B4M36ESW: Efficient software

Lecture 6: Data structure serialization, Remote Procedure Calls

Michal Sojka
michal.sojka@cvut.cz

CZECH TECHNICAL UNIVERSITY IN PRAGUE

June 23, 2025

Outline

Outline

Communication between programs

- Over network
  - Communication protocol (e.g. over TCP or HTTP)
  - Structured data → serialization (JSON, protobuf, ...)
  - Remote Procedure Call (RPC)
    1. Serialize the procedure name and arguments
    2. Send the request and wait for a response
    3. Deserialize the reponse
  - Remote Method Invocation (RMI) – almost the same as RPC
  - HTTP-based API – conceptually similar
- On local host
  - Single address space (threads)
    - Data structures in memory
    - Language type system helps you to avoid mistakes!
  - Different address spaces (processes)
    - Same as "over network"
    - Ideally zero-copy via shared memory
      In Linux: `shm_open(...)` and `mmap()`

Outline

Outline

XML

- eXtensible Markup Language

```xml
<employees>
    <employee>
        <firstName>John</firstName> <lastName>Doe</lastName>
    </employee>
    <employee>
        <firstName>Anna</firstName> <lastName>Smith</lastName>
    </employee>
    <employee>
        <firstName>Peter</firstName> <lastName>Jones</lastName>
    </employee>
</employees>
```

- Very high overhead (both size and computation)
- Complex parser

Outline

JSON

- JavaScript Object Notation

```
{"employees":[
    { "firstName":"John", "lastName":"Doe" },
    { "firstName":"Anna", "lastName":"Smith" },
    { "firstName":"Peter", "lastName":"Jones" }
]}
```

- lower overhead, simpler parser

json-c parser

https://github.com/json-c/json-c

```
#include <json.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  struct json_tokener *tok = json_tokener_new();
  char buf[1024*1024];
  struct json_object *jobj;

  FILE *f = fopen("test.json", "r");

  do {
    size_t len = fread(buf, 1, sizeof(buf), f);
    jobj = json_tokener_parse_ex(tok, buf, len);
  } while (json_tokener_get_error(tok) == json_tokener_continue);
  fclose(f);
  return 0;
}
```

Profiling json-c

47 MB JSON file

- `perf stat ./bench-json-c`

```
 Performance counter stats for './bench-json-c':

      3001.802390      task-clock (msec)         #    0.974 CPUs utilized
              412      context-switches          #    0.137 K/sec
                5      cpu-migrations            #    0.002 K/sec
          478,891      page-faults               #    0.160 M/sec
    9,368,533,705      cycles                    #    3.121 GHz
    3,377,028,216      stalled-cycles-frontend   #   36.05% frontend cycles idle
   14,910,459,852      instructions              #    1.59  insn per cycle
                                                 #    0.23  stalled cycles per insn
    3,144,829,442      branches                  # 1047.647 M/sec
       31,808,151      branch-misses             #    1.01% of all branches

      3.082290868 seconds time elapsed
```

- `perf record --freq 10000 -e cycles ./bench-json-c`

```
21.28%  bench-json-c  bench-json-c       [.] json_tokener_parse_ex
10.67%  bench-json-c  bench-json-c       [.] _int_malloc
 9.28%  bench-json-c  bench-json-c       [.] _IO_vfscanf_internal
 4.30%  bench-json-c  bench-json-c       [.] __libc_calloc
 3.37%  bench-json-c  bench-json-c       [.] ____strtod_l_internal
 3.30%  bench-json-c  bench-json-c       [.] __memset_sse2_unaligned_erms
 3.05%  bench-json-c  [kernel.kallsyms]  [k] clear_page_c_e
 2.60%  bench-json-c  [kernel.kallsyms]  [k] page_fault
```

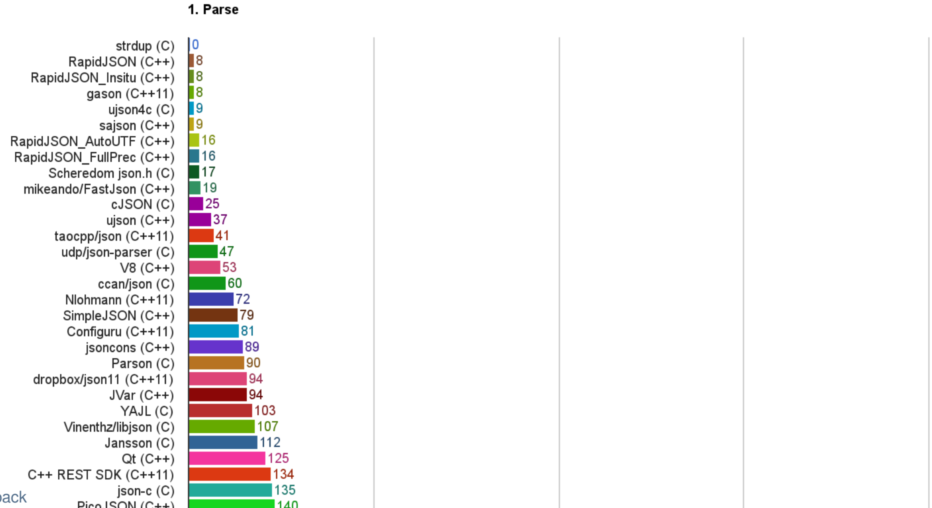
Where is time spent in `json_tokener_parse_ex`?

- Run `perf report` and let it *annotate* the function
- It shows C code intermixed with assembly and the percentage of profiling samples for each instruction

```
     |             while (isspace((int)c)) {
0.21 |         movsbq %dl,%rax
0.07 |         testb  $0x20,0x1(%rcx,%rax,2)
7.08 |       ↓ je     361
0.29 |         xchg   %ax,%ax
     |               if ((!ADVANCE_CHAR(str, tok)) || (!PEEK_CHAR(c, tok)))
0.02 | 330:    mov    0x20(%rbx),%eax
```

JSON benchmark

https://github.com/miloyip/nativejson-benchmark



Trying RapidJSON

- It's C++ ⇒ nicer to read syntax
- bench-rapidjson.cpp →

```
#include <rapidjson/document.h>
#include <rapidjson/filereadstream.h>
using namespace rapidjson;

int main(int argc, char *argv[]) {
  FILE* fp = fopen("test.json", "r");
  char readBuffer[1024*1024];
  FileReadStream is(fp, readBuffer, sizeof(readBuffer));
  Document d;
  d.ParseStream(is);
  fclose(fp);
  return 0;
}
```

- perf stat bench-rapidjson

```
 Performance counter stats for './bench-rapidjson':

        389.890403      task-clock (msec)         #    0.998 CPUs utilized
                12      context-switches          #    0.031 K/sec
                 0      cpu-migrations            #    0.000 K/sec
            43,392      page-faults               #    0.111 M/sec
     1,106,686,422      cycles                    #    2.838 GHz
       206,781,432      stalled-cycles-frontend   #   18.68% frontend cycles idle
     2,467,762,722      instructions              #    2.23  insn per cycle
                                                  #    0.08  stalled cycles per insn
       593,437,567      branches                  # 1522.063 M/sec
            61,403      branch-misses             #    0.01% of all branches

       0.390790908 seconds time elapsed
```

What about spaces?

- perf record/report

```
 23.66%  bench-rapidjson    [.] rapidjson::GenericReader<...>::ParseString<0u, ra...
 22.43%  bench-rapidjson    [.] rapidjson::GenericReader<...>::ParseValue<0u, rap...
 18.94%  bench-rapidjson    [.] rapidjson::GenericReader<...>::ParseNumber<0u, ra...
 11.66%  bench-rapidjson    [.] rapidjson::SkipWhitespace<rapidjson::FileReadStream>
  5.70%  libc-2.24.so       [.] __memmove_sse2_unaligned_erms
  2.75%  bench-rapidjson    [.] rapidjson::GenericDocument<rapidjson::UTF8<char>, rapidjson::MemoryPoolAllocator<rapidjson::CrtAllocator>, rapidjs
  1.96%  [kernel.kallsyms]  [k] page_fault
  1.68%  [kernel.kallsyms]  [k] clear_page_c_e
```

- perf annotate rapidjson::GenericReader<...>::ParseString...

```
       |                     Ch c = is.Peek();
       |                     if (RAPIDJSON_UNLIKELY(c == '\\')) {    // Escape
 12.22 |  96:   cmp    $0x5c,%r14b
       |      ↓ je     178
       |                             TEncoding::Encode(os, codepoint);
       |                         }
       |                         else
       |                             RAPIDJSON_PARSE_ERROR(kParseErrorStringEscapeInvalid, escapeOffset);
       |                     }
       |                     else if (RAPIDJSON_UNLIKELY(c == '"')) {    // Closing double quote
  6.01 |        cmp    $0x22,%r14b
       |      ↓ je     200
       |                         is.Take();
       |                         os.Put('\0');   // null-terminate the string
       |                         return;
       |                     }
```

- What is RAPIDJSON_UNLIKELY?
  Branch predition hint (see `__builtin_expect()` in gcc manual)

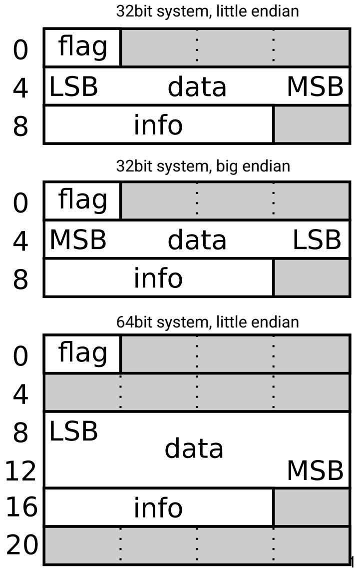
Outline

Raw memory

- Sending/receiving directly the content of memory:

```
struct data {
    char flag;
    long int data;
    char info[3];
};

void sendData(struct data &d) {
    send(sock, &d, sizeof(d));
}
void recvData(struct data &d) {
    recv(sock, &d, sizeof(d));
}
```



Raw memory

Problems & solutions

- `int` doesn't have fixed size ⇒ `#include <stdint.h>` ⇒ `int32_t`
- endianing ⇒ `#include <endian.h>` ⇒ `htole32()` etc.
  (host to little-endian 32 bits)
- padding ⇒ `__attribute__((__packed__))`
-

```
struct __attribute__ ((__packed__)) data {
        char flag;
        int32_t data;
        char info[3];
};

void recvData(struct data &d) {
        struct data dd;
        recv(sock, &dd, sizeof(dd));
        d = dd;
        // deserialization
        d.data = le32toh(dd.data); // little-endian 32b. to host
}
```

0	flag	LSB data
4	MSB	info

Raw memory

Properties

- Blazingly fast, but inflexible
- The receive side must know the format of data
  - What if the sender uses newer version of the data structure than the receiver?
    - e.g. a field added/removed, type changed

Versioning of the protocol

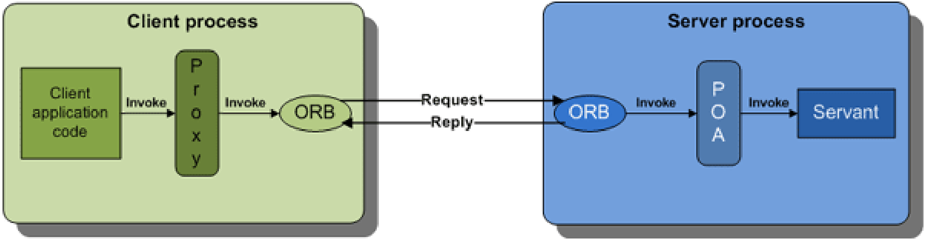
Whenever you design a communication protocol, always include a version number (or feature flags) to allow older and newer versions to coexist. It saves you some pain when upgrading your software, especially in the cloud.

Outline

Outline

Common Object Request Broker Architecture (CORBA)

- Language independent “RPC framework” from 1990
- Interface Description Language (IDL)
- Automatic generation of (de)serialization code (IDL compiler)
- Description of the data structure is not normally sent with the data
- CORBA is **not very popular today**, perhaps because of its complexity and difficulty of using parts of it (such as CDR – see later) independently
- Many of its core technologies/mechanisms were designed correctly and a lot of people **reinvent the wheel** today.



Interface Description Language (IDL)

- Called “schema” in other frameworks
- Defines only data types and interfaces
- IDL compiler generates corresponding definitions in target language as well as conversion code to/from the *Common Data Representation* (CDR) form.

Example

```
module Finance {
  typedef sequence<string> StringSeq;
  struct AccountDetails {
    string     name;
    StringSeq  address;
    long       account_number;
    double     current_balance;
  };
  exception insufficientFunds { };
  interface Account {
    void deposit(in double amount);
    void withdraw(in double amount) raises(insufficientFunds);
    readonly attribute AccountDetails details;
  };
};
```

Common Data Representation (CDR)

- Defines “wire” representation of data (as it appears on network)
- Most today’s serialization schemes converge to something similar
- Endian
  - Data is sent in sender’s endian
  - Message header contains information about the used endian
  - $\Rightarrow$ No expensive endian conversion between similar hosts
- Data padding as in memory – efficient (de)serialization
- TypeCodes – CDR representation of any IDL data type
  - Allows to send Any data type (TypeCode + actual data) and the receiver can reconstruct it

Outline

Google Protocol Buffers (protobuf)

https://developers.google.com/protocol-buffers/

- Data description – conceptually similar to IDL
- Automatic code generation
- Partial description of data sent with the data
  - Less problems with protocol versioning
- Easy to use API
- Supports multiple languages: Java, Python, C++, C#, ...

```
syntax = "proto3";

message SearchRequest {
  string query = 1;
  int32  page_number = 2;
  int32  result_per_page = 3;
}
```

- Numbered “tags” uniquely identify fields
- Tags also help with maintaining backward compatibility (versioning)

Wire encoding

- Key-value pairs
- Key = the tag (field number) + type information
  - ⇒ unknown keys (and their values) can always be skipped

Type	Meaning	Used For
0	Varint	int32, int64, uint32, uint64, sint32, sint64, bool, enum
1	64-bit	fixed64, sfixed64, double
2	Length-delimited	string, bytes, embedded messages, packed repeated fields
3	Start group	groups (deprecated)
4	End group	groups (deprecated)
5	32-bit	fixed32, sfixed32, float

- Key encoding: `(field_number << 3) | wire_type`
  - Stored as Varint (see next slide)

Wire encoding – Varint

- Encoded in variable number of bytes, small numbers take only one byte
- 7^{th} bit is 1 in all but last byte.
- Bits 0–6 store the value.
- Examples:
  - 9 = 0000 1001b → 0000 1001b
  - 300 = 1 0010 1100b → 1010 1100 0000 0010
- Signed integers (sint) use Zigzag encoding (i.e., numbers close to zero are encoded into a single byte):
  - `n` is encoded as `(n << 1) ^ (n >> 31)`

$$\begin{array}{rcc} 0 & \rightarrow & 0 \\ -1 & \rightarrow & 1 \\ 1 & \rightarrow & 2 \\ -2 & \rightarrow & 3 \end{array}$$

- Varint represents a trade-off between size of the encoded data and speed of encoding/decoding.

Wire encoding – String and Message

- Varint-encoded length + bytes of string/message

Example

- ```
  message Test2 {
    required string b = 6;
  }
  ```
- b = “testing”
- Encoded as (hex):
  32 07 74 65 73 74 69 6e 67
- 32h = `(6 << 3) | 2  // 6 = tag, 2 = length delimited`
- 07h = length
  ```

Wire encoding – repeated fields

- ```
  message Test4 {
    repeated int32 d = 4 [packed=true];
  }
  ```
- ```
  0x22            // tag (field number 4, wire type 2)
  0x06            // payload size (6 bytes)
  0x03            // first element (varint 3)
  0x8E 0x02       // second element (varint 270)
  0x9E 0xA7 0x05  // third element (varint 86942)
  ```
- When reading, the field can be skipped without decoding all values.

Message streaming

- Parsing code does not know where a message begins and ends
- Solution: Put the length of the message before it

Protobuf example – OpenStreetMap

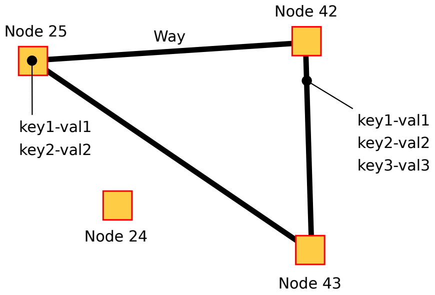
https://wiki.openstreetmap.org/wiki/PBF_Format

```
message Node {
  required sint64 id = 1;
  // Parallel arrays.
  repeated uint32 keys = 2 [packed = true]; // String IDs.
  repeated uint32 vals = 3 [packed = true]; // String IDs.
  optional Info info = 4; // May be omitted in omitmeta
  required sint64 lat = 8;
  required sint64 lon = 9;
}

message Way {
   required int64 id = 1;
   // Parallel arrays.
   repeated uint32 keys = 2 [packed = true];
   repeated uint32 vals = 3 [packed = true];

   optional Info info = 4;

   repeated sint64 refs = 8 [packed = true];  // DELTA coded
}
```



Czech republic: PBF – 670 MB, XML – 16 GB

From .proto to C++

addressbook.proto

```
package tutorial;

message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phones = 4;
}

message AddressBook {
  repeated Person people = 1;
}
```

mycode.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include "addressbook.pb.h" // generated from .proto
using namespace std;

// Iterates though all people in the AddressBook and prints info about them.
void ListPeople(const tutorial::AddressBook& address_book) {
  for (int i = 0; i < address_book.person_size(); i++) {
    const tutorial::Person& person = address_book.person(i);

    cout << "Person ID: " << person.id() << endl;
    cout << "  Name: " << person.name() << endl;
    if (person.has_email()) {
      cout << "  E-mail address: " << person.email() << endl;
    }

    for (int j = 0; j < person.phone_size(); j++) {
      const tutorial::Person::PhoneNumber& phone_number = person.phones(j);

      switch (phone_number.type()) {
        case tutorial::Person::MOBILE:
          cout << "  Mobile phone #: ";
          break;
        case tutorial::Person::HOME:
          cout << "  Home phone #: ";
          break;
        case tutorial::Person::WORK:
          cout << "  Work phone #: ";
          break;
      }
      cout << phone_number.number() << endl;
    }
  }
}
```

Outline

Cap’n’proto

https://capnproto.org/

- Developed by the original author of protobuf
- Some years later – lessons learnt from protobuf
- Very efficient for communication via shared memory (e.g. between different languages)
- Still usable over network
- No de/encoding needed – serialized form is usable as a native form (unless packing is used)
  - ⇒ It’s possible to just `mmap` a file to memory to work with the data.

Cap’n’proto encoding

- Bool: 1 bit
- Integers: Little endian, native size, aligned to multiple of their size (padding)
- Default values: always encoded as zero, i.e. `enc = val ^ default`
  `^` = XOR
- Optional packing = getting rid of zero bytes
  - Set bits in the first byte indicate which of the following 8 bytes are non-zero. The nonzero bytes follow.
  - ```
    unpacked (hex):  08 00 00 00 03 00 02 00  19 00 00 00 aa 01 00 00
    packed (hex):  51 08 03 02   31 19 aa 01
    ```
- Structures: Pointer (= index) to data and sub-structures
    ```

Message + structure encoding

https://capnproto.org/encoding.html

```
struct Person {
  id @0 :UInt32;   # 0xab
  name @1 :Text;   # Alice
  email @2 :Text; # alice@example.com
  phones @3 :List(PhoneNumber);

  struct PhoneNumber {
    number @0 :Text; # "555-1212"
    type @1 :Type;    # mobile

    enum Type {
      mobile @0;
      home @1;
      work @2;
    }
  }

  employment :union {
    unemployed @4 :Void;
    employer @5 :Text;
    school @6 :Text; # MIT
    selfEmployed @7 :Void;
  }
}
```

```
         Message Struct Name Email Phones Employment
00000000 00 00 00 10 00 00 00 00  00 00 00 00 01 00 04 00  |................|
00000010 ab 00 00 00 02 00 00 00  0d 00 00 00 32 00 00 00  |............2...|
00000020 0d 00 00 00 92 00 00 00  15 00 00 00 17 00 00 00  |................|
00000030 25 00 00 00 22 00 00 00  41 6c 69 63 65 00 00 00  |%..."...Alice...|
00000040 61 6c 69 63 65 40 65 78  61 6d 70 6c 65 2e 63 6f  |alice@example.co|
00000050 6d 00 00 00 00 00 00 00  04 00 00 00 01 00 01 00  |m...............|
00000060 00 00 00 00 00 00 00 00  01 00 00 00 4a 00 00 00  |............J...|
00000070 35 35 35 2d 31 32 31 32  00 00 00 00 00 00 00 00  |555-1212........|
00000080 4d 49 54 00 00 00 00 00                           |MIT.....|
00000088
```

- Tree-like data structure. Allows skipping of unknown or unwanted data.
- Packing allows getting rid of all 83 zero bytes above and adds 17 more bytes.

From .capnp to C++

addressbook.capnp

```
struct Person {
  id @0 :UInt32;
  name @1 :Text;
  email @2 :Text;
  phones @3 :List(PhoneNumber);

  struct PhoneNumber {
    number @0 :Text;
    type @1 :Type;

    enum Type {
      mobile @0;
      home @1;
      work @2;
    }
  }

  employment :union {
    unemployed @4 :Void;
    employer @5 :Text;
    school @6 :Text;
    selfEmployed @7 :Void;
    # We assume that a person is only one of these.
  }
}

struct AddressBook {
  people @0 :List(Person);
}
```

mycode.cpp

```
#include "addressbook.capnp.h"
#include <capnp/message.h>
#include <capnp/serialize-packed.h>
#include <iostream>

void printAddressBook(int fd) {
  ::capnp::PackedFdMessageReader message(fd);

  AddressBook::Reader addressBook = message.getRoot<AddressBook>();

  for (Person::Reader person : addressBook.getPeople()) {
    std::cout << person.getName().cStr() << ": "
              << person.getEmail().cStr() << std::endl;
    for (Person::PhoneNumber::Reader phone: person.getPhones()) {
      const char* typeName = "UNKNOWN";
      switch (phone.getType()) {
        case Person::PhoneNumber::Type::MOBILE: typeName = "mobile"; break;
        case Person::PhoneNumber::Type::HOME: typeName = "home"; break;
        case Person::PhoneNumber::Type::WORK: typeName = "work"; break;
      }
      std::cout << "  " << typeName << " phone: "
                << phone.getNumber().cStr() << std::endl;
    }
    Person::Employment::Reader employment = person.getEmployment();
    switch (employment.which()) {
      case Person::Employment::UNEMPLOYED:
        std::cout << "  unemployed" << std::endl;
        break;
      case Person::Employment::EMPLOYER:
        std::cout << "  employer: "
                  << employment.getEmployer().cStr() << std::endl;
        break;
      case Person::Employment::SCHOOL:
        std::cout << "  student at: "
                  << employment.getSchool().cStr() << std::endl;
        break;
      case Person::Employment::SELF_EMPLOYED:
        std::cout << "  self-employed" << std::endl;
        break;
    }
  }
}
```

FlatBuffers

`https://flatbuffers.dev/`

- Similar to Cap’n’Proto (access to serialized data without parsing/unpacking)
- From Google

Outline

Apache Avro

- Schema in JSON
- Schema handshake after connection establishment
- No tags in data, because the schema is known to all parties
- File storage
  - Compression
  - Blocks allowing skip through the data without deseralizing them

Conclusion

- Data serialization format trade-offs:
  - Human readability/ease of data manipulation
  - Data size
  - (De)serialization speed
- Select the right technology for your needs