Pack ein pack aus

SERIALISIERUNG IN C++

Inhalt

- Was ist Serialisierung?
- Formate und Schemata
- Übersicht Libraries
- Benchmarks und Beispielcode
- Ergebnisse / Empfehlungen

Was ist Serialisierung?

- “Abbildung von strukturierten Daten in eine sequentielle Darstellungsform” (Wikipedia)

- Anwendung: kodieren von Objekten zur
  - Speicherung im Dateisystem
  - Übertragung per Netzwerk
  - [Drucken, Debuggen, Logging, …]

- Deserialisieren: einlesen von Daten in interne Repräsentation

Was ist Serialisierung?

- Wichtige Anforderungen
  - Standardisierung
  - Eindeutiges Schema (implizit oder explizit)
    - Langzeitspeicherung: explizites Schema!
  - Portabilität
    - Sprache, Datums/Zeitangaben
    - Zeichenkodierung, Datentypgrößen, Byte-order …
  - Platzbedarf
  - Serialisierungsgeschwindigkeit
  - Deserialisierungsgeschwindigkeit

} oft ein Trade-off

Old C Style: MEMCPY (DON'T TRY THIS AT HOME!)

```
struct MyData {
  int x;
  int64_t y;
  const char* boom;
}

void serialize(void** out, size_t* outsize,
               const MyData * const in)
{
  *outsize = sizeof(size_t)
           + sizeof(MyData);

  *out = malloc(*outsize);
  memcpy(*out, outsize, sizeof(size_t));

  memcpy(((char*)*out)+sizeof(size_t),
         in, *outsize-sizeof(size_t));
}
```

```
void deserialize(MyData* out,
                 const size_t insize,
                 const void * const in)
{
  size_t size_from_data = *((size_t*)in);

  memcpy(out, ((char*)in)+sizeof(size_t),
         size_from_data-sizeof(size_t));
}
```

MEMCPY – Probleme

- Pointer !!! Vererbung und Vtables
- Logik in Konstruktoren / Destruktoren
- manuelle Größen- / Grenzenchecks
- Byte-order
- String Kodierungen
- […]


Every time you use memcpy for serializing data
God kills a kitten
Please, think of the kittens!

Frameworks / Libraries

- “Big Company Frameworks”
  - Google Protobuf
  - Apache (Facebook) Thrift
  - Microsoft Bond
- “Native C++”
  - Boost Serialization
  - Cereal
- “Neu / Speziell”
  - Cap’n’proto
  - AVRO
  - Flatbuffers (gzip)
- Referenz
  - ASN.1 BER
  - [XML / EXI]

Interessante Merkmale …

- Text vs. Binary
- schemabasiert vs. schemalos
- standard/open-source vs. proprietär
- unterstützten Sprachen
- Anwendung:
  - Plattenspeicher (Archivierung) vs. Netzwerk/Kommunikation
  - langsame/fehleranfällige vs. schnelle/stabile Verbindung

- Code: Kompiliert vs. Interpretiert, Validierung
  - Codegenerierung oder manuelles Parsing / dynamische Validierung?

- Framework Umfang – “nur” Serialisierung vs. Netzwerk / RPC Funktion

Kodierungen / Formate

- Textformate
  - Flach: CSV, “Tables” oder “Record” orientiert – Problem für Objekte !
  - Baumstruktur: XML, JSON
- Binärformate
  - Standards
    - ASN.1 BER / PER
    - XML EXI
  - Encodings:
    - Text - UTF-8, ASCII, weitere Codecs?
    - Integer - feste vs. variable Länge, “Endianness”
    - Float Darstellungen
    - Längenangaben
  - Andere i.d.R. ähnlich aufgebaut
    - “TLV” – Tag Length Value

Schema / Metadaten

- “Was ist hier gespeichert”?
- Inline
  - Basistypen + “inline” mit Daten definierte Strukturen
  - Vorteil: Datenstrom “standalone” eindeutig interpretierbar, “dynamisch”
  - Nachteil: redundante Information
- Extern
  - z.B. XML Schema Links, IDLs für Serialisierungsframeworks
  - Vorteil: kompaktere Nutzdaten
  - Nachteil: Schema muss bekannt sein (ID)

- Tagged: “Type Tag” (Verweis auf Schema) an jedem Datum
  - unvollständige Schemata / Versionierung
- Untagged: Struktur vollständig in Schema – sehr kompakt!

Versionierung / “Schema Evolution”

- Tagged vs. untagged, inline vs. externs Schema
  - z.B. Apache Thrift

```
message Person {
    1: required i64         id;
    2: optional string      name;
    3: list<string>         telefon;
}
```

- Evolution:
  - Renaming, neue Felder i.d.R OK
  - Reihenfolge: Tags!
- “required” Attribute sind “für immer”
- Ohne Tags / inline Schema: Versionierung manuell !

Schemadarstellungen

ASN.1 Schema

```
MixedData DEFINITIONS ::=
BEGIN

Enum1 ::= ENUMARATED {
     ONE (1), TWO (2)
}

Mixed::= SEQUENCE {
     id        INTEGER,
     int1      [3] INTEGER,
     int2      [4] INTEGER,
     uint1     INTEGER (0..65535),
     float1    REAL OPTIONAL,
     text1     OCTET STRING SIZE(32),
     enum1     Enum1
}
END
```

Microsoft Bond IDL

```
namespace uti.serialize;

enum Enum1 {
     ONE, TWO
}

struct mixed {
     0: int32    id;
     1: int32    int1;
     2: int64    int2;
     3: uint32   uint1;
     4: double   float1;
     5: string   text1;
     6: Enum1    enum1 = ONE;
}
```

Benchmarks - Setup

- Alle Benchmarks mit Google Benchmark
  - Speicher soweit möglich vorallokiert
  - Zu serialisierende Objekte “wiederverwendet”
  - Encoding / Decoding durchläuft
    - Für “memory mapped” Formate: einfache Zugriffe / Validierung
- Gemessene Werte:
  - Nachrichten / Sekunde
  - xB/s (kodiert → bei besserer Kompression weniger Daten!)
  - Durchschnittliche Nachrichtengröße

Benchmarks - Szenarien

Klein - nur Integer

```
struct ints_t {
    int64_t id;
    int64_t int1;
    int64_t int2;
    int32_t int3;
    int32_t int4;
};
```

Gemischt

```
struct mixed_t {
    int32_t id;
    int32_t int1;
    int64_t int2;
    uint32_t uint1;
    double float1;
    std::string text1;
    ENUMERATION enum1;
};
```

Komplex

```
struct complex_t {
    int64_t id;
    std::string text1;
    std::string text2;
    std::string text3;
    int32_t int1;
    int32_t int2;
    int32_t int3;
    std::string text4;
    ENUMERATION enum1;
    double float1;
    int64_t int4;
    uint32_t uint1;
    uint64_t uint2;
    int64_t int5;
    int32_t int6;
    double float2;
    std::string text5;
};
```

Benchmarks - Szenarien

Vector

```
struct container_vec_t {
    int32_t id;
    int64_t int1;
    std::string text1;
    std::vector<std::string> stringvec;
    std::vector<int64_t> intvec;
};
```

Map

```
struct container_map_t {
    int32_t id;
    int64_t int1;
    std::string text1;
    std::map<int32_t, std::string> map1;
}
```

Benchmarks - Disclaimer

1. Alle Benchmarks sind “künstlich” – Anwendung macht “nichts anderes”
2. Nicht nur Geschwindigkeit bewerten - Qualität/API, Doku, Stabilität, Verbreitung / Standards
   - Falsch benutzte API, Unterschiede → deutlich andere Ergebnisse
3. Möglicherweise messen Benchmarks im vor allem malloc/memcpy!
4. „Synthetische“ Daten
   - Gut komprimierbar
   - Einfluss Datengenerierung
     - → Nicht optimiert (z.B. std::to_string), aber überall gleich
5. Testhardware: 2 CPU Xeon E5 3.2 GHZ, 2GB RAM VM, Redhat EL 7

Microsoft Bond

- Internes Microsoft Projekt (z.B. für Azure), Open Source 12/2015
- Neues "modernes" C++ Framework
  - erweiterbare modulare Architektur (Protokolle, Codegen)
  - Fast/Compact Binary und JSON Protokolle
  - "Tagged" (fast binary, compact v1) oder "untagged" (simple binary, compact v2) Protokolle
  - Nicht so verbreitet wie Thrift/Protobuf
- IDL ähnlich Protobuf/Thrift, Compiler in Haskell
- Sprachen: C++, C#, Python (momentan kein Java!)
- "Specials": Transcoder, Runtime Schema (lazy deserialization), "Bond Comm" (momentan nur CS)

https://github.com/Microsoft/bond/

Bond Beispiel

DEMO !

(+ Google Benchmark)

Google Protobuf

- “altes Framework” (frühe 2000er), Google intern sehr verbreitet
  - Aktuelles Release: v3 – Allocator Optimierung (im Test nicht relevant)
- Einfache IDL, Compiler für C++, C#, Java, Python, Go
- Nur ein Binärformat, proprietät (~compact binary)
  - Textformat für Debugging
- Umfangreich ge-reviewed und getestet: sicher und stabil
- Aktuelle Entwicklung: RPC per GRPC / HTTP2

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

Protobuf Beispiel

IDL

```
syntax = "proto3";
option cc_enable_arenas = false;
package serialize;

enum ENUMERATION {
  ONE = 0; TWO = 1;
}
[…]
message Container_Vec {
  sint32 id = 1;
  sint64 int1 = 2;
  string text1 = 3;
  repeated string stringvec = 4;
  repeated int64 intvec = 5;
}
message Container_Map {
  sint32 id = 1;
  sint64 int1 = 2;
  string text1 = 3;
  map<sint32, string> map1 = 4;
}
```

Serialization Code

```
serialize::Container_Vec obj;

unsigned char* charBuf = buffer;
bool res;

for (int i = 0; i < count; ++i)
{
  obj.Clear();
  obj.set_id(i);
  obj.set_int1(int_val++);
  obj.set_text1(string_short);
  for (int j = 0; j < cont_count; ++j)
  {
    obj.add_stringvec(string_short);
    obj.add_intvec(int_val);
  }
  uint16_t size = obj.ByteSize();
  *reinterpret_cast<uint16_t*>(buf) = htons(size);
  res = obj.SerializeToArray(buffer+2, bufSize);
  buffer += size + 2;
} // or repeated / union types for complex msgs.

// deserialize:
res = obj.ParseFromArray(buffer+2, size);
```

Cap'n'proto

- Author früher Protobuf (v2) Entwickler → Redesign "from scratch"
- Ziel: In-Memory Repräsentation == Serialisierungsformat
  - Kompaktes Format: schnelle formatspezifische Kompression
  - Lesen: memcpy + Validierung
  - OO Zugriff: getter/setter Methoden (→ Protobuf)
  - Reader / Builder Konzepte → separiert Logik von Datenstrukturen

- C++ Introspection / dynamisches Schema
- Interessante RPC Implementierung → "Pipelining" von Requests
- Version 0.5.3, aber schon im Produktiveinsatz
  - weitere Sprachen: externe Github Projekt (Java, C#, JS, Rust, Python, …)

https://capnproto.org/

Cap'n'Proto Beispiel

IDL

```
@0xc95f38542f78fdf9;

using Cxx = import "/capnp/c++.capnp";
$Cxx.namespace("capnp_bench");

enum ENUMERATION {
   one @0; two @1;
}

struct Mixed {
   id    @0 : Int32;
   int1  @1 : Int32;
   int2  @2 : Int64;
   uint1 @3 : UInt32;
   float1 @4 : Float64;
   text1 @5 : Text;
   enum1 @6 : ENUMERATION;
}

struct MixedList {
   mixed @0 : List(Mixed);
}
```

Serialization Code

```
size_t genContVec(kj::OutputStream& out,
                  unsigned count) {
  capnp::MallocMessageBuilder builder;
  auto msg = builder.initRoot<
             capnp_bench::ContainerVecList>();
  auto cont_builder = msg.initContVec(count);
  for (unsigned i = 0; i < count; ++i)
  {
      auto current = cont_builder[i];
      current.setId(i);
      current.setInt1(int_val++);
      current.setText1(string_short);
      auto strvec = current.initStringvec( 50 );
      auto intvec = current.initIntvec( 50 );
      for (unsigned j = 0; j < 50; ++j)
      {
         strvec.set(j, string_short);
         intvec.set(j, int_val);
      }
   }
   capnp::writePackedMessage(out, builder);
   return capnp::computeSerializedSizeInWords
                 (builder) * sizeof(capnp::word);
}
```

Cap'n'Proto Beispiel

Deserialization

```
void deserialize(benchmark::State& st, kj::ArrayPtr<kj::Byte>& buffer, size_t bufsize) {
  capnp::MallocMessageBuilder builder;
  capnp::ReaderOptions readerOpts;
  readerOpts.traversalLimitInWords = MAX_BUF_SIZE * 8;

  while (st.KeepRunning()) {
    kj::ArrayInputStream in(buffer);
    capnp::PackedMessageReader messages(in, readerOpts);

    // generic reader -> setRoot will copy the messages and validate the structure
    out.setRoot(messages.getRoot<capnp::AnyPointer>());

    for (ContainerVec::Reader reader : out.getContVec()) {
      // not from benchmark …
      mytext1 = reader.getText1().cStr();

      // dynamic / introspection based usage:
      DynamicValue::Reader dyn = reader.getAs<DynamicStruct>();
      for (auto field : dyn.getSchema().getFields()) {
        if (dyn.has(field) && field.getProto().getName().cStr() == "text1")
          mytext1 = field.as<Text>.cStr();
  } } }
  // […] process bechmark measurements
}
```

Flatbuffers

- Google Projekt als Ersatz für Protobuf bei Spezialanwendungen:
  - Spieleentwicklung, Mobile Kommunikation
  - Vgl. Protobuf: einfachere IDL, “Table” für Datentypen
  - „Allocator in flachen Buffer”
  - Interface ist etwas “unhandlich”
- In-Memory == Protokoll wie Cap’n’proto, schnelle Leseperformance
  - Bandbreiteneffizienz durch standard (externe) Kompression
  - In Benchmark zlib, besser Brotli? Langsamer als C‘n‘P Packed!
- Sprachen: C++, Java, C#, Go, Python, JS, PHP
  - RPC per “GRPC”

https://google.github.io/flatbuffers/

Flatbuffers Beispiel

Serialization Code

```
void generateContainerVec(flatbuffers::FlatBufferBuilder& builder, size_t count) {

  auto vecbuf = reinterpret_cast<flatbuffers::Offset<ContainerVec>*>(global_buffer.data());
  std::vector<flatbuffers::Offset<flatbuffers::String>> stringvec;

  for (size_t i = 0; i < count; ++i) {
    ContainerVecBuilder current(builder);
    stringvec.clear(); // reuse buffers

    current.add_id(static_cast<int>(i));
    current.add_int1(int_val++);
    auto text1 = builder.CreateString(string_short.c_str(), string_short.size());
    current.add_text1(text1);
    for (int j = 0; j < cont_count; ++j) {
      // […] intvec left out for brevity
      stringvec.push_back(builder.CreateString(string_short.c_str(), string_short.size()));
    }
    auto stringvec_flat = builder.CreateVector(stringvec);
    current.add_stringvec(stringvec_flat);
    vecbuf[i] = current.Finish();
  }
  auto contvec_vec = builder.CreateVector(vecbuf, count);
  auto root = CreateRoot(builder, 0, 0, 0, contvec_vec);
  FinishRootBuffer(builder, root);
}
```

Flatbuffers Beispiel

IDL

```
file_identifier "benc";
file_extension "fb_bench";
namespace flatbuf_bench;

enum ENUMERATION : byte { ONE = 0, TWO }

table Root {
  ints: [Ints];
  […]
  cvec: [ContainerVec];
  cmap: [ContainerMap];
}
[…]

table ContainerVec {
   id: int;
   int1: long;
   text1: string;
   stringvec: [string];
   intvec: [long];
}
```

Setup Code / Deserialize

```
void deserialize(uint8_t* buf,
                 size_t bufsize) {

  auto verifier = flatbuffers::Verifier(
                   buf, bufsize);

  if (VerifyRootBuffer(verifier)) {
    auto root = GetRoot(buf);
    long long int counter = 0;
    for (const ContainerVec* ptr
           : *root->cvec())
    {
      counter += ptr->id();
    }
  } else {
    throw std::runtime_error("Error
            parsing Root Buffer");
  }
}
```

Apache Thrift

- Entwickelt von Facebook, an OSS Apache Projekt “übergeben”
  - Kompletter Netzwerk Stack: Server, Processor, Protocol, Transport Layers
- IDL / Library für sehr viele Sprachen verfügbar
  - C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, JavaScript, Node.js, Smalltalk, OCaml and Delphi …
- Formate modular: JSON, Compact Binary, Fast Binary
- Dokumentation oft veraltet / unvollständig

https://thrift.apache.org/

Thrift Beispiel

IDL

```
namespace cpp thrift_serialize

enum ENUMERATION {
    ONE = 0, TWO = 1
}
[…]

struct container_vec {
    1: i32 id,
    2: i64 int1,
    3: string text1,
    4: list<string> stringvec,
    5: list<i64> intvec
}

struct container_map {
    1: i32 id,
    2: i64 int1,
    3: string text1,
    4: map<i32, string> map1
}
```

Setup Code / Deserialize

```
while (st.KeepRunning())
{
  auto buf =
    make_shared<transport::TMemoryBuffer>(
      raw_buf, raw_size,
      transport::TMemoryBuffer::OBSERVE);

  auto tp =
    make_shared
     <transport::TBufferedTransport>(buf);

  auto proto =
    make_shared
     <protocol::TCompactProtocol>(tp);

  for (auto& obj: result)
  {
    bytes_read += obj.read(proto.get());
  }
}
```

Thrift Beispiel

Serialization / Objektzugriff

```
size_t genContVec(protocol::TProtocol* out, int count) {
  thrift_serialize::container_vec current;
  int int_val = 123456, cont_count = 50, generated = 0;

  for (int i = 0; i < count; ++i)
  {
    current.stringvec.clear();
    current.intvec.clear();
    current.id = i;
    current.int1 = int_val++;
    current.text1 = string_short;
    for (int j = 0; j < cont_count; ++j)
    {
      current.stringvec.push_back(string_short);
      current.intvec.push_back(int_val);
    }
    generated += current.write(out);
   }
   out->getTransport()->flush();
   return generated;
}
```

Boost Serialization

- Serialisierung aus Boost Library
- Keine IDL, C++ Klassen/Structs direkt verwendbar
  - nur C++ !
- serialize<ARCHIVE>() als Member (intrusive)
  - überladen von boost::serialization::serialize (nicht intrusive)
  - ARCHIVE Typ (input / output) entscheidet "Richtung" und Format
  - oder save() / load()
- Formate: Binary, XML, Text - erweiterbar
  - Binary ist NICHT PORTABEL (z.B. endianness)
  - Versionierung manuell

http://www.boost.org/doc/libs/release/libs/serialization/

Cereal

- Ersetzt / erweitert boost::serialization
  - Datenstrukturen und serialize() *IDENTISCH* zu Boost
  - Formatwahl per boost::archive / cereal::*Archive Template Argument
- C++11 header only Interface (keine Lib!)
  - Fast alle STL Typen “out-of-the-box”, statis_assert, Introspection
  - Achtung: “binary.hpp” nicht portable, „portably_binary.hpp“ benutzen
- Kompaktere Formate als Boost
  - Compact Binary, XML, JSON
- Verzicht auf Metainfos (z.B. Versionierung), kein “Pointer Tracking”

http://uscilab.github.io/cereal/

Boost Serialization / Cereal Beispiel

DEMO !

(weil zu viel Code für Folien ☺)

boost_serialize.cpp

cereal_serialize.cpp

dataset.h

cereal/serialize.h

Avro

- Apache Projekt speziell für gute Integration dynamischer Sprachen
  - Schnelle JS, Python & Co. Implementation
- Untagged Format, Schema wird vorausgesetzt
  - Schema kann implizit möglich, IDL / Codegenerierung optional
- Sehr effizientes Wire-Format
- Sprachen: C++, Java, C#, PHP, Python, Ruby, JS
  - Implementierung meist Java

https://avro.apache.org/

Avro Beispiel

IDL (JSON)

```
{"name": "root", "type": "record",
"fields": [
{"name": "cont_vec_vec", "type":
  ["null", {"type": "array", "items": {
    "type": "record",
    "name": "container_vec",
    "fields": [
      {"name": "id", "type": "int"},
      {"name": "int1", "type": "long"},
      {"name": "text1", "type": "string"},
      {"name": "stringvec", "type":
       {"type": "array","items": "string"}
      },
      {"name": "intvec", "type":
       { "type": "array","items": "long"}
      }
    ]
} }] },
[…]
] }
```

Deserialize

```
void deserialize(uint8_t* buf,
                 size_t size) {
  avro::ValidSchema sch =
    avro::compileJsonSchemaFromFile(
            schema_path);

  auto decoder = avro::jsonDecoder(sch);
  // auto decoder = avro::binaryDecoder();

  unique_ptr<avro::InputStream> in =
    avro::memoryInputStream(buf, size);
  avro_bench::root root;

  decoder->init(*in);
  avro::decode(decoder, root);
  int count = 0;
  for (auto& cont_vec : root.cont_vec_vec
                            .get_array()) {
    count += cont_vec.int1;
  }
}
```

Avro Beispiel

Serialization / Objektzugriff

```
void runSerialize(size_t count) {
  auto encoder = avro::binaryEncoder();
  std::unique_ptr<avro::OutputStream> out = avro::memoryOutputStream(MAX_BUF_SIZE);

  std::vector<avro_bench::container_vec> contvec(count);
  encoder.init(*out);
  avro_bench::root root;
  for (size_t i = 0; i < count; ++i) {
    auto& current = contvec[i];
    current.stringvec.clear();
    current.id = static_cast<int>(i);
    current.int1 = int_val++;
    current.text1 = string_short;
    for (size_t j = 0; j < cont_count; ++j) {
      current.stringvec.push_back(string_short); // intvec left out for brevity
  } }
  root.ints_vec.set_null();
  […]
  root.cont_vec_vec.set_array(contvec);
  avro::encode(encoder, root);
  encoder.flush();
  return out->byteCount();
}
```

ASN.1 / BER

- Alter Standard – Telko / Security Industrie
  - komplex und fehleranfällig!
    - Z.B. 3 Varianten für Längenkodierungen
  - Spezielle Datentypen (TBCD, T61String, Ia5String …)

- ASN.1 als komplexe IDL, relative mächtig
- Weitere Encodings:
  - PER (“packed”), XER (XML), DER (“distinguished” – eindeutiges BER)

- Interner Referenz Encoder/Decoder zum Vergleich
- ASN1C als Compiler/Codegenerator Alternative !

ASN.1 Beispiel

Serialization / schemalose API

```
int serialize(size_t count, char* buf, int bufSize) {
  BerEnc codec(buf, bufSize));
  int resultSize = 0;

  cd->BerBegin(BER_CONSTRUCTED | BER_SEQUENCE);

  for (size_t i = 0; i < count; ++i) {
    cd->BerBegin(BER_CONSTRUCTED + BER_CONTEXT_SPECIFIC + 1);
    cd->BerPutInt(BER_CONTEXT_SPECIFIC + 1, static_cast<int>(i));
    cd->BerPutLongLong(BER_CONTEXT_SPECIFIC + 2, (int_val++)+i);
    cd->BerPutStdString(BER_CONTEXT_SPECIFIC + 3, string_short);
    // stringvec
    cd->BerBegin((BER_CONSTRUCTED | BER_SEQUENCE) + 4); //stringvec
    for (size_t j = 0; j < cont_count; ++j) {
      cd->BerPutStdString(BER_CONTEXT_SPECIFIC + j, string_short);
    }
    cd->BerEnd();
    // intvec left out ...
    cd->BerEnd();
  }
  cd->BerEnd();
  codec.BerResultSize(&resultSize);
  return resultSize;
}
```

ASN.1 Beispiel

Deserialize

```
void deserialize(size_t count, char* buf, int bufSize) {
  std::vector<container_vec_t> data(count); int ec = 0;
  BerEnc codec(buf, bufSize);
  BerEnter(cd);
  for (auto& current : data) {
    ec = BerEnter(codec); // don't switch on tags, just assume correct layout
    if (!ec) ec = BerGetInt(codec, &current.id);
    BerNext(codec);
    if (!ec) ec = BerGetLongLong(codec, current.int1);
    BerNext(codec);
    if (!ec) ec = BerGetStdString(codec, current.text1);
    BerNext(codecd);
    BerEnter(codec);
    current.stringvec.resize(cont_count);
    for (size_t i = 0; i < cont_count; ++i) {
      if (!ec) ec = BerGetStdString(codec, current.stringvec[i]);
    }
    BerLeave(codec); // intvec left out
    BerLeave(codec); ec = BerNext(codec);
    if (ec) throw std::runtime_error("Error decoding cont_vec value");
  }
  BerLeave(codec);
  BerFree(codec)
}
```

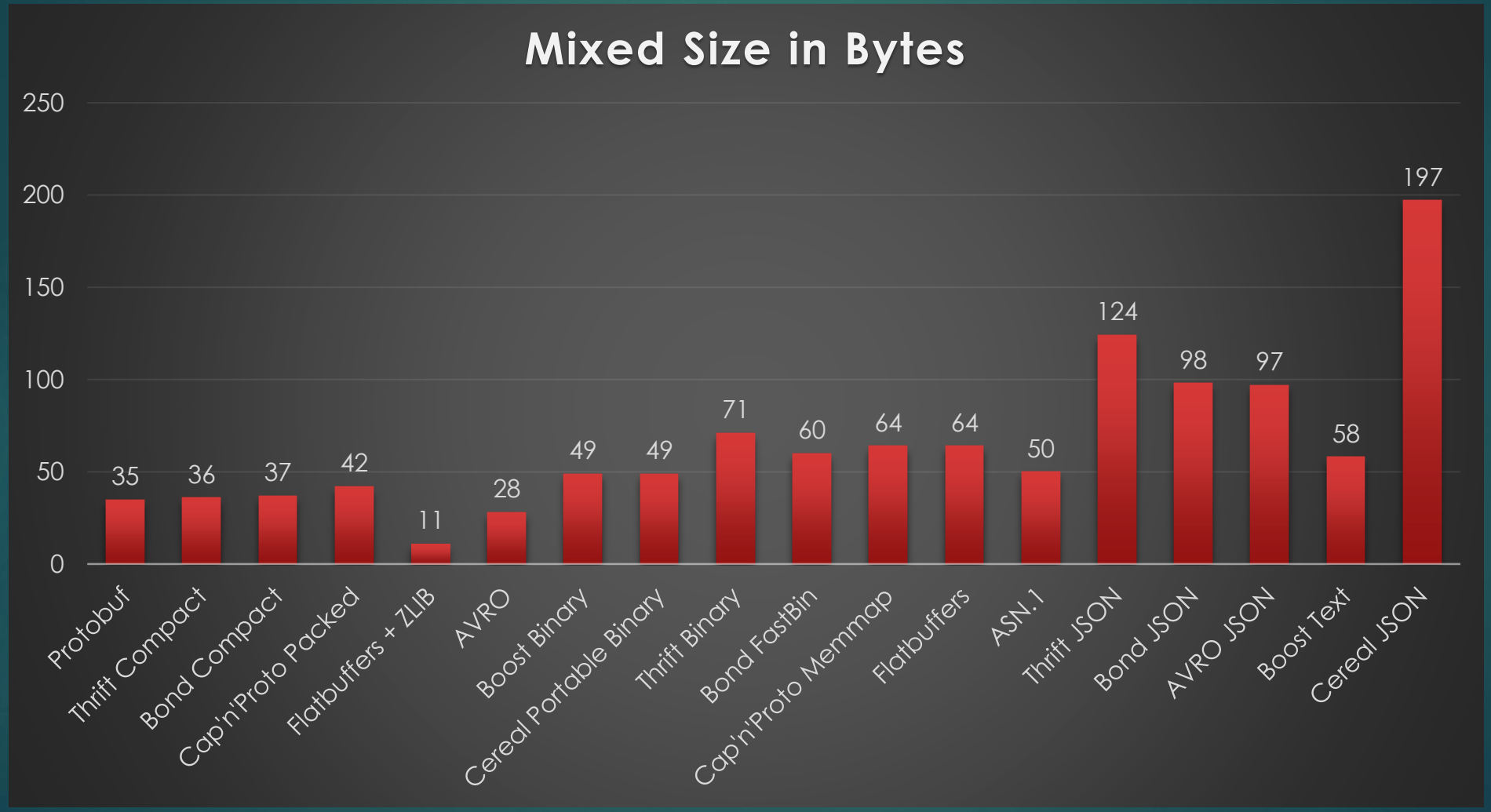
Alternativen

- Interessant, aber hier nicht betrachtet:

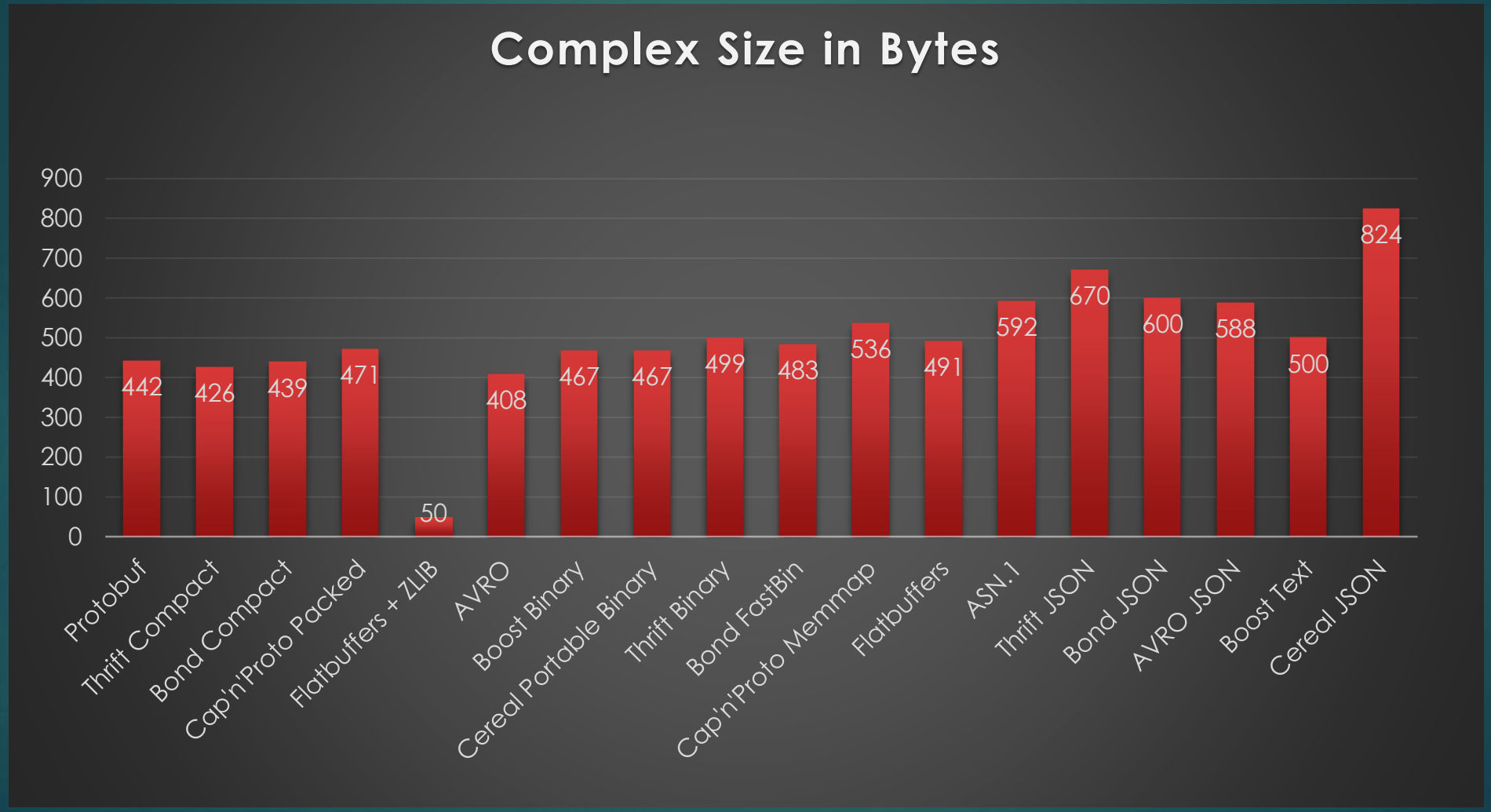
- EXI – Binary XML Format
  - Keine stabile / verbreitete Open Source Bibliothek
  - Aber interessant weil:
    - sehr gute Kompression (teilweise seltsame "Optimierungen", z.B. float)
    - Umwandlung XML Text, XML Schema

- MSGPack
  - "It's like JSON, but fast and small", effizientes Binary Encoding
  - Sehr viele unterstützte Sprachen
  - Kein Schema, komplizierte C++ Interfaces

Benchmark - Paketgrößen



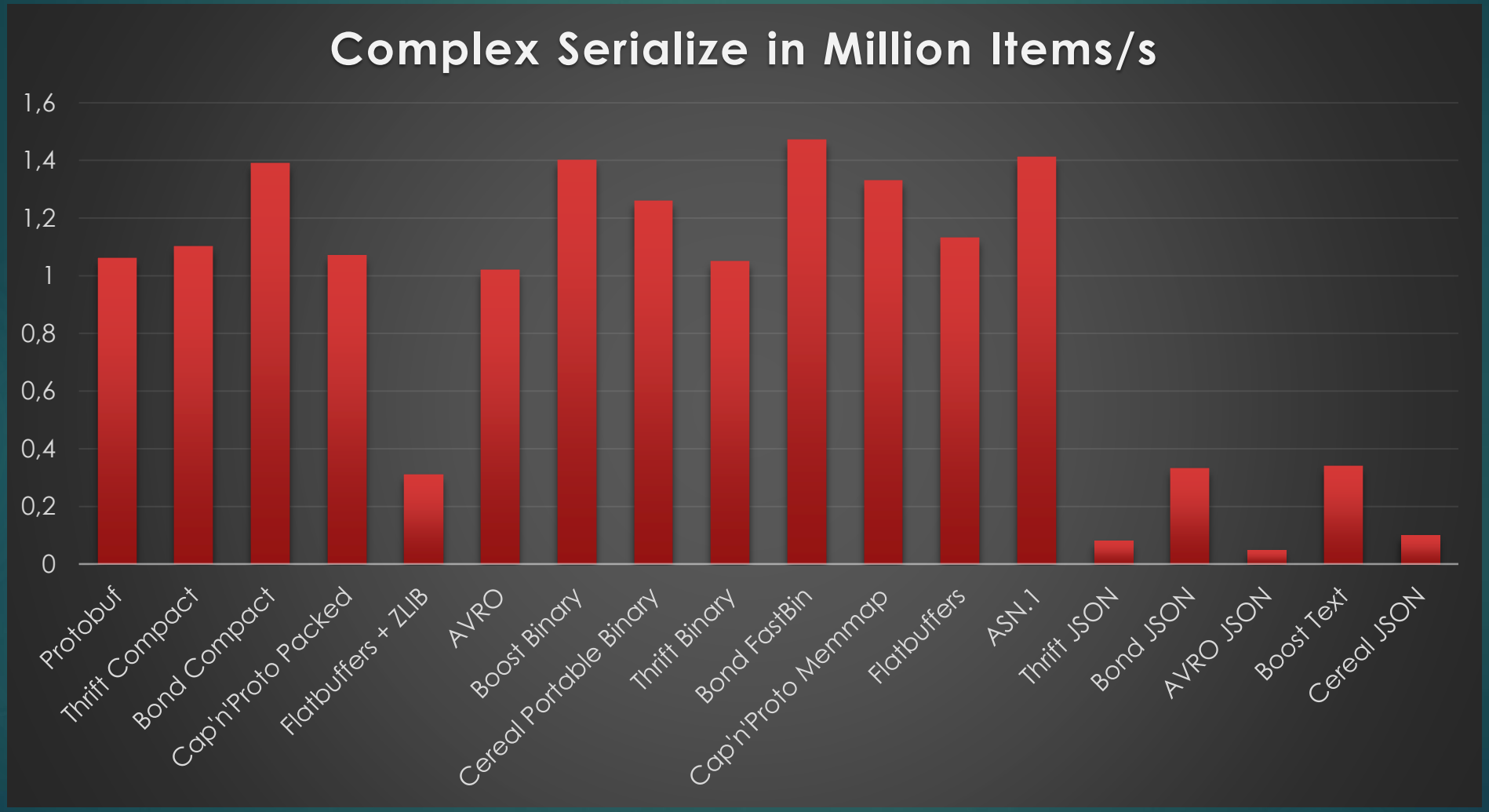
Benchmark - Paketgrößen



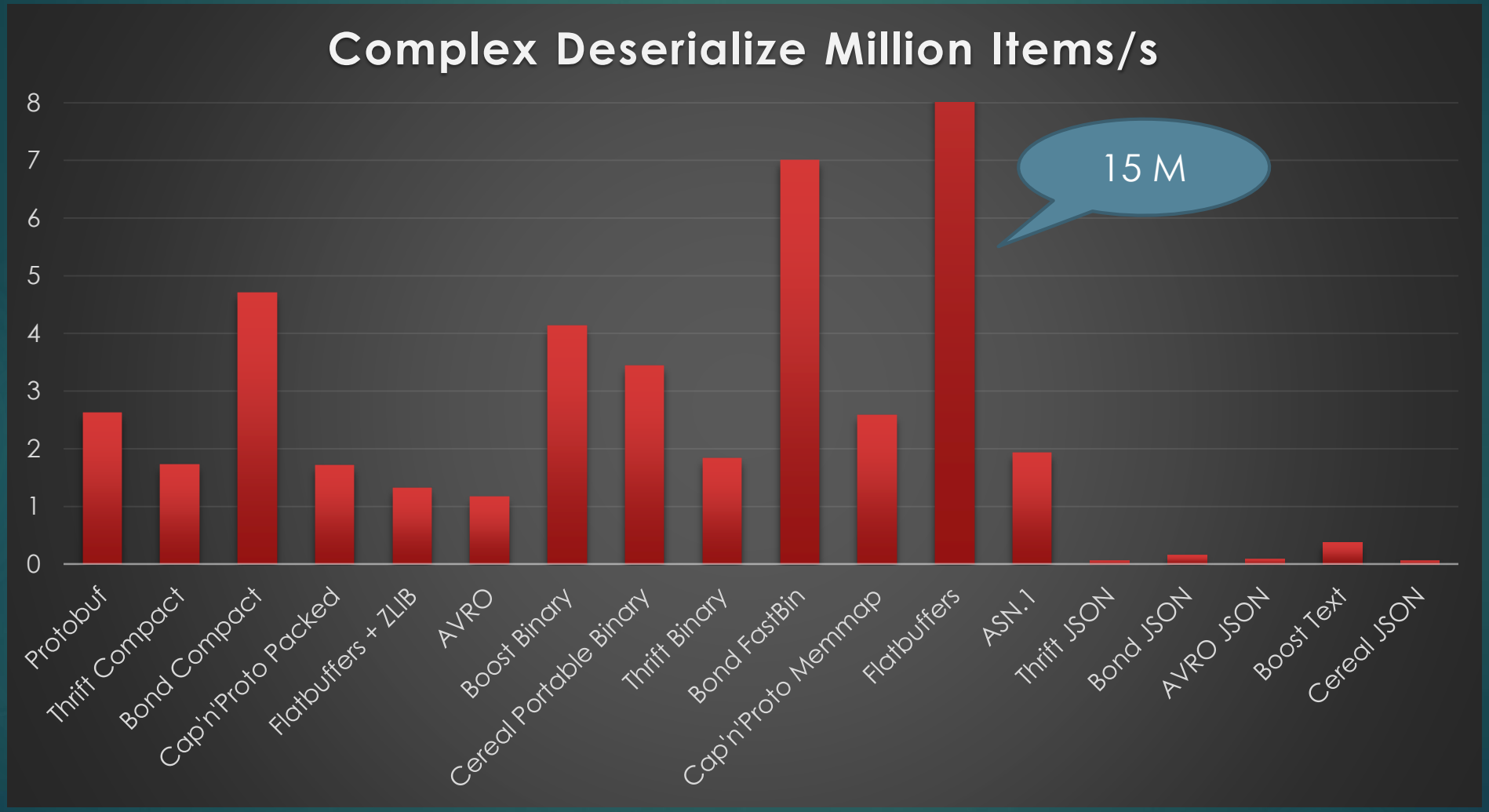
Benchmarks – Packet Größen

	Ints	Mixed	Complex	ContVec	Map
Protobuf	17	35	442	722	319
Thrift Compact	17	36	426	675	242
Bond Compact	17	37	439	674	243
Cap'n'Proto Packed	13	42	471	1043	442
Flatbuffers + ZLIB	5	11	50	27	88
AVRO	11	28	408	669	267
Boost Binary	32	49	467	1299	461
Cereal Portable Binary	32	49	467	1295	457
Thrift Binary	48	71	499	1101	384
Bond FastBin	47	60	483	941	317
Cap'n'Proto Memmap	32	64	536	1680	888
Flatbuffers	32	64	491	1460	604
ASN.1	32	50	592	906	321
Thrift JSON	86	124	670	1056	421
Bond JSON	59	98	600	1019	343
AVRO JSON	60	97	588	1020	384
Boost Text	25	58	500	983	318
Cereal JSON	145	197	824	2415	1985

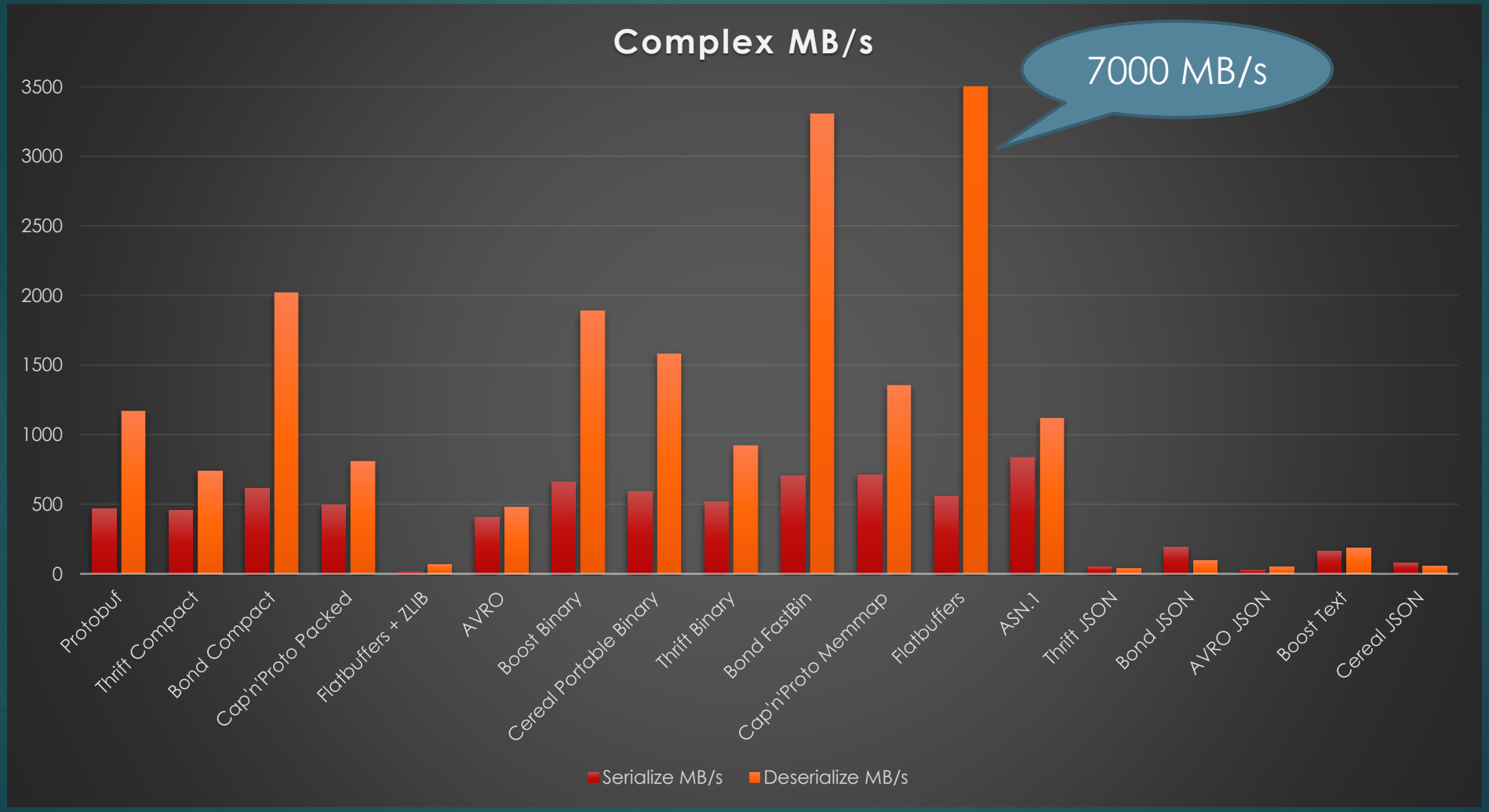
Benchmark - Pakete / s



Benchmark - Pakete / s



Benchmark – MByte/s



Fazit - Bewertungsfaktoren

- Auswahl nach Anwendungsfall!
  - schnelles/langsames LAN, Internet, Archivierung?
  - Sprachen, Datenaustausch, asymmetrische Encoder/Decoder?

Framework	Doku	Stabilität	API	Sicherheit
Protobuf	+++	+++	++	++++
Thrift	+	+++	++	+++
Bond	++	++	+++	+++
Flatbuffers	+	+	+	++
Cap'n'Proto	++	+	+++	++
AVRO	+	+	+	+++
Boost	+++	++	++	++
Cereal	++	+	+++	+
ASN.1	-	++++	+	+

Fazit – Stabil und sicher

- Google Protobuf
  - sinnvolle Geschwindigkeit
  - sehr stabil, verbreitet und ausführlich ge-reviewed
- Apache Thrift als Alternative
  - viele große Unternehmen als Nutzer bekannt
- Microsoft Bond jünger, wenig verbreitet
  - aber durch Microsoft “guten Support”
  - gute API und schnell!

Fazit – “Maximale Geschwindigkeit”

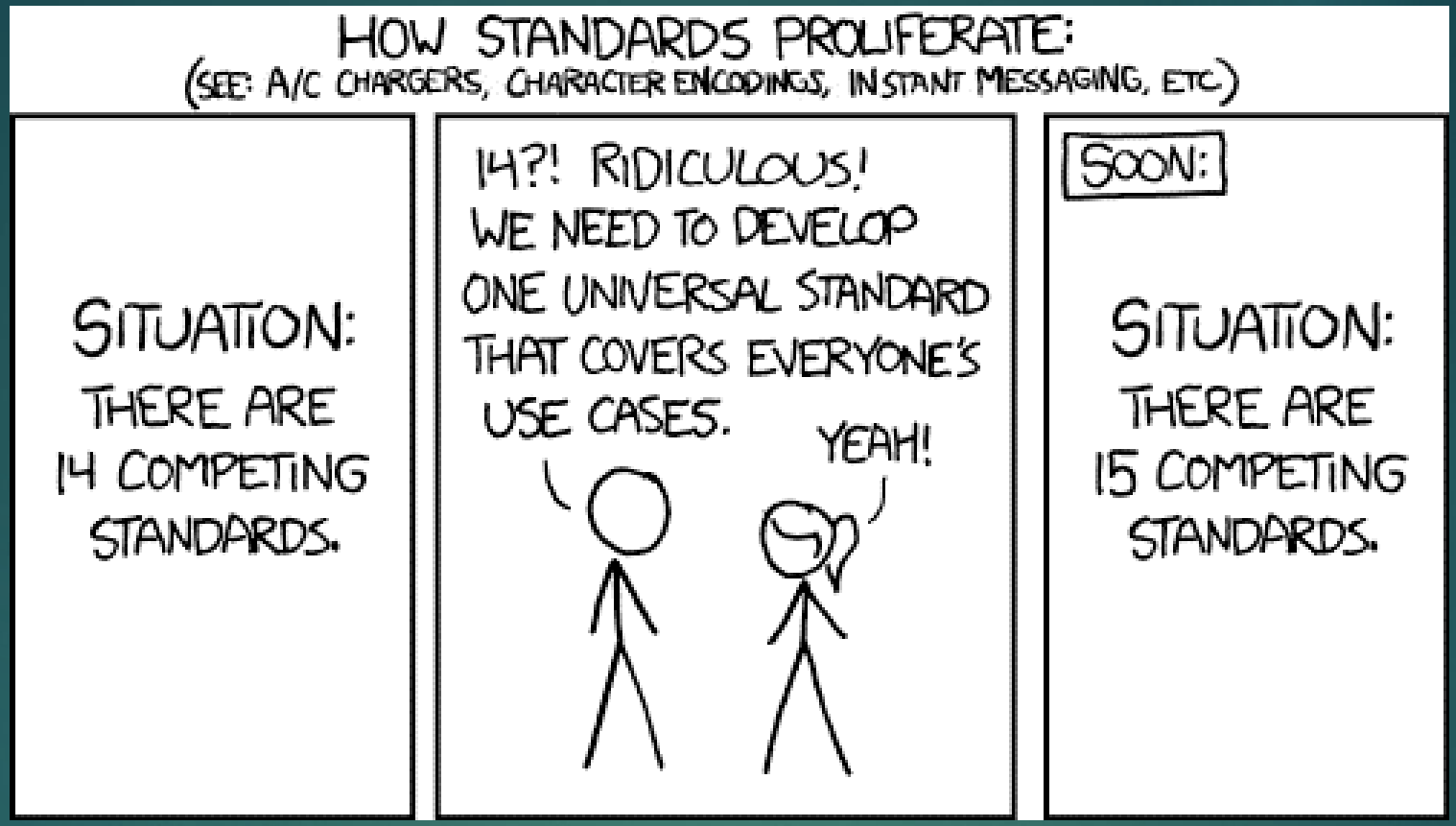
- Cap’n’Proto oder Microsoft Bond sehr schnell
- Bond (und Protobuf) schnell mit “compact” Format
  - “LAN” / “Internet” Kommunikation
- Thrift oft nicht viel langsamer + sehr flexibel
- Flatbuffers für spezielle Anwendungen!
  - sehr schnell auf Decoder Seite
  - Kompression mit ZLIB / Brotli
- ACHTUNG: high performance >= Gbit/s
  - meistens sind alle Binary Formate “schnell genug”

Fazit – “C++ only / keine IDL”

- Ziel: Serialisierung ohne IDL / Codegenerierung
- Typdefinition direkt in C++
  - vollständige Kontrolle über Datenstrukturen!
  - Pointer Tracking! (Boost: alle, Cereal: smart Pointer)
- Cereal als reste Wahl
  - schneller und einfacher zu nutzen als Boost
  - Header-only, mehr Formate,Portabel
- Boost Serialization als Standard

Fazit – “Kompatibilität”

- XML
  - durch Schema sehr gut validierbar
- JSON, CSV
  - So gut wie überall lesbar
- Apache Thrift
  - Sehr breite Sprachunterstützung
- Microsoft Bond
  - Gutes Schema Handling
  - Inline / dynamische Schemata



http://xkcd.com/927/

Fragen oder Anmerkungen?

Oder einfach per Mail: Volker Aßmann (volker.assmann@gmail.com)

Code: https://github.com/volka/talks