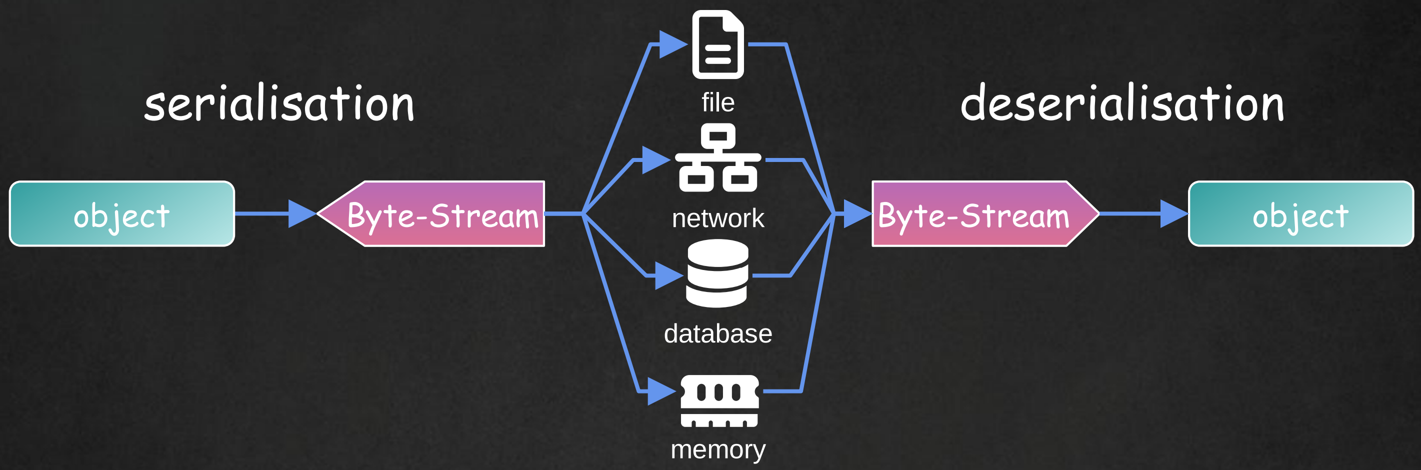
Yet-Another-Serialiser (YAS)

or: <u>why we are not reusing</u> and opted to write yet another custom data serialiser

Ralph J. Steinhagen



- Serialisation is key when information has to be transmitted, stored and later retrieved by (often quite) different subsystems, architectures and/or programming languages.
  - with a multitude of different serialiser libraries, a non-negligible subset of these claim to be the fastest, most efficient, easiest-to-use or *<add your favourite superlative here>*
  - this is true for most libraries' original use-case but often breaks down for other applications.
- This talk aims at motivating our compile-time reflection based approach used in OpenCMW

Ralph J. Steinhagen, R.Steinhagen@GSI.de, 2022-10-13

Our FAIR/OpenCMW focus

1. **performance: minimise end-to-end latency** between server-/client- processing worker function
2. facilitate **multi-protocol** implementations & **protocol evolution**
   (i.e. loose coupling between server-/client-side data object definitions)
3. decoupling client-/server-side logic/worker from wire-formats
   (i.e. independent on specific format: binary@1, binary#2, JSON, XML, YML, …)
4. **derive schemas directly class structures & basic types** (PoCos/PoJos)
   rather than a 3rd-party Interface-Description-Languages (IDL) definitions
5. allow **user-level schema extensions** through optional custom (de-)serialiser routines
6. **self-documented data-structures** to communicate the data-exchange-API-intend to the client
7. **minimise code-bloat** ↔ performance (L1/L2/L3 cache sizes) & minimises maintenance overhead
8. **test driven development**
9. **free- and open-source code w/o strings-attached**

Performance

minimise end-to-end latency

Bandwidth:
maximum rate
of data transfer

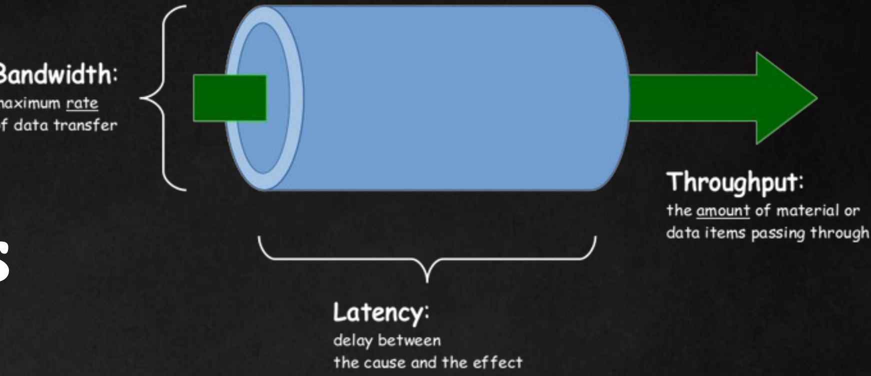
Throughput:
the amount of material or
data items passing through

Latency:
delay between
the cause and the effect

- Some trade-offs that affect bandwidth and latency
  - bandwidth: meta-data, compression,
  - latency: numerical complexity (e.g. due to compression), necessity to read-ahead

1. Performance

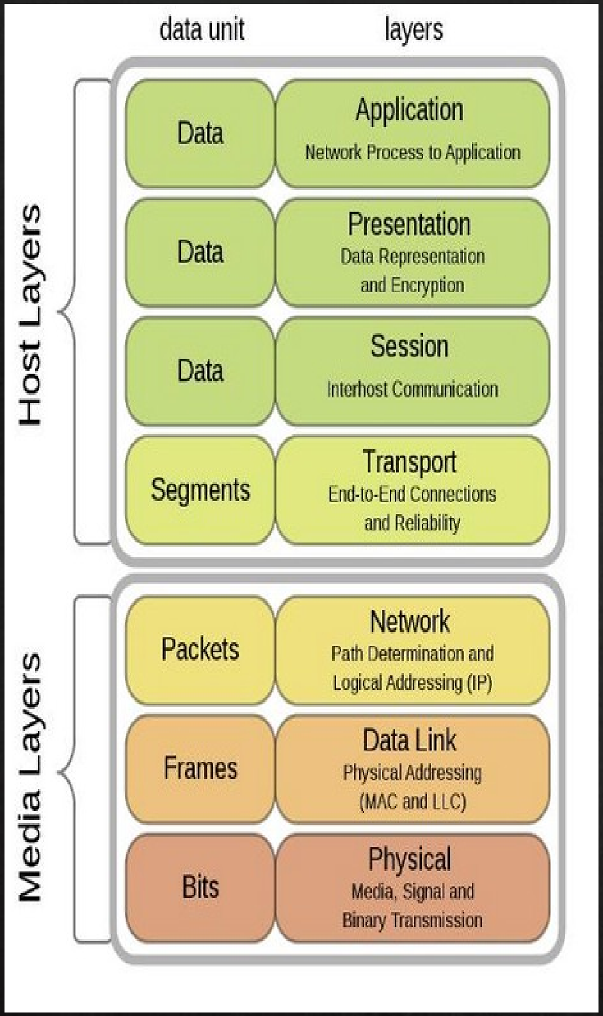
Some Trade-offs/Choices



- meta-data (e.g. field name, hashes, type, size, ...)
  - ✗ overhead, esp. if server & client share same object definition and CPU architecture
    (N.B. some "fast" serialiser transmit plain POCOs)
  - ✓ supports different architecture (e.g. ARM, x86, PowerPC) and programming languages/tools (e.g. C++, Java, Python)
  - ✓ protocol evolution i.e. decouples server/client object definition
    e.g. server can add/remove/change new field while client reads only fields that are relevant to its application
- compression/size i.e. reduces amount of data
  notably: only transmit what is actually needed by the client
  - ✓ less data → less to encode → less to transmit → lower latency
    esp. for network- or bus-limited applications ↔ internet, CAN, low-power wifi/IoT, ...
  - ✗ numerical compression complexity increases CPU load/latency
    esp. dominant if the main/cache memory bandwidth limit (i.e. > few GB/s)
- read-ahead meta-information i.e. storing absolute markers for next field
  - ✗ wastes additional bytes for type and/or size information
  - ✓ some data structure where the (skipping) length of the wire-format is only known after de-/encoding
    - e.g. 'float' → String conversion: '0.0' ... '0.12345678' ... '3.402823466E+38'
    - e.g. 'std::map<std::string, my_data>' → binary/string wire-format
- intermediate representation .e.g. wire-format → std::map<std::string, meta_data> → object
  - ✓ run-time parsing and/or unknown object formats
  - ✗ compile-time compiler/performance optimisations & may need to read the wire-format multiple times

2. Multi-Protocol Support

& protocol evolution

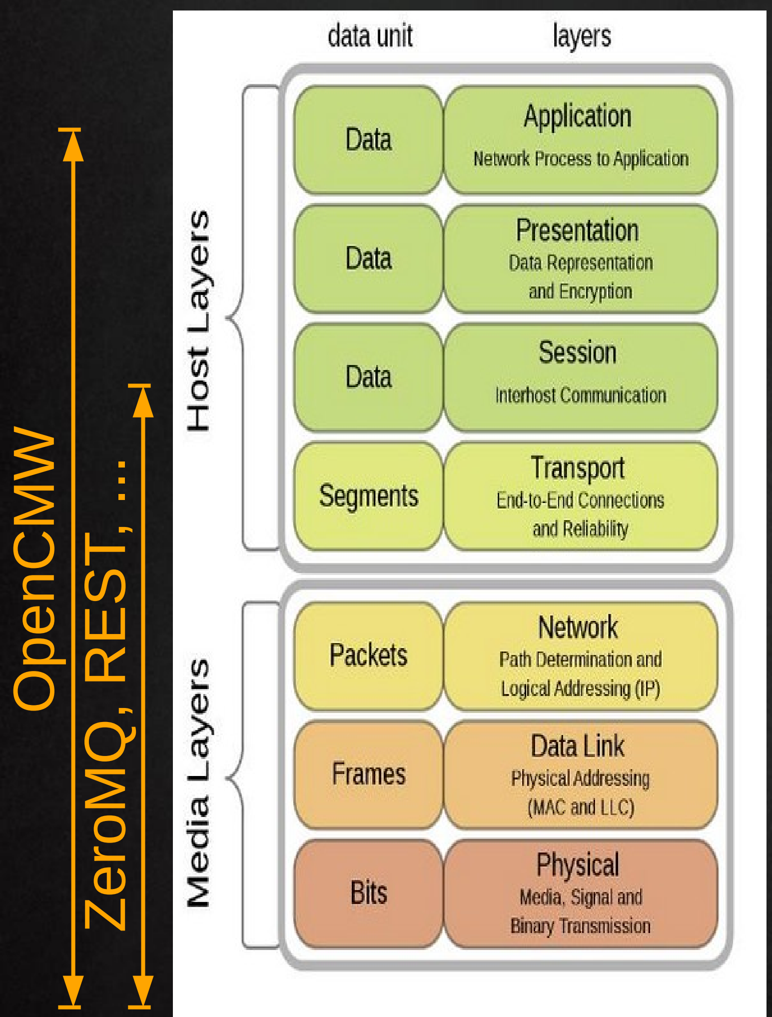


protocol stack of the OSI model

- There are a multitude of middleware protocols
  - e.g. Kafka, ActiveMQ, Flow, myriad of <ZeroMQ-based>, MQTT, RabbitMQ, Mosquitto, HTTP, OpenCMW, ...
- and serialiser implementation out there
  - e.g. Cap'n Proto, FlatBuffer, Protobuff, XML, YML, JSON, 'CSV', HTML, SBE, YaS, ...
- highly biased topic:
  some aim at world-domination but reality in larger facilities consists often of a diverse middleware landscape
  → nearly always need some adapter, proxies, …
  - high-risk of "onion-layered" integrations
    N.B. OpenCMW aims at solving this issue through 'composition'

2. OpenCMW 's Majordomo Transport Protocol

aims at an extendable full protocol stack implementation



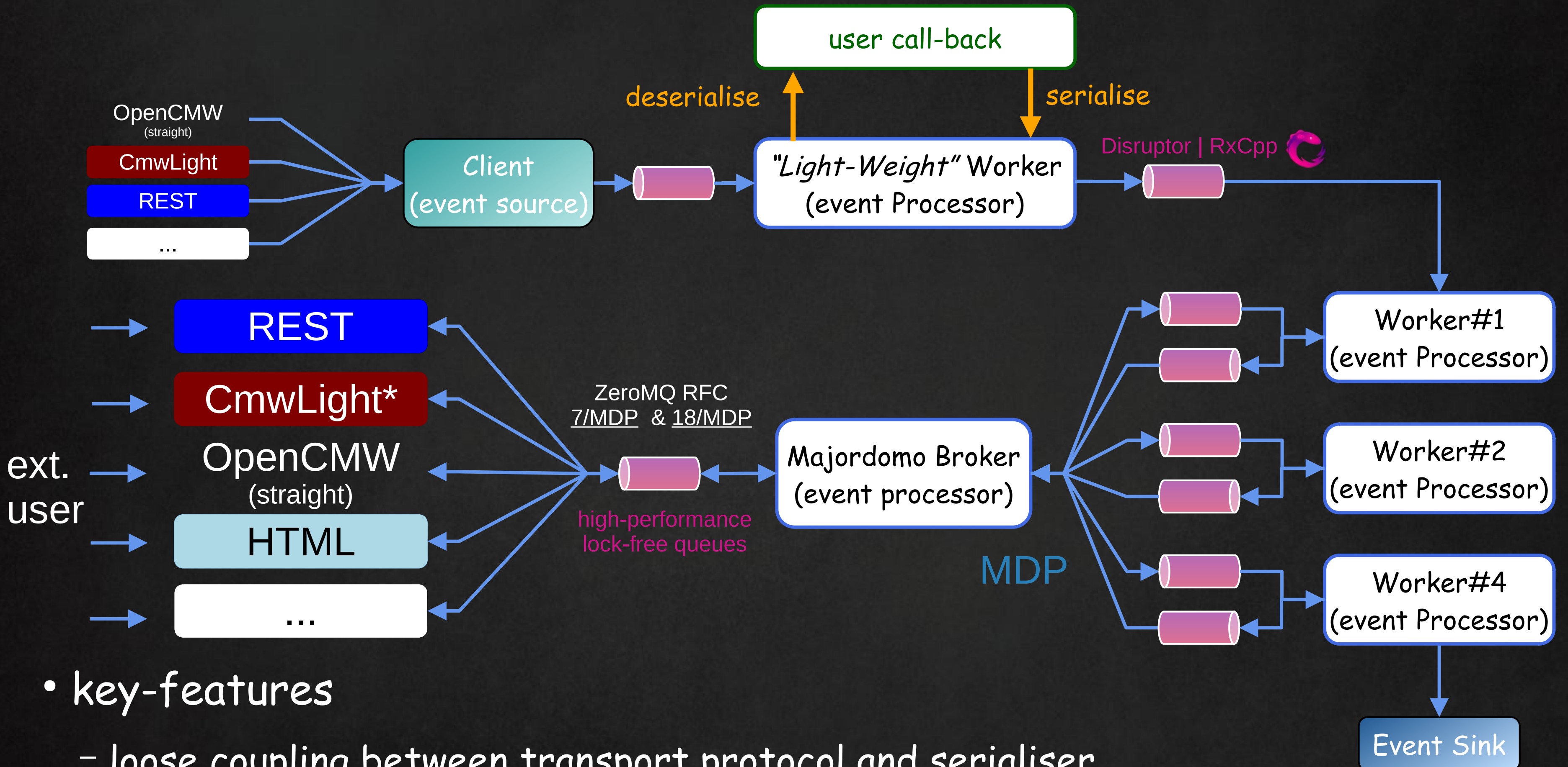
protocol stack of the OSI model

… with (optional) 'batteries' included:

- Transport protocols:
  - Majordomo (ZeroMQ: RFC 7/MDP & 18/MDP),
  - RDA3 (proprietary GSI/CERN transport
  - HTTP/REST (long-polling, SSE): web-services, routable to non-GSI/FAIR networks
  - RADIO/DISH (low-latency UDP) – WIP
  - *… <add your protocol here>*
- Serialisers
  - YaS (binary): annotated type- and physical unit-safe
  - CmwLight (binary): ACC-specific binary protocol
  - JSON (text): data exchange with web-based REST clients
  - YML (text): service config management (human readable)
  - HTML: server-side rendered fixed-displays (+WASM), expert diagnostics tools, ...
  - *… <add your serialiser here>*
- RPC/Streaming call-backs: lambda → convenience classes
- lock-free circular buffers → event sourcing pattern
- thread-affinity, -tools & -pools
- settings management, ...

3. decoupling logic ↔ wire-format

OpenCMW's Microservice Architecture



- key-features
  - loose coupling between transport protocol and serialiser
  - worker logic independent from specific wire-format

3. decoupling logic ↔ wire-format

basic REQ/REP & PUB/SUB example

```
struct AddressRequest {                                    // request domain-object
    int id;
};
ENABLE_REFLECTION_FOR(AddressRequest, id)
```

```
struct AddressEntry {                                      // reply domain-object
    int                                                    id;
    Annotated<std::string, NoUnit, "Name of the person">   name;
    std::string                                            street;
    Annotated<int, NoUnit, "Number">                       streetNumber;
    std::string                                            postalCode;
    std::string                                            city;
    bool                                                   isCurrent;
};
ENABLE_REFLECTION_FOR(AddressEntry, name, street, streetNumber, postalCode, city, isCurrent)
```

```
struct RequestContext {
    const MdpMessage request;
    MdpMessage       reply;
    MIME::MimeType   mimeType = MIME::BINARY;
```

key: transport and wire-format are independent

low-level expert-dev fall-back

```
struct TestContext {
    TimingCtx      ctx;
    MIME::MimeType contentType = MIME::HTML;
};
ENABLE_REFLECTION_FOR(TestContext,ctx,contentType)
```

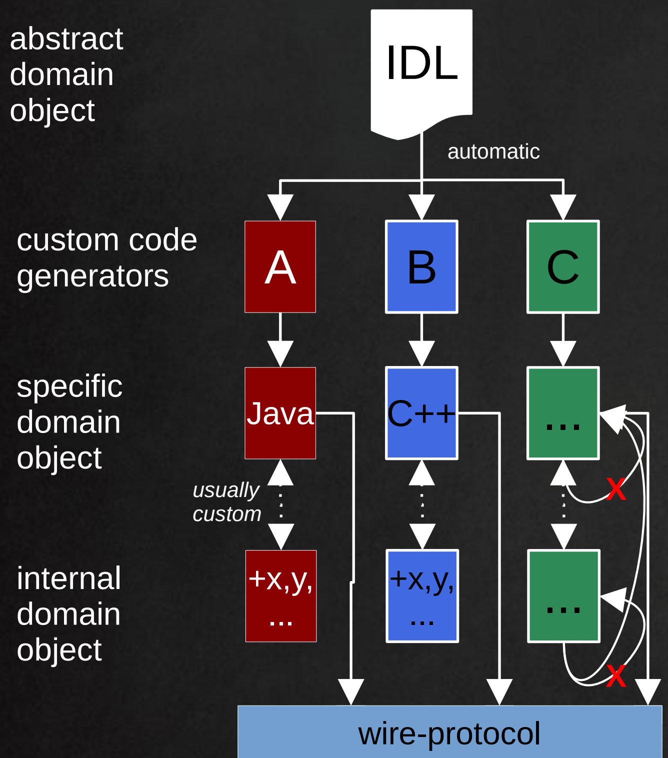
filter/query domain-object

```
struct TestAddressHandler {
    std::unordered_map<int, AddressEntry> _entries;

    TestAddressHandler() { _entries.emplace(42, AddressEntry{ 42, "Santa Claus", "Elf Road", 123, "88888", "North Pole", true }); }

    void operator()(opencmw::majordomo::RequestContext &rawCtx, const TestContext & /*requestContext*/, const AddressRequest &request,
                                                                TestContext & /*replyContext*/,         AddressEntry   &reply) {
        if (rawCtx.request.command() == Command::Get) {
            const auto it = _entries.find(request.id);
            if (it == _entries.end()) {
                reply = _entries.cbegin()->second;
            } else {
                reply = it->second;
            }
        } else if (rawCtx.request.command() == Command::Set) {
            /* do nothing */
        }
    }
};
```

4. Core Serialiser Choice

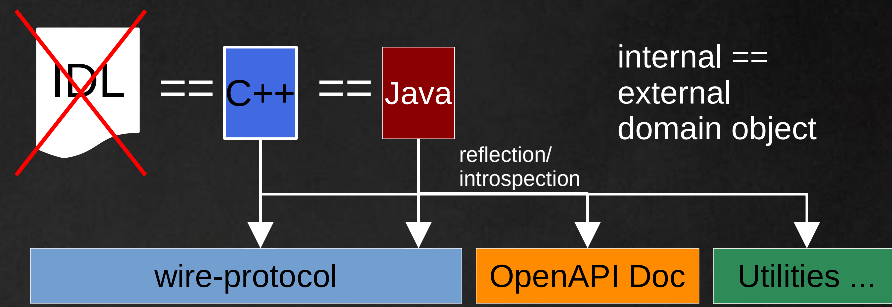
derive schemas directly class structures & basic types

- classic IDL-based design:
  (i.e. protobuf, FastBuffer, CapNProto, …)



- either: generic serialiser not necessarily optimised for our data use-case
- or: need to maintain multiple custom generators
  → custom (eventually buggy/out-dated) generator
- users need to learn IDL syntax & build-system magic
- requires very disciplined developers (three interfaces to maintain)

- static compile-time based reflection
  (IDL is identical to PoCo/PoJo)



- **cognitive complexity reduction**
  regardless of primary programming language preference:
  everyone can read & comprehend basic C++/Java/... class definitions
- **relies on <u>existing</u> Java & C++23 standard** (refl-cpp: visitor-pattern)
  → less external dependencies
  → very little custom code to maintain since most is std::c++
- allows less disciplined/new developers (one interface to maintain)
- self-describing format, OpenAPI definitions, utilities, ….
- turned out to be more performant that most generic serialisers ...

5. user-level schema extensions

through 'Duck'-Typing Polymorphism

"If it walks like a duck and it quacks like a duck, then it must be a duck"

```
template<SerialiserProtocol protocol, typename T>
struct IoSerialiser {
    constexpr static uint8_t getDataTypeId() { return 0xFF; } // default value
    constexpr static void    serialise(IoBuffer    &/*buffer*/, FieldDescription auto const &field, const T &value) {
        throw ProtocolException("not implemented IoSerialiser<{}>::serialise(IoBuffer&, field: '{}', type '{}' value: '{}')", /* … */ );
    }
    constexpr static void deserialise(IoBuffer & /*buffer*/, FieldDescription auto const &field, T &value) {
        throw ProtocolException("not implemented IoSerialiser<{}>::deserialise(IoBuffer&, field: '{}', type '{}' value: '{}')",  /* … */ );
    }
};

template<SerialiserProtocol protocol, bool writeMetaInfo = true>
constexpr void serialise(IoBuffer &buffer, ReflectableClass auto const &value, FieldDescription auto const parent) { // […] N.B. simplified
    for_each(refl::reflect(value).members, [&](const auto member, const auto fieldIndex) {
        auto &&fieldValue = member(value);
        using UnwrappedMemberType = std::remove_reference_t<decltype(getAnnotatedMember(unwrapPointer(fieldValue)))>;
        FieldDescription auto field = …;

        if constexpr (isReflectableClass<UnwrappedMemberType>()) { // […] field is a nested data-structure
            // […] write leading start-marker
            serialise<protocol, writeMetaInfo>(buffer, getAnnotatedMember(unwrapPointer(fieldValue)), field); // recursive call
            // […] write trailing stop-marker
        } else { // […] field is a (possibly annotated) primitive type → write (optional) field meta-data and primitive value
            IoSerialiser<protocol, writeMetaInfo>(buffer, field, getAnnotatedMember(unwrapPointer(fieldValue)));
        }
    });
}
```

… similar for deserialise function (albeit ~150 lines of code)

- primary optimisation goals:
  - read and process data only once, avoid if-else or jump branches → favours direct wire-format to object conversion
  - do field-name to index lookups already during compile-time (constexpr reflection/visitor pattern)

5. user-level schema extensions

through 'Duck'-Typing Polymorphism

"If it walks like a duck and it quacks like a duck, then it must be a duck"

constexpr map<K,V> trick courtesy Jason Turner (C++ Weekly)

N.B std::map<K,V> and std::unordered_map<K,V> aren't constexpr

```
template<typename Key, typename Value, std::size_t size>
struct ConstExprMap {
    const std::array<std::pair<Key, Value>, size> data;

    [[nodiscard]] constexpr Value at(const Key &key) const {
        const auto itr = std::ranges::find_if(begin(data), end(data), [&key](const auto &v) {
            return v.first == key;
        });
        return (itr != end(data)) ? itr->second : throw std::out_of_range(fmt::format("key '{}' not found", key));
    }

    [[nodiscard]] constexpr Value at(const Key &key, const Value &defaultValue) const noexcept {
        auto itr = std::ranges::find_if(begin(data), end(data), [&key](const auto &v) { return v.first == key; });
        return (itr != end(data)) ? itr->second : defaultValue;
    }
};
```

… works well for smallish maps.

N.B. further optimisation potential: constexpr/compile-time sorting of keys (hard for strings)
→ opens option of binary search $O(N) \rightarrow O(\log(N))$

5. user-level schema extensions

through 'Duck'-Typing Polymorphism

"If it walks like a duck and it quacks like a duck, then it must be a duck"

```
template<SerialiserProtocol protocol, typename T>
struct IoSerialiser {
    constexpr static uint8_t getDataTypeId() { return 0xFF; } // default value
    constexpr static void    serialise(IoBuffer    &/*buffer*/, FieldDescription auto const &field, const T &value) {
        throw ProtocolException("not implemented IoSerialiser<{}>::serialise(IoBuffer&, field: '{}', type '{}' value: '{}')", /* … */ );
    }
    constexpr static void deserialise(IoBuffer & /*buffer*/, FieldDescription auto const &field, T &value) {
        throw ProtocolException("not implemented IoSerialiser<{}>::deserialise(IoBuffer&, field: '{}', type '{}' value: '{}')",  /* … */ );
    }
};

template<Number T> // catches all numbers
struct IoSerialiser<YaS, T> {
    static constexpr uint8_t getDataTypeId() { return yas::getDataTypeId<T>(); }
    constexpr static void    serialise(IoBuffer &buffer, FieldDescription auto const    &/*field*/, const T &value) noexcept {
        buffer.put(value);
    }
    constexpr static void deserialise(IoBuffer &buffer, FieldDescription auto const & /*field*/, T &value) noexcept {
        value = buffer.get<T>();
    }
};
```

5. user-level schema extensions

through 'Duck'-Typing Polymorphism

"If it walks like a duck and it quacks like a duck, then it must be a duck"

```
template<SerialiserProtocol protocol, typename T>
struct IoSerialiser {
    constexpr static uint8_t getDataTypeId() { return 0xFF; } // default value
    constexpr static void    serialise(IoBuffer    &/*buffer*/, FieldDescription auto const &field, const T &value) {
        throw ProtocolException("not implemented IoSerialiser<{}>::serialise(IoBuffer&, field: '{}', type '{}' value: '{}')", /* … */ );
    }
    constexpr static void deserialise(IoBuffer & /*buffer*/, FieldDescription auto const &field, T &value) {
        throw ProtocolException("not implemented IoSerialiser<{}>::deserialise(IoBuffer&, field: '{}', type '{}' value: '{}')",  /* … */ );
    }
};

template<Number T> // catches all numbers
struct IoSerialiser<Json, T> {
    inline static constexpr uint8_t getDataTypeId() { return IoSerialiser<Json, OTHER>::getDataTypeId(); }
    constexpr static void           serialise(IoBuffer &buffer, FieldDescription auto const           &/*field*/, const T &value) {
        const auto  start  = buffer.size();
        auto            size  = buffer.capacity();
        auto            data  = reinterpret_cast<char *>(buffer.data());

        std::to_chars_result result;
        if constexpr (std::is_floating_point_v<T>) {
            result = std::to_chars(data + start, data + size, value, std::chars_format::scientific);
        } else {
            result = std::to_chars(data + start, data + size, value); // fall-back
        }

        if (result.ec != std::errc()) { throw ProtocolException("error({}) serialising number at buffer position: {}", result.ec, start); }
        buffer.resize(static_cast<size_t>(result.ptr - data)); // new position
    }
    constexpr static void deserialise(IoBuffer &buffer, FieldDescription auto const & /*field*/, T &value) {  /* […] */   }
};
```

5. user-level schema extensions

Example: IoSerialiser<YaS> wire-format

- Frame Header – written once (pretty standard):
  magic number (compatibility), protocol name 'YAS\0', major|minor|micro version
- Field Header – for each struct/class member field:
  - uint8_t: data type ID – fixed for default types, 0xFD→OTHER/custom
  - int32_t: unique field hashCode identifier ↔ N.B. performance bottleneck (will probably drop this)
  - int32_t: dataStart offset
  - int32_t: dataSize – *N.B. 'headerStart' + 'dataStart + dataSize' == start of next field header*
  - std::string: full field name
  - optional – *N.B. send for each REQ/REP, suppressed/send only once for PUB/SUB pair*
    - std::string physical field unit ↔ use of mp-units
    - std::string user-/application-specific field description
    - uint8_t external field modifier → used communicates protocol evolution
      - i.e. enum: RO, RW_DEPRECATED, RO_DEPRECATED, RW_PRIVATE, RO_PRIVATE, UNKNOWN
      - N.B. '_PRIVATE' == private/non-production API
  - *<actual data block starts here>*
  - <… data body …>

5. user-level schema extensions

Example: IoSerialiser<YaS> wire-format

- primitive types – int[8, 64]_t || float || double || bool || uint8_t
  via: std::is_signed_v<RawType> || is_same_v<RawType, bool> || is_same_v<RawType, uint8_t>
  - <straight x86_64 type> – preferred architecture/others need converter
    - N.B. dropped host/network since little-endian architectures dominate
- UTF-8 Strings, array, vector or other std container of primitives/strings
  - int32_t: size of vector N
  - N x [<type>]: vector elements/char
  - for strings: enforces trailing '\0'
- N-Dim arrays ↔ idea: follow-up C++23 std::mdspan<> concept
  - N_d x [<type>]: N_d-dim array storing the cardinality of each dimension
  - N x [<type>]: strided array storing the flat (dense) matrix
- maps ↔ composite of map size and arrays of key and value
- nested structs
  - uint8_t: START_MARKER
    - <… nested data body …>
  - uint8_t: END_MARKER
- optional custom types – essentially: server-/client need to agree/use the same definition, e.g.
  - std::string type name
  - <custom data format> user defined, …

this where C++, meta-programming, and constexpr shines

10-fold SLOC reduction

example: IoSerialiser<YaS>

```cpp
#include <IoSerialiserYaS.hpp>

struct className {
    int         field1;
    float       field2;
    std::string field3;
    bool operator==(const className &) const = default;
};
ENABLE_REFLECTION_FOR(className, field1, field2, field3)  // needed for compile-time reflection until C++23/26

int main() {
    className a{ 1, 0.5F, "Hello World!" };
    className b{ 1, 0.501F, "Γειά σου Κόσμε!" };


    diffView(std::cout, a, b); // convenience diff function

    opencmw::IoBuffer buffer;
    assert(a != b && "a & b should be unequal here");

    opencmw::serialise<opencmw::YaS>(buffer, a); // serialise 'a' into the byte buffer


    try { // de-serialise the byte buffer into 'b'
        opencmw::deserialise<opencmw::YaS, opencmw::ProtocolCheck::LENIENT>(buffer, b);
    } catch (const ProtocolException &e) {
        std::cout << "caught deserialisation exception: " << e.what() << std::endl;
    }
    assert(a == b && "a & b should be equal here"); // just checking
}
```

```
Output:
diffView: className(
  0: int32_t               className::field1             = 1
  1: float_t               className::field2             = 0.5 vs. 0.501
  2: string                className::field3             = Hello World! vs. Γειά σου Κόσμε!
 )
```

6. self-documented structures

more elaborate unit-test example

```
struct NestedDataY {
    Annotated<int8_t, length<metre>, "nested int8_t">                          annByteValue   = 11;
    Annotated<int16_t, si::time<second>, "custom description for int16_t">     annShortValue  = 12;
    Annotated<int32_t, NoUnit, "custom description for int32_t">               annIntValue    = 13;
    Annotated<int64_t, NoUnit, "custom description for int64_t">               annLongValue   = 14;
    Annotated<float, energy<gigaelectronvolt>, "custom description for float"> annFloatValue  = 15.0F;
    Annotated<double, mass<kilogram>, "custom description for double", RW>     annDoubleValue = 16.0;
    Annotated<std::string, NoUnit, "custom description for string">            annStringValue = std::string("nested string");
    Annotated<std::array<double, 10>, NoUnit>                                   annDoubleArray = std::array<double, 10>{ /*…*/ };
    Annotated<std::vector<float>, NoUnit>                                       annFloatVector = std::vector{ /*…*/ };
    Annotated<std::array<bool, 4>, NoUnit, "description for bool array!">      annBoolArray   = std::array<bool, 4>{ /*…*/ };
    auto operator<=>(const NestedDataY &) const = default;
};
ENABLE_REFLECTION_FOR(NestedDataY, annByteValue, annShortValue, annIntValue, annLongValue, annFloatValue, annDoubleValue,
annStringValue, annDoubleArray, annFloatVector, annBoolArray)

struct DataY {
    bool                                          boolValue        = false;
    int8_t                                        byteValue        = 1;
    int16_t                                       shortValue       = 2;
    int32_t                                       intValue         = 3;
    int64_t                                       longValue        = 4;
    float                                         floatValue       = 5.0F;
    double                                        doubleValue      = 6.0;
    std::string                                   stringValue      = "bare string";
    std::string const                             constStringValue = "unmodifiable string";
    std::array<double, 10>                        doubleArray      = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    std::array<double, 10> const                  constDoubleArray = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    std::vector<float>                            floatVector      = { 0.1F, 1.1F, 2.1F, 3.1F, 4.1F, 5.1F, /*…*/ };
    opencmw::MultiArray<double, 2>                doubleMatrix{ { 1, 3, 7, 4, 2, 3 }, { 2, 3 } };
    NestedDataY                                   nestedData;
    Annotated<double, resistance<ohm>>            annotatedValue   = 0.1;
    std::map<std::string, std::string, std::less<>> map            = { { "key1", "value1" }, /*…*/ };
    std::map<std::string, std::string, std::less<>> smallMap       = { { "key1", "value1" } };

    bool                                          operator==(const DataY &) const = default;
};
ENABLE_REFLECTION_FOR(DataY, boolValue, byteValue, shortValue, intValue, longValue, floatValue, doubleValue, stringValue,
constStringValue, doubleArray, floatVector, /*doubleMatrix,*/ nestedData, annotatedValue, map, smallMap)
```

6. self-documented structures

example: IoSerialiser<YAML>

```
---
io_serialiser_yaml_test::DataY:
  boolValue: false
  byteValue: 1
  /* … */
  stringValue: "bare string"
  constStringValue: "unmodifiable string"
  doubleArray: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
  floatVector: [0.1, 1.1, 2.1, 3.1, 4.1, 5.1, 6.1, 8.1, 9.1, 9.1]
  nestedData:
    annByteValue: 11
    annShortValue: 12
    annIntValue: 13
    annLongValue: 14
    annFloatValue: 15
    annDoubleValue: 16
    annStringValue: "nested string"
    annDoubleArray: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    annFloatVector: [0.1, 1.1, 2.1, 3.1, 4.1, 5.1, 6.1, 8.1, 9.1, 9.1]
    annBoolArray: [true, false, true, true]
  # end - nestedData
  annotatedValue: 0.1
  map:
    key1: "value1"
    key2: "value2"
    key3: "value3"
    key4: "value4"
    key5: "value5"
    key6: "value6"
  smallMap: {key1: "value1"}
# end - io_serialiser_yaml_test::DataY
…
```

optional meta info

```
# b
# int8_t

# string
# string
# array<double_t,10>
# vector<float_t>
```

type unit user/server-side description

```
# int8_t    - [m] - nested int8_t
# int16_t   - [s] - custom description for int16_t
# int32_t   - custom description for int32_t
# int64_t   - custom description for int64_t
# float_t   - [GeV] - custom description for float
# double_t  - [kg] - custom description for double
# string    - custom description for string
# array<double_t,10>
# vector<float_t>
# array<b,4>      - description for bool array!

# double_t  - [ohm]
# map<string,string>
```

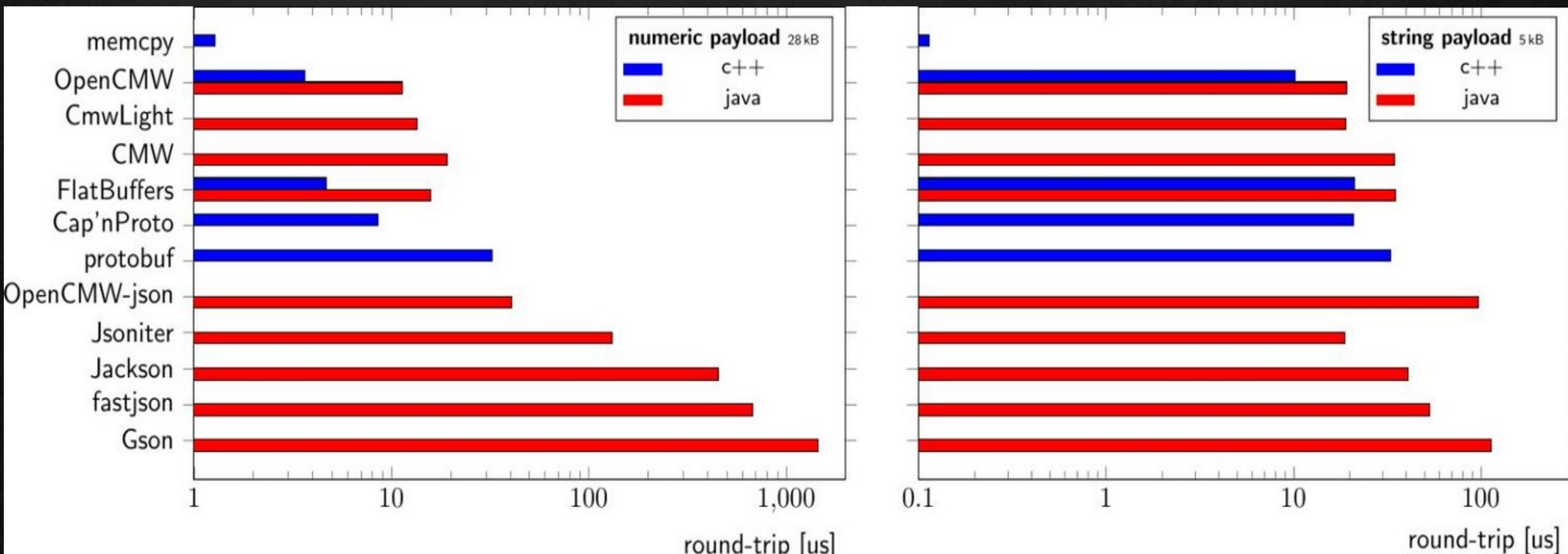
key: independent on and same PoCos for each wire-format

```
# map<string,string>
```

- **use-case: human-readable/-editable microservice config files (stored on GSI's GitLab)**

Serialiser performance

latency & bandwidths



C++ benchmark output:

protocol	ALWAYS	LENIENT	IGNORE
YAML	84.3 MB/s ± 8.1 MB/s	83.6 MB/s ± 427.6 kB/s	86.3 MB/s ± 539.8 kB/s
Json	283.2 MB/s ± 1.6 MB/s	283.0 MB/s ± 1.8 MB/s	287.0 MB/s ± 854.8 kB/s
YaS	6.0 GB/s ± 33.0 MB/s	6.3 GB/s ± 22.6 MB/s	7.7 GB/s ± 47.9 MB/s
CmwLight	8.0 GB/s ± 39.8 MB/s	8.2 GB/s ± 24.9 MB/s	9.3 GB/s ± 104.0 MB/s

7. & 9. Overall Design

minimise code-bloat & free- and open-source

- lean-programming paradigm
  - removing wastes by design
    → improves performance through reduced L1/L2/L3 code cache sizes
  - minimises dependencies & maintenance overhead
  - improves likeliness of others to use, fix, and/or contribute
    - enabling factors: C++20, constexpr, meta-programming, compile-time reflection, unit-safety

- free- and open-source software w/o strings-attached
  - prerequisite for any scientific reproducibility, verifiability … or peer review
  - building a eco-system – op. integrated, exchangeable & transferable both across different labs as well as applications
  - public-private partnerships & "building people & systems"

References & Good Reads

- https://en.wikipedia.org/wiki/Lean_software_development
- Brian Goetz, "Towards Better Serialization", 2019
- Pieter Hintjens et al., "ZeroMQ's ZGuide on Serialisation" in 'ØMQ - The Guide', 2020
- Google's own Protobuf Serialiser
- Google's own FlatBuffer Serialiser
- Implementing High Performance Parsers in Java
- Is Protobuf 5x Faster Than JSON? (Part 1, Part 2) and reference therein

That's all

Questions? Feedback?

Appendix

Serialiser performance

test-class

```
struct TestDataClass {
    // basic data types
    bool   bool1;
    bool   bool2;
    int8_t byte1;
    int8_t byte2;
    char   char1;
    char   char2;
    short  short1;
    short  short2;
    int    int1;
    int    int2;
    long   long1;
    long   long2;
    float  float1;
    float  float2;
    double double1;
    double double2;
    string string1;
    string string2;

    // 1-dim arrays
    vector<char>   boolArray;
    vector<int8_t> byteArray;
    vector<int16_t> shortArray;
    vector<int32_t> intArray;
    vector<int64_t> longArray;
    vector<float>   floatArray;
    vector<double>  doubleArray;
    vector<string>  stringArray;

    // generic n-dim arrays - N.B. striding-arrays: low-level format is the same except of 'nDimension' descriptor
    array<int, 3>  nDimensions;
    vector<char>   boolNdimArray;
    vector<int8_t> byteNdimArray;
    vector<int16_t> shortNdimArray;
    vector<int32_t> intNdimArray;
    vector<int64_t> longNdimArray;
    vector<float>   floatNdimArray;
    vector<double>  doubleNdimArray;

    unique_ptr<TestDataClass> nestedData;// nested class
};
ENABLE_REFLECTION_FOR(TestDataClass, bool1, bool2, …)
```

Compile-Time Reflection

Example based upon: https://github.com/veselink1/refl-cpp (C++17)

```
struct className {
    int         field1;
    float       field2;
    std::string field3;
    bool operator==(const className &) const = default;
};
ENABLE_REFLECTION_FOR(className, field1, field2, field3)  // needed until C++23/26

// here with additional unit type-safety checks (see also: constexpr, strict type-checking & mp-units)

using opencmw::Annotated;
struct otherClass {
   Annotated<float, thermodynamic_temperature<kelvin>, "device specific temperature">      temperature     = 23.2F;
   Annotated<float, electric_current<ampere>, "this is the current from ...">              current         = 42.F;
   Annotated<float, energy<electronvolt>, "SIS18 energy at injection before being captured"> injectionEnergy = 8.44e6F;
   std::unique_ptr<NestedClassType>                                                        nested;
   // [..]

   // just good common practise to define some operators
   bool operator==(const otherClass &) const = default;
};
ENABLE_REFLECTION_FOR(otherClass, temperature, current, injectionEnergy, nested)
```