . ./uplink =
. . . . {_
. . . . . . 0:
. . . . . . . . 4,                           . . . . . . (bird.org/time) {
. . . . . . 2:                               . . . . . . . . ./hour =
. . . . . . . . 13,                          . . . . . . . . 13
. . . . . . 3:                               . . . . . . . . ./min =
. . . . . . . . 47,                          . . . . . . . . 47
. . . . . . 4:                               . . . . . . . . ./sec =
. . . . . . . . 37                           . . . . . . . . 37
. . . . },                                   . . . . . . } /* bird.org/time */
. . 7:                                       . . . . . . ./type =
. . . . 7,                                   . . . . . . 7
. . 8:                                       . . . . . . ./src =
. . . . 0,                                   . . . . . . 0
. . 9:                                       . . . . . . ./attrs =
. . . . [                                    . . . . . . [

/* route attributes omitted for brevity */

. . . . ],                                   . . . . ] /* ./attrs */
. . 8:                                       . . ./uplink_from =
. . . . 3519881177,                          . . . . 3519881177
. . 17:                                      . . ./as_no =
. . . . 49724,                               . . . . 49724
}                                            } /* bird.org/rte */
```

Figure 4.2.5 NetBufs encoding (`nbdiag` dump, shortened, simplified).

As you can see, with each group, information about its type is encoded along with its other attributes: the first key-value pair in the group is required to be of the form `0:` *group-id*, where *group-id* is the pID of the group.

4.2.4 Introducing Keys

As we do not send string keys anymore and use integers instead, the correspondence of the string and the pID has to be established before a key is used for the first time. The producer does this by sending an 'key insert attribute' (KIR).

Before explaining the key-insertion mechanism, we should say that keys (pIDs really) are used not only to identify attributes in a group, but also to identify the groups. This can be seen in Figure 4.2.5: the decoder was able to deduce that the `uplink` attribute's group type is `bird.org/time`. We know data types of all binary-level items, it makes sense to know data types of protocol-level items, too.

4.3 Formal Description

In this section, we will describe the rules of NetBufs encoding.

4.3.1 CBOR Usage Rationale

There are many binary encoding formats: Abstract Syntax Notation One (ASN.1) with its various encoding options such as Basic Encoding Rules (BER) or Distinguished Encoding Rules (DER), MessagePack, Binary JSON (BSON) and RFC 713, to name only a few.

Table 4.3.1 shows several binary encoding standards and the size of encoding for two example inputs ('—' means that there is no equivalent of indefinite-length items). CBOR is the most compact scheme on these inputs, and, according to our opinion, the most "elegant" scheme. Also, the RFC which specifies CBOR encoding is well written.

The other candidate would be MessagePack. Further development of MessagePack, however, is held back by only having a few bytecodes available for extension of the encoding itself, and it insists on strict backward compatibility. We have chosen to use CBOR instead, as it seems to have fixed several of MessagePack's problems.

Format	`[1, [2, 3]]`	`[_ 1, [2, 3]]`
RFC 713	7 B	—
ASN.1 BER	13 B	15 B
MessagePack	5 B	—
BSON	34 B	—
UBJSON	10 B	11 B
CBOR	5 B	6 B

Table 4.3.1 Encoding size of various binary encodings.

4.3.2 Full CBOR Compatibility

We have decided that NetBufs-encoded streams will be CBOR-compatible at the binary level. While the gains of deviating from the RFC are uncertain, the benefits of being compatible are obvious: there are already tools that have built-in support for CBOR. Deviating from the standard would effectively mean we have lost the opportunity to use these tools for debugging.

To name one such tool, Wireshark, a well known utility for dissection of network streams, features built-in CBOR filtering and display capabilities. As we intend to use NetBufs over the network, too, the ability to use Wireshark will certainly be a benefit.

We have, however, created one incompatibility deliberately, which hopefully will not cause too much trouble: we only support `uint32` (4-byte) tags. This greatly improves `struct cbor_item`'s topology, resulting in much smaller memory footprint when DOM API is used (explained in Chapter 6). Tags are registered by IANA and their assignment is likely to be sequential, so it should take a rather very long time before the 4-byte range is depleted. This way, NetBufs-encoded streams are valid CBOR streams, but not all valid CBOR streams are valid NetBufs-streams.

4.3.3 Terminology

- 'Group' is an indefinite-length map.
- The keys of a group are called 'pIDs', standing for 'protocol IDs'.
- The key-value pairs of a group are called 'attributes'.
- Attribute with pID 0 is 'group type'.
- Attribute with pID 1 is 'extension attribute'.
- Attribute with a negative pID is 'key insertion request' (KIR).
- The value of a KIR is an indefinite-length map, whose keys are non-empty strings and whose values are pIDs.
- A pID is 'registered', if it was contained as a value in some KIR map.
- Attribute with a pID ≥ 2 is a 'data attribute'.

4.3.4 Specification of the NetBufs Encoding

The encoding is specified as a minimal set of constraints. Data stream fulfilling all the requests is a valid NetBufs stream.

1) A NetBufs stream is valid CBOR.
2) The top-level items in a NetBufs stream are groups.
3) Extensions attribute is reserved and its meaning is undefined as yet.
4) Group may only contain KIRs, group type attributes and data attributes.
5) Each group contains[1] a group type.
6) Group type precedes all data attributes in the data stream.
7) Every data attribute which appears in a group has a pID which was registered by a KIR which precedes the attribute in the data stream.
8) The keys of KIRs are unique in the whole file, start with -1 and the sequence of KIR keys decreases by one with each KIR.
9) The value of group type attribute is a registered pID.

4.3.5 Rationale And Comments

The rules of Section 4.3.4 deserve a few comments.

The requirement that a NetBufs stream is a valid CBOR stream is a natural one. It allows for decoding of NetBufs documents by tools which only support CBOR. Anyone who knows NetBufs and sees a NetBufs document in its CBOR diagnostic notation should be able to understand its contents easily.

The requirement that top-level items are groups makes decoding easier. This way, either end-of-file is reached, or an indefinite-length map follows, which is easy to check as the latter is comparison of the header byte with a constant.

[1] Precisely one, as implied by CBOR encoding rules.

The requirement that each group only contains KIRs, group type and data attributes makes decoding easier, as the keys of the map which makes up the group are integers. Using negative integers for KIRs allows us to detect KIRs quickly, because their type will be 'negative integer' (upper three bits `001`) as opposed to group type and data attributes, which are unsigned integers (upper bits `000`). The test for group type, too, is fast: the 0 will be encoded in a single zero byte, which can be tested quickly.

The requirement that a group contains group type is a necessity, otherwise the meaning of the data attributes would be unknown. This is why we also require that group type precedes data attributes. The same idea stands behind the requirement that the value of group type is a registered pID.

The uniqueness requirement is to facilitate debugging. This way, it is possible to say that 'pID 24 was inserted by KIR number 42', for example, when the key of the KIR which defined pID 24 was -42.

Chapter 5
Benchmarks

In Section 1.4, we have laid out some of the problems of the encoding scheme that BIRD currently employs and have expressed our concerns mostly about performance and about the space complexity of the output.

In this section, we will formalize these notions and run a set of benchmarks against several competing solutions. The output of the experiment will play important role in the final decision which scheme will be implemented in BIRD.

This section only presents our findings and interprets the measured data. See Chapter 6 for description of the benchmarking code.

5.1 Benchmarking Scheme

Figure 5.1.1 shows the "benchmark round-trip" that we use to measure performance of individual schemes/protocols. The upper dotted box shows input and output file, which are compared after each run using the standard `diff` utility, while the lower dotted box shows the work which is actually benchmarked.

As a corner case, when the BIRD scheme is being benchmarked, data is read into memory from the input file using `deserialize_bird`, then the clock is started, then the data is serialized using `serialize_bird` and time is read; the the clock is started over and the data deserialized again using `deserialize_bird` and again time is read; and then, finally, data is serialized for the last time using `bird_serialize` and written to output file.

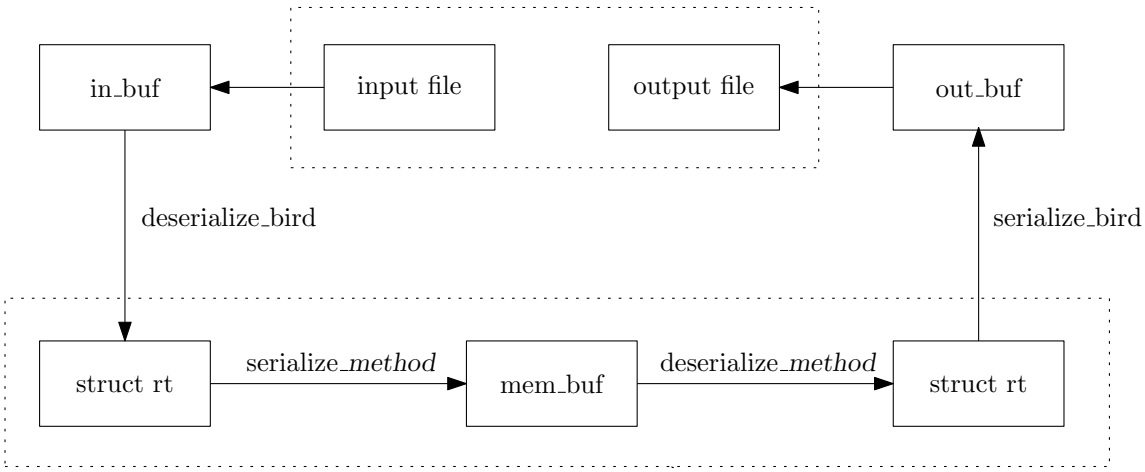

Figure 5.1.1 Benchmarking scheme.

5.2 Measurements

We have been pleased to see how well our scheme compares to mature solutions. The times in Table 5.2.1 were measured in the following setup:

- System Memory: 4 GB DDR3
- CPU: Intel Core i5-2540M
- Chipset: Intel QM67

- Operating System Kernel: Linux david-x220 4.11.9-1-ARCH
- GCC 7.1.1 20170621

Three rounds were run on an otherwise idle system and then times were averaged. Encoding size was reported by `nb_buffer_get_written_total`.

Method	Send	Receive	File Size
`binary`	0.192 s	0.517 s	100 MiB
`bird`	4.903 s	2.443 s	167 MiB
`cbor`	0.323 s	0.710 s	48 MiB
`netbufs`	0.604 s	1.186 s	79 MiB
`protobuf`	0.466 s	1.03 s	72 MiB
`protobuf-all`	1.466 s	1.73 s	72 MiB
`xml`	5.613 s	11.011 s	331 MiB

Table 5.2.1 Benchmark results.

5.3 Measurements Interpretation

5.3.1 Fixed Binary/Fixed CBOR/NetBufs

Measuring these three schemes, we can estimate the overhead of each.

The Fixed Binary scheme is, not surprisingly, the fastest to both encode and to decode. The measured time accounts for all the I/O operations and all `be64toh` and `htobe64` calls to convert numbers to/from host representation from/to network representation. Also, it accounts for the overhead of the `for` loops and the `switch` statements in the serialization and deserialization code, as well as for the operations involving dynamic arrays, such as calls to `array_size` or `array_push`.

Why do we mention all these operations in such detail? Because the CBOR benchmarks perform them all, too—and the extra time it takes to encode and decode CBOR is due to CBOR, not a result of having substantially different benchmark code.

CBOR takes 68 % longer to encode and 37 % longer to decode than the Fixed Binary scheme. We have tried to optimize the decoder for reasonable performance. We feel that the benefits of CBOR outweigh the minor decrease in performance it imposes. Also, CBOR encoding is indeed extremely compact, requiring only 48 % of the size of Fixed Binary encoding.

The measurements for CBOR account for all work the Fixed Binary scheme performs, plus extra work in all the `cbor_encode_*` and `cbor_decode_*` calls, which involves time spent reading and writing metadata and encoding/decoding variable-length integers. The CBOR encoder and decoder perform tens of sanity checks on the data, including integer range checks, checks for invalid metadata, etc. The extra time is also due to these checks and strictly speaking, CBOR encoding and decoding itself is slightly faster than we measured.

On to NetBufs, which take roughly 186 % of CBOR's time to encode and 229 % of CBOR's time to decode. NetBufs provide the ability to decode items in any order, the ability to encode and decode optional items without having

to write bitmaps or other flags, the ability to encode and decode completely unknown items through the DOM API and a decentralized and automated key-exchange mechanism. The writer and reader of NetBufs perform lots of checks to help the programmer spot mistakes. The encoding size of NetBufs is 164 % the size of CBOR encoding.

We feel that the increase in file size and encoding/decoding time pays itself in the added flexibility. Furthermore, we believe there are opportunities for further fine-tuning of the code, further reducing encoding and decoding time. As with CBOR, we think this is a reasonable price to pay for the added features.

5.3.2 Bird/NetBufs

The comparison of BIRD and NetBufs is not strictly fair in terms of processing time. As we explained, BIRD encodes a lot of numbers in decimal, and this was not optimized in the benchmarked encoder. The parser, however, is quite efficient and the decoding of BIRD format is surprisingly fast, probably because decoding decimal numbers is much faster than encoding them.

In terms of size, however, the measurements are exact: the NetBufs encoding is only 47 % the size of BIRD encoding.

Even if we admit that the BIRD serialization and deserialization code could be optimized, the measure performance gap between BIRD and NetBufs seems large enough to be conclusive—after all, encoding of NetBufs is almost an order of magnitude faster and the decoding, which is roughly three times faster, does not allow for much improvement on BIRD's side.

We conclude that NetBufs are substantially faster to both encode and decode than BIRD's contemporary format and the resulting encoding of data is much smaller.

5.3.3 NetBufs/Google Protocol Buffers

Google Protocol Buffers are very efficient, both in terms of speed and in terms of encoding size.

As for size, `protobuf` is genuinely efficient and takes about 10 % less than NetBufs' encoding, to our irritation. First of all, `protobuf` does not transfer string keys, but that is approximately 350 B worth of data in the case of this particular benchmark, therefore negligible. But `protobuf` seems not to use a construct equivalent to CBOR's 'break' code. More precisely, `protobuf` encoding is not self-describing at all, an external description of the data must be available to decode it. (The description is contained implicitly in the code generated from the `.proto` files.)

In terms of speed, `protobuf` is slightly faster. But the `protobuf` times in Table 5.2.1 do not account for the time it took to translate internal structures of our program to the structures provided by the `protoc` compiler (this problem was described in detail in Section 3.2.3).

The inclusive measurement of the time it takes to construct `protobuf` structs and then serialize them, resp. to decode `protobuf` data into benchmark data structures, is shown in the table as `protobuf-all`. It is roughly 2.5 times slower in encoding and 1.5 times slower in decoding than our NetBufs implementation.

It seems that `protobuf` is a great solution where messages are small, sent individually and their topology is not important to the application.

5.3.4 BIRD/XML

We compare BIRD to XML, because comparison with binary formats does not seem fair. XML as an encoding scheme is much more regular and flexible than the original BIRD solution, but the price seems unacceptable. The encoding size is enormous, roughly twice the size of BIRD, which is too much itself.

The encoding is produced by calling `printf`-like functions, just as BIRD's encoding, but there is more information to be written, therefore the encoding is slower. The decoding is a lot slower—roughly 6 times as slow. Strictly speaking, this comparison is a unfair, because we use `libxml2` for decoding, a popular XML library which turned out to be very slow. But we suspect no XML parsing library would be as fast as the BIRD parser we have written.

5.4 Conclusion

The purpose of the benchmarks was (i) to compare the NetBufs encoding described in Chapter 4 to the encoding BIRD uses and (ii) to compare NetBufs to standard solutions, neither of which was a perfect fit.

We conclude that the NetBufs encoding is significant improvement upon BIRD's encoding scheme and that it is comparable in terms of speed and encoding size to Google Protocol Buffers.

Chapter 6

Implementation

Our solution is naturally decomposed into several subsystems, each of which were developed, tested and documented independently.

The provided sources consist of two main parts: the NetBufs source code in `src` directory, and the benchmarking code within `benchmark/src`.

6.1 Dependencies

The following tools are required to build `nbdiag`: `make`, `gcc`. To run tests, `jq`, `curl` and `xxd` are needed. To build the benchmarks, `libxml2` and `libprotobuf` are required.

6.1.1 Building

Run `make` within `build/` to compile NetBufs sources into object files and to build executables required for testing and benchmarks. Refer to BIRD documentation for information how to build BIRD [2]. (NetBufs are built along automatically.)

6.2 Interfaces

To simplify deployment of NetBufs, memory-management and I/O are performed through well-defined interfaces, which allow the implementers of NetBufs to customize them to fit the application.

6.2.1 Memory-management Interface

The memory interface is defined by the header file `include/memory.h`. The default implementation of this interface is provided by `memory.c` and `mempool.c` for cases with no special memory-management requirements.

The interface provides `nb_malloc`, `nb_realloc` and `xfree` functions intended as wrappers around the Standard C Library functions to handle out-of-memory situations. Their default implementation provided in `memory.c` prints an error message and exits.

Furthermore, the interface assumes existence of a memory-pool to aggregate small allocations together and allow for easier freeing of memory. The mempool provided by `mempool.c` is a simple, yet quite powerful memory pool without support for individual deallocation of objects.

6.2.2 I/O Interface

NetBufs need to perform a lot of I/O in order to encode and decode data. The functions for I/O are provided by the `buffer` module in `include/buffer.h`.

The `buffer` module provides `struct nb_buffer`, which is an abstraction of a data storage and retrieval mechanism. The code which uses `buffer` calls functions such as `nb_buffer_read`, providing pointer to `nb_buffer` as an argument,

regardless of whether data is read from memory a file[1], or a completely different storage mechanism.

When `nb_buffer_write` is called, for example, then data is buffered internally in `struct nb_buffer` and when the internal buffer fills up, the function pointer `fill` from `struct nb_buffer_ops` is called to handle the situation. This may consist of calling `write` to write the data to a file, or by saving the buffer contents in memory, or by doing something else, depending on the storage mechanism.

The `buffer` module is easily extensible. The implementer shall define several functions to fill the buffer, to flush the buffer, to delete the buffer and to tell the position within the buffer. The `struct nb_buffer_ops` is then initialized with the appropriate function pointers and called by the `nb_buffer_*` functions as needed to perform the actual I/O.

At the moment, only the `buffer-memory` and `buffer-file` modules are provided by NetBufs. The `buffer-memory` module is used as the intermediate buffer in the benchmarking code to which data is encoded and from which it is later decoded.

6.3 CBOR Encoder and Decoder

We have developed a CBOR encoder and decoder. The encoder supports a subset[2] of features defined by CBOR, which the decoder is capable of decoding correctly.

The encoder and decoder provide two main APIs: stream-based and DOM-based. The stream based API encodes or decodes CBOR items to/from the stream one at a time. To encode a complex structure like an array, first the function `cbor_encode_array_begin` is called, then various calls are made to encode the items of the array, and then `cbor_encode_array_end` is called. The stream-based API was optimized for high-performance and minimum memory footprint.

The DOM-based API, on the other hand, provides only two functions called `cbor_encode_item` and `cbor_decode_item`, which take `struct cbor_item` as an argument. The `struct cbor_item` encapsulates a generic CBOR item, regardless of its kind. It allows the programmer to receive items without knowing their type in advance, and send them later.

To keep track of open "blocks"—arrays, maps and indefinite-length byte or text streams—the encoder and decoder use stack of blocks. The stack functionality is provided by the generic `stack` module in `stack.c`. The block stack keeps context of currently open blocks, their type (array, map, etc.) and the number of items in the block. Upon closing definite-length blocks, the encoder and decoder are therefore capable of checking that the appropriate number of items was encoded to/decoded from the block.

The block-stack used by CBOR encoder and decoder is actually used by NetBufs encoder and decoder as well, even though only to write meta-data, not to push or pop it. We explicitly note that here as well-documented hacks may be called "engineering decisions". This

[1] A 'file' in the Unix sense—anything that has a file descriptor.

[2] It does not support encoding or decoding of floating-point numbers and only 32-bit tags are supported for performance reasons.

particular engineering decision avoids code duplication, as both the CBOR and the NetBufs module need to keep track of open blocks and NetBufs' open blocks happen to be a subset of CBOR open blocks. That is because each NetBufs group is an indefinite-length map, as you may recall from Section 4.2.3.

When an error occurs during the encoding or decoding of a CBOR stream, the internal `error` function is called, which in turn calls the user-registered error handler. By default, `cbor_default_error_handler` is registered which handles the situation by printing an error message and exiting with an error code. In many scenarios, however, this behavior is unacceptable, because the application may first need to free other resources or terminate network connections, etc.

Therefore, `cbor_stream_set_error_handler` may be called to set up a different handler which will perform the cleanup. The custom handler is required to terminate further processing of the CBOR stream, otherwise behavior is undefined. This may be done by terminating the process, by canceling the thread or by using a `longjmp`-ing to previously `setjmp`'d location to simulate exception handling.

When the `DIAG_ENABLE` macro is `#defined` in `include/diag.h` to a value which evaluates to `true`, the decoder performs calls to various `diag_*` calls which provided the user of the decoder with detailed diagnostics of the CBOR decoding process. This slows the decoding process down by a factor of two and in production code, all diagnostics should be disabled to maximize performance.

Several functions on the hot path, such as `cbor_is_break`, `cbor_peek` and `cbor_decode_uint64`, are defined `static inline` in `include/cbor.h` to speed up the decoding process.

6.4 Serialization and Deserialization Routines

The functions for sending and receiving data using NetBufs are defined in `send.c` and `receive.c` respectively, with common code placed in `netbufs.c` which handles the configuration of a NetBuf.

At the moment, NetBufs only support sequential decoding of items. The DOM-based API will be created later when required by BIRD. The API is very similar to the CBOR streaming API, with the exception that the functions for sending of items take one more argument—the ID of the item which is being sent, as described in Chapter 4.

To configure a NetBuf, several functions are provided for defining groups are their attributes. These are describe in Chapter 7.1.

6.5 Coroutines

To overcome the difficulties of BIRD's I/O handling described in Section 1.5, the BIRD team[1] has equipped us with coroutine support. The interface of BIRD's coroutines module resides in `$BIRD_ROOT/lib/coroutine.h` and the implementation is system-dependent and resides in `$BIRD_ROOT/sysdep/unix/coroutine.c`.

The implementation of coroutines in BIRD relies on `<ucontext.h>`, but a fallback version which is implemented using POSIX Threads API for systems which

[1] One of my adviser's many nicknames.

do not provide <ucontext.h>. The functions provided by <ucontext.h> have been declared obsolete by POSIX.1-2008 and their destiny remains uncertain.

6.5.1 Coroutines and <ucontext.h>

The <ucontext.h> header provides functions for user context manipulation. These functions allow one to implement coroutines in C.

Coroutines

6.5.2 The I/O Problem

One of the problems we had to solve was that all reads on sockets in BIRD are non-blocking. Assume that the following sequence of calls is made:

```
nb_recv_group(nb, BIRD_ORG_COMMAND);
        while (nb_recv_attr(nb, &id)) {
                switch (id) {
                case BIRD_ORG_COMMAND_TEXT:
                        nb_recv_string(nb, &cmd_text);
                        break;
                ...
                }
        }
}
nb_recv_group_end(nb);
```

Which will most likely translate to the following sequence of calls to the CBOR decoder:

```
cbor_decode_map_begin_indef(nb->cs);
cbor_decode_uint64(nb->cs, ...); /* recv_pid, will likely decode a 0 */
cbor_decode_uint64(nb->cs, &pid); /* read the group type */
/* check that pid == BIRD_ORG_COMMAND, call nb_error otherwise */
/* nb_recv_attr(nb, &id) */
cbor_is_break(nb->cs); /* returns false */
cbor_decode_uint64(nb->cs, &id)
/* case BIRD_ORG_COMMAND_TEXT: */
cbor_decode_text(nb->cs, &cmd_text);
 /* nb_recv_attr(nb, &id) */
cbor_is_break(nb->cs); /* returns true */
cbor_decode_break(nb->cs);
```

Each of the cbor_* calls above translates to something like the following:

```
nb_buffer_read(cs->buf, &hdr, 1); /* to decode the header */
/* find out the type of the item */
nb_buffer_read(cs->buf, &payload, n); /* to decode item payload */
```

The nb_buffer_read(buf, dst, n) call would perform something like:

```
size_t avail = buf->len - buf->pos; /* how many bytes are in the buffer? */
if (avail >= count) {
    /* memcpy data from buffer to dst */
}
else {
    buf->ops.fill(buf);
}
```

And the fill handler would (finally!) call read on the socket's file descriptor (if the nb_buffer_file back-end was used). Assume that the read on the non-blocking socket would return E_WOULDBLOCK to signalize that no data is available at the moment at the socket. What happens next? We may find ourselves in

the middle of the decoding of an integer and our only way to get the rest of the integer—not to mention the rest of the `BIRD_ORG_COMMAND` group—is to return control to `io_loop` to call `rx_hook` the next time data is received on the CLI socket.

But if we return control to `io_loop` in the "common sense", that is, return from the `fill` handler, the `nb_buffer_read` function, etc., all the processing context will be lost. Ultimately, the `cbor_*` call which initiated the reading will fail, no data will be decoded and next read position within the stream will remain stuck in the middle of an item. So, simply returning from the function cascade is not an option.

The situation we have described is a very mild version of a much worse problem in general. Imagine you were decoding an array of integers, which itself is a value in a map, inside your custom `decode_foo` function, which was called in a `for` cycle from your `decode_bar` function. If that seems as an artificial example, have look at the `deserialize_netbufs` function in `benchmark/src/deserialize-netbufs.c`.

If we really wanted to return from the leaf function in the call-tree, we would have to save all the processing context (all the open blocks, array indices, etc.), which would complicate the API and certainly affect performance negatively.

6.5.3 Coroutines Solution

Our solution uses coroutines to solve the problem. Figure 6.6.1 shows the process (cf. Figure 1.5.1). First, BIRD's `main` is called, which starts the `io_loop`. Once data arrive at the CLI socket, that socket's `rx_hook` gets invoked. This hook calls `coro_resume`, which, when called for the first time, calls `entry_point`.

The entry point itself is an infinite loop, which parses client's commands and calls the appropriate handlers. As the parsing of the command is really a decoding of a NetBufs message, when some of `nb_recv_*` calls would block on reading from a socket, control is returned to the main program by calling `coro_suspend`. The `io_loop` continues processing and when data is received on the CLI socket again, `coro_resume` is called for the second time, returning control precisely to the point where `coro_suspend` was previously called.

This way, no returning from the deep context has to be done. Control is handled to `io_loop` in a natural way, storing context of the whole program instead, which is an elegant, universal solution to many practical problems, which would otherwise be challenging to solve.

6.5.4 Performance of `<ucontext.h>`

We have not had the time to do a proper benchmark of the functions provided by `<ucontext.h>`, but we should point out that the context switch is not for free. When context is switched by a call to `swapcontext`, on the AMD64 platform, the general purpose non-scratch registers have to be saved, as well as the MMX registers, which is around half a kilobyte of data. Furthermore, a syscall has to be performed to adjust signal masks.

6.6 BIRD Integration

NetBufs' source code is deployed into `$BIRD_ROOT/netbufs`. A `Makefile` is located in the same directory which is later included by `$BIRD_ROOT/Makefile.in`. "Bridges"—the `buffer-cli` module and `nbids.h`—are located in the directory `$BIRD_ROOT/netbufs-bridges`.

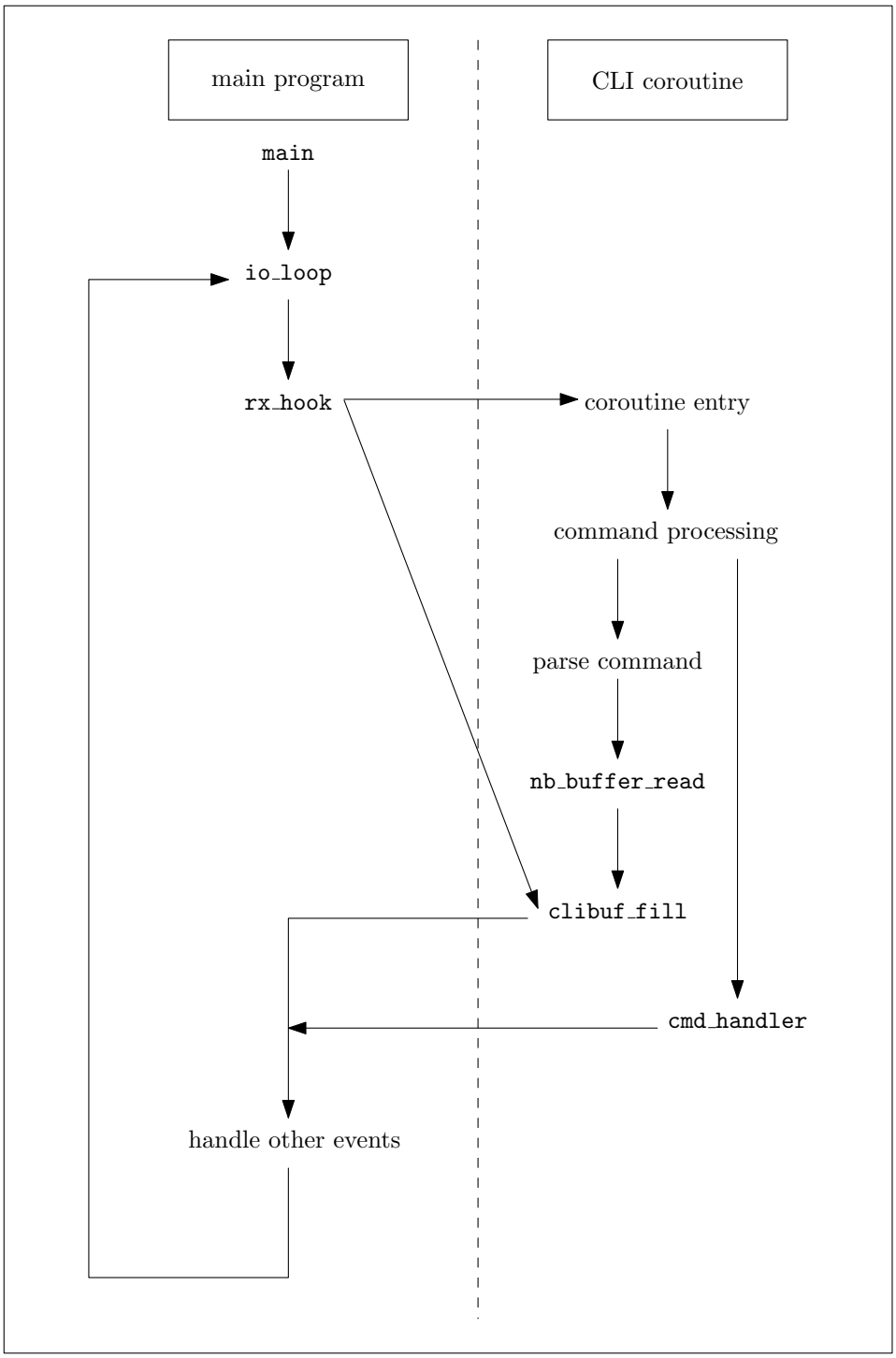

Figure 6.6.1 The main program and the CLI coroutine (simplified).

The changes required to connect NetBufs to BIRD were minimal, as the code was designed to allow for that from the first moment. The only interesting part is

the CLI buffer encapsulation in `$BIRD_ROOT/netbufs-bridges/buffer-cli.c`, which encapsulates reading from CLI TX/RX buffers.

6.6.1 Limitations

We wanted to demonstrate the power of NetBufs on a large example, such as transferring of routing tables. Unfortunately, we have underestimated the complexity of the code which generates routing table dumps, which is caused by BIRD's tangled implementation of route attributes, especially. Therefore, we have replace BIRD's CLI protocol, with clients remaining fully compatible, only; the use of NetBufs encoding in BIRD is limited to the transfer of `BIRD_NIC_CZ_CMD` and `BIRD_NIC_CZ_REPLY` groups, which are used to transfer commands and BIRD's replies, respectively.

6.6.2 Protocol Replacement

First, the CLI buffers have been implemented in a straightforward way, using the coroutine-based I/O. Then the command parsing logic in `cli_read_line` in `$BIRD_ROOT/nest/cli.c` was replaced and command buffering logic was altered in `cli_printf` in the same file.

The response-processing logic was overridden in `server_read`, which is located in `$BIRD_ROOT/client/client.c`.

On the client side, netbufs are initialized in `server_connect`, on the server side, they are initialize in `cli_new` (`$BIRD_ROOT/cli/cli.c`).

Compared to the complexity of the NetBufs benchmarking code and to the complexity of I/O issues which had to be handled, the implementation of NetBufs to replace the text-based protocol was trivial and deserves no further commentary.

6.7 Benchmarking Code

The code to obtain benchmarks exhibits a very "pragmatic" approach to programming. This does not mean it is incorrect or shoddy, but rather that it was written as a throw-away piece of software and was not designed for use out of the benchmarking environment.

This means that, for example, `serialize-binary.c` does not provide a fixed binary encoder one could use in production without modification, as it does not communicate errors and instead calls `abort` (through `assert`) on all failures, etc.

In the rest of this section, all paths are relative to the directory `benchmark/src`.

The entry point for benchmarks is located in `benchmark.c`, which handles command-line options and establishes required resources (files and buffers). Also, it uses `bird_deserialize` to load test data (a BIRD routing table dump) into memory. Later it serializes data into the BIRD format using `bird_serialize` to allow for output comparison.

The comparison is carried out as a safeguard against many of the bugs which may manifest in unexpected ways in the process of "transcoding" of the data.

The following modules exist whose names are hopefully descriptive enough: `binary`, `bird`, `cbor`, `netbufs`, `protobuf` and `xml`. Each of these modules provides the following two functions:

- `void serialize_module(struct rt *rt, struct nb_buf *buf);`
- `struct rt *deserialize_module(struct nb_buf *buf);`

These methods and their cooperation with the rest of the benchmarking environment is depicted in Figure 5.1.1. They are defined along with other (de)serialization routines in eponymous files. Their declarations are present in the global `benchmark.h` header file.

A script to run all the benchmarks `../run-benchmarks.sh` is provided and the reader is encouraged to run it to verify the results presented in Chapter 5. The data for benchmarking may be found in `../data`.

6.8 The `nbdiag` Diagnostic Utility

`nbdiag` is a tool we have written to aid in the debugging process. `nbdiag` takes advantage of both CBOR and NetBufs design and, through various `diag`-calls in various parts of the code which handles decoding, shows the actual progress of the decoding of the stream.

This is an awkward way of saying that the output of `nbdiag` is created sequentially by the *actual code* that processes the stream. Therefore, when an error occurs, the location of the error is simply at the tail of `nbdiag`'s output. This way, the developer can see exactly how the stream she attempts to decode was understood by various parts of the system.

The output of `nbdiag` consists of several columns of output, which always appear in the following order from left to right:

1) `offset` column shows position within the stream where the first byte of the `raw` column was located,
2) `raw` column contains the actual bytes in the stream which were decoded,
3) `item` column shows the stream of CBOR items as decoded from `raw` data,
4) `cbor` column shows CBOR structures in the CBOR Diagnostic Notation using indentation as a visual aid, and finally
5) `proto` column shows the NetBufs objects reconstituted from the `cbor` data.

`nbdiag` has various options to customize the output to fit the developer's needs, such as which columns should be shown or what is the maximum length of byte- and text-string previews. Run `nbdiag --help` for usage information.

For example, to obtain Figure 4.2.4, one would run (within `/build`):

```
make && ./benchmark bc-ex1 ../benchmark/data/rt4.small /tmp/out
     && ./nbdiag /tmp/out
```

The tool was written before we have settled on the last details of NetBufs encoding, which was an advantage as we could witness the operation of the protocol in various setups. The tool turned indispensable in debugging of CBOR and NetBufs benchmarks.

Furthermore, while benchmarks for NetBufs were written, the NetBufs-related code was still very new and one could not be sure what causes the errors—whether there is a bug in the way NetBufs are used or in the NetBufs code-base itself. Without `nbdiag`, the debugging would surely take much longer. This advertisement block is to point out that we have not

only created a working implementation of a data exchange protocol, but have also provided debugging tools for developers who happen to use it.

The source files relevant for `nbdiag` are `src/nbdiag.c` and `src/diag.c`.

6.8.1 Library Functions For Diagnostics

`nbdiag` depends largely on library code to produce the output. It really doesn't do anything very interesting in its own right; it is merely a parameterized "runner" of various library routines.

These library routines all reside in `diag.c`. The header `diag.h` defines some macros for convenience, such as `diag_print_cbor`, which first test whether diagnostics is enabled and if it is, call `diag_print_cbor_internal` which handles the actual printing.

A "diagnostics buffer"—called `struct diag`—is passed as an argument to most `diag_*` functions. This is not the most elegant data structure in the universe, but it does its job: it alleviates the rest of the code from having to handle various formatting corner-cases and minor technical difficulties. As many of the diagnostics calls are scattered around the various decoding code, it makes sense to have as simple calls as possible so that the diagnostics code is not too distracting.

There is a minor exception: in some cases, the decoder checks whether diagnostics is set in JSON mode and if so, it prints slightly different output. See for example the `cbor_decode_array_begin_indef` function in `decode.c`.

6.9 Test Suite

The CBOR encoder and decoder and the buffers are covered by tests located in `tests/`, where the script `run-tests.sh` resides which sets-up the testing environment and then commences the tests.

CBOR tests are located in `tests/cbor` and test the CBOR decoder. There are four "categories" of tests: `local`, `corrupt`, `rfc` and `random` in eponymous directories. The `local` tests were written by us, the `corrupt` files are deliberately broken is sophisticated ways to test the stability of the decoder. The `rfc` tests are a large tests suite which is generated by `run-tests.sh` upon first run from the official CBOR test vectors. A JSON specification of the tests is downloaded from GitHub, tests are extracted from the file and converted into the format used by `run-tests.sh`. Finally, `random` tests are random files which should not break the decoder under any circumstances.[1]

The buffers are tested using a simple tool called `buffer-echo` which reads a given file and writes it back through the buffers; the results are then diffed to make sure that no corruption of the stream has occurred. Several random files are presented to the program which are generated by `run-tests.sh` on every run.

`run-tests.sh` tests several things. First, it runs the test in Valgrind once and checks Valgrind's verdict about the number of memory errors detected. If

[1] But they sometimes produce false positives: if an otherwise correct item is decoded from the stream, such as an array with several billions of items declared in the header, the decoder exits with an error due to running out of memory, which is correct behavior, but is reported as incorrect by `run-tests.sh`, which expects the test to fail with a parse error.

Valgrind spots any error, the test fails. Second, it tests the input file to reference output in JSON/CBOR diagnostic notation. Third, it checks the exit code[1] to spot runtime errors.

An important source of tests are the `cbor` and `netbufs` benchmarks. As their results are diffed by `run-benchmarks.sh`, any difference produced by, for example, reading of invalid memory will manifest itself in difference of the outputs or runtime error. Running the benchmark in Valgrind is also possible, but very slow for large data.

Altogether, there are 86 different active[2] tests for the CBOR decoder, a variable number of auto-generated tests for the buffers and the two extensive benchmarks which test CBOR and NetBufs thoroughly.

In the future, we would like to add `corrupt` tests for NetBufs.

[1] For `corrupt` tests, the exit code must be `EXIT_CBOR_ERROR` to indicate a graceful parse error. This makes it possible to distinguish deliberate failure of the program to a failed `assert`, for example. This is why failing with out-of-memory error results in a failed test.

[2] Some tests in the RFC suite are disabled, such as float tests.

Using Netbufs

In this short chapter, we would like to provide information about NetBufs usage. Please refer to Chapter 4 for more information about NetBufs, their internal operation and philosophy.

7.1 Limitations

At the moment, checking that required values are present in a message is not implemented. Also, receiving of an unknown group is an error.

7.2 Configuration

A buffer must be created before communication can begin. `nb_buffer_new_file` will open new file buffer:

```
struct nb_buffer *nb_buf = nb_buffer_file_new(fd);
```

where `fd` is the file descriptor to be used, obtained by previous call to `open`.

Next, NetBuf is initialized using `nb_init`. The, `nb_group` and `nb_bind` are used to create groups and to bind their attributes, respectively:

```
nb_init(&nb, nb_buf);

rt = nb_group(&nb, BIRD_ORG_RT, "bird.org/rt");
nb_bind(&nb, rt, BIRD_ORG_RT_ROUTES, "./routes", true);
nb_bind(&nb, rt, BIRD_ORG_RT_VERSION, "./version", true);

rte = nb_group(&nb, BIRD_ORG_RTE, "bird.org/rte");
nb_bind(&nb, rte, BIRD_ORG_RTE_AS_NO, "./as_no", false);
nb_bind(&nb, rte, BIRD_ORG_RTE_ATTRS, "./attrs", true);
nb_bind(&nb, rte, BIRD_ORG_RTE_GWADDR, "./gwaddr", true);
nb_bind(&nb, rte, BIRD_ORG_RTE_IFNAME, "./ifname", false);
nb_bind(&nb, rte, BIRD_ORG_RTE_NETADDR, "./netaddr", true);
nb_bind(&nb, rte, BIRD_ORG_RTE_NETMASK, "./netmask", true);
nb_bind(&nb, rte, BIRD_ORG_RTE_SRC, "./src", true);
nb_bind(&nb, rte, BIRD_ORG_RTE_TYPE, "./type", true);
nb_bind(&nb, rte, BIRD_ORG_RTE_UPLINK_FROM, "./uplink_from", false);
nb_bind(&nb, rte, BIRD_ORG_RTE_UPLINK, "./uplink", false);

...
```

Figure 7.2.1 NetBuf configuration (`benchmark/src/nbids.h`).

The arguments of `nb_group` are pointer to the NetBuf configured, the integer which will be used to identify the group (will be used for `nb_send_group` and `nb_recv_group` calls) and a semantic name.

The arguments to `nb_bind` are pointer to the NetBuf configured, the integer which will be used to identify the argument (will be used by all `nb_send_*` calls for attribute serialization and returned by `nb_recv_attr`), the semantic name of the attribute and 'required' flag.

The integers you provide to `nb_group` and `nb_bind` are arbitrary. Preference of small integers is reasonable, as it improves performance. The semantic names, too, are arbitrary.

Please note that both NetBufs participating in communication have to be configured, preferably the same way at the moment, as receiving of unknown groups is treated as an error.

7.3 Sending/Receiving Data

Data is sent and received on a NetBuf by calling the numerous `nb_send_*` and `nb_recv_*` functions.

7.3.1 Sending API

In these functions, the `nb_lid_t` argument is the integer value associated with the attribute using `nb_bind`.

- `void nb_send_bool(struct nb *nb, nb_lid_t id, bool b);`
- `void nb_send_i8(struct nb *nb, nb_lid_t id, int8_t i8);`
- `void nb_send_i16(struct nb *nb, nb_lid_t id, int16_t i16);`
- `void nb_send_i32(struct nb *nb, nb_lid_t id, int32_t i32);`
- `void nb_send_i64(struct nb *nb, nb_lid_t id, int64_t i64);`
- `void nb_send_u8(struct nb *nb, nb_lid_t id, uint8_t u8);`
- `void nb_send_u16(struct nb *nb, nb_lid_t id, uint16_t u16);`
- `void nb_send_u32(struct nb *nb, nb_lid_t id, uint32_t u32);`
- `void nb_send_u64(struct nb *nb, nb_lid_t id, uint64_t u64);`

- `void nb_send_array(struct nb *nb, nb_lid_t id, size_t nitems);`
- `void nb_send_array_end(struct nb *nb);`

- `void nb_send_string(struct nb *nb, nb_lid_t id, char *str);`
- `void nb_send_group(struct nb *nb, nb_lid_t id);`
- `nb_err_t nb_send_group_end(struct nb *nb);`

7.3.2 `while/switch` construct for receiving

To receive data in any order, a `while/switch` construct is recommended which uses `nb_recv_attr` to decode attributes in a group until there are no attributes left. This allows for receiving of attributes in any order. Figure 7.3.1 shows the construct.

```
static nb_err_t recv_bgp_cflag(struct nb *nb, struct bgp_cflag *cflag)
{
        nb_lid_t id;

        nb_recv_group(nb, BIRD_ORG_BGP_CFLAG);
        while (nb_recv_attr(nb, &id)) {
                switch (id) {
                case BIRD_ORG_BGP_CFLAG_FLAG:
                        nb_recv_u32(nb, &cflag->flag);
                        break;
                case BIRD_ORG_BGP_CFLAG_AS_NO:
                        nb_recv_u32(nb, &cflag->as_no);
                        break;
                }
        }
        nb_recv_group_end(nb);
}
```

Figure 7.3.1 The `while/switch` construct.

7.3.3 Receiving API

- `bool nb_recv_attr(struct nb *nb, nb_lid_t *id);`

- `void nb_recv_bool(struct nb *nb, bool *b);`
- `void nb_recv_i8(struct nb *nb, int8_t *i8);`
- `void nb_recv_i16(struct nb *nb, int16_t *i16);`
- `void nb_recv_i32(struct nb *nb, int32_t *i32);`
- `void nb_recv_i64(struct nb *nb, int64_t *i64);`
- `void nb_recv_u8(struct nb *nb, uint8_t *u8);`
- `void nb_recv_u16(struct nb *nb, uint16_t *u16);`
- `void nb_recv_u32(struct nb *nb, uint32_t *u32);`
- `void nb_recv_u64(struct nb *nb, uint64_t *u64);`

The space will be allocated from the string by NetBufs and should be freed by a call to `free` when not needed anymore.

- `void nb_recv_string(struct nb *nb, char **str);`

- `/* nb_recv_array is a macro */`
- `void nb_recv_array_end(struct nb *nb);`

7.4 Example

See `serialize-netbufs.h` and `serialize-netbufs.c` in `benchmark/src` for a complete example of serialization and deserialization of a complex data structure. See `benchmark/src/nbids.h` for configuration.

Conclusion

The goal of this work was to explore the options for encoding of data in the BIRD CLI, and to implement it within BIRD as a proof-of-concept. Upon reviewing the existing approaches to data serialization, we have decided to design our own solution.

In the benchmarks, our solution performed well. It bears comparison to Google Protocol Buffers in terms of speed and encoding size, but is more flexible and a better fit for BIRD. The solution is covered by tests, although testing it yet more could not do any harm.

Our solution was implemented in BIRD. To overcome the limitations BIRD has, especially its event-driven approach to I/O, we had to employ coroutines. The solution was used to replace both the CLI protocol used by BIRD today. The patched version of BIRD is however considered experimental and not ready for production use.

The solution we have developed for BIRD is independent of BIRD both in code and in design and may be used—and hopefully, will be—independently by other projects. Our primary concern was however successful operation of NetBufs in BIRD, and NetBufs require a lot of refactoring and documentation before they are mature enough for production use. It would be interesting to re-implement them without the separate CBOR encoder and decoder layer, which would probably yield yet faster implementation and would lead to simpler code.

As the ultimate goal is to use NetBufs in the production version of BIRD, and switching directly would be a risk, the NetBufs version of the client and the legacy protocol will coexist for several years. To allow for the coexistence, the BIRD code which deals with CLI, and which happens to touch a critical part of BIRD, the I/O processing, will need to be refactored significantly. Along with these refactorings, NetBufs will gradually be employed to encode the various data structures that BIRD sends.

The source code of NetBufs and a patched version of BIRD are available on the CD attached to this work.

Bibliography

[1] C. Borman et al.: Concise Binary Object Representation (CBOR). RFC 7049. The Internet Society, 2013.

[2] The BIRD team (May 15 2017). BIRD Programmer's Documentation. Retrieved from `http://bird.network.cz/?get_doc&f=prog.html`.

[3] Bray, T. et al.: The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159. Internet Engineering Task Force, 2014.

[4] Google Developers (July 10, 2017). Protocol Buffers documentation. Retrieved from `https://developers.google.com/protocol-buffers`.