ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUPV009

OpenCMW - A MODULAR OPEN COMMON MIDDLE-WARE LIBRARY FOR EQUIPMENT- AND BEAM-BASED CONTROL SYSTEMS AT FAIR

Ralph J. Steinhagen, H. Bräuning, D. Day, A. Krimm, T. Milosic,
D. Ondreka, A. Schwinn, GSI Helmholtzzentrum, Darmstadt, Germany

Abstract

OpenCMW is an open-source modular event-driven micro- and middle-ware library for equipment- and beam-based monitoring as well as feedback control systems for the FAIR Accelerator Facility.

Based on modern C++20 and Java concepts, it provides common communication protocols, interfaces to data visualisation and processing tools that aid engineers and physicists at FAIR in writing functional high-level monitoring and (semi-)automated feedback applications.

The focus is put on minimising the required boiler-plate code, programming expertise, common error sources, and significantly lowering the entry-threshold that is required with the framework. OpenCMW takes care of most of the communication, data-serialisation, data-aggregation, settings management, Role-Based-Access-Control (RBAC), and other tedious but necessary control system integrations while still being open to expert-level modifications, extensions or improvements.

ARCHITECTURE

OpenCMW is a light-weight modular middle-ware twin-library that combines ØMQ, REST and micro-service design patterns illustrated in Fig. 1. The Majordomo Protocol Broker (MDP) provides reliable (a)synchronous request-reply as well as publish-subscribe (and related radio-dish) communication patterns between external clients and workers, implementing the business logic that may reside either internally or externally to the MDP process. Its core relies primarily on the high-performance and low-latency ØMQ-based transport layer but can, for example, be optionally extended by HTTP to also support REST or HTML-based communication patterns, and optionally secure worker access via RBAC [1–16].

OpenCMW strongly embraces lean-code principles and hence aims at minimising boiler-plate code and to aid light-weight development of network-based, semi- to fully automated real-time feedback applications. It thus provides template implementations for common tasks such as to

a) aggregate, synchronise, and to sanitise data received from multiple devices,

b) allow to inject custom domain-specific user-code to numerically post-process the received data, and

c) derive control signals and forward these to other services using common communication libraries.

This business logic is implemented by workers that cover:

- optional class-based domain-object definitions for the input parameter and return value, and
- two event-handler interfaces that are registered either with the MDP or Event Store, and that implement the call-back functions further described below.

Notably, developers do not need to rely on IDL-type or other proprietary definitions that, based on our experience, may become hard to synchronise and maintain in later development iterations. The interfaces are defined by standard POCO or POJO domain-objects that OpenCMW analyses using `constexpr` compile-time (C++) or run-time (Java) reflection, further described below.

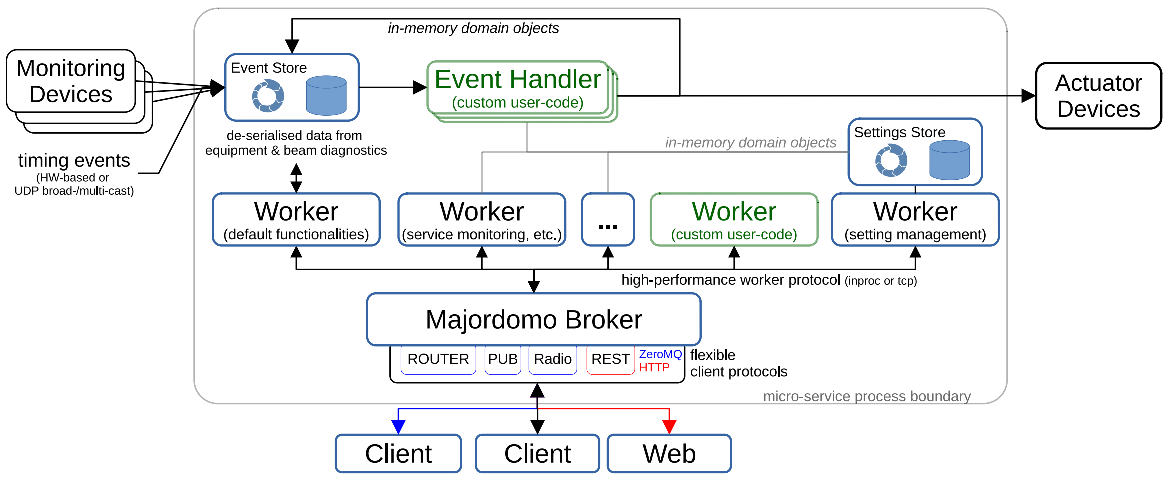

Figure 1: OpenCMW micro-service system architecture combining the Majordomo and event-sourcing design pattern.

DEVICE MODELLING

FAIR provides a wide variety of particle beams with different properties and extraction parameters that are transported to multiple experiments or storage rings either time-multiplexed or even simultaneously. Accelerator operation is thus organised into semi-repetitive 'patterns' and nested substructures (aka. 'timing contexts' or '<ctx>') that are triggered by timing events in order to orchestrate the required rapid real-time succession of individual actions, data acquisition, or device reference settings changes.

OpenCMW is flexible with its domain-object modelling and supports arbitrary setting-types from single scalar fields up to complex nested tree-like data-structures. However, based on operational experience at GSI, CERN and other accelerator facilities, and to be compatible with the existing implementations, most equipment is modelled using a reduced flat 'device/property' scheme only [21]. This scheme models settings essentially through ASCII-based endpoints, that include filters and loosely follows the more commonly-known RFC 1808 URI definition [22]:

```
<device>/<property>?ctx=<ctx>;
  <filter_i=val_j>; ... ;<filter_n=val_m>
```

with the ordering of the optional context selector and filters after the initial '<device>/<property>' being arbitrary and separated by a semi-colon (';') or ampersand ('&').

DATA AGGREGATION

Most monitoring and feedbacks are rarely SISO but more commonly MIMO systems. OpenCMW thus provides facilities that aid in the aggregation and synchronisation of input data that may arrive through different transport channels, protocols, and from a large number of equipment- or beam-based systems. To process this data efficiently, as well as to simplify and reduce potential error sources for developers, two aggregation strategies are provided:

a) continuous sample-by-sample or chunked data not aligned to a <ctx>: these are typically processed by a GNU-Radio-based flow-graph, commonly used in the domain of software-defined-radios (SDRs [17]), as illustrated in Fig. 2;

b) chunked data that is already (partially) aligned to a timing <ctx>: these are typically processed using the event-sourcing pattern that combines the data across multiple input sources or larger superordinate <ctx> or to perform signal post-processing routines on a chunk-by-chunk basis as illustrated in Fig. 3.

The event-sourcing relies on different types of workers:

- 'adapter': tasked with receiving and de-serialising the specific (possibly foreign) wire-format, and storing the resulting domain-object alongside its meta-information into the event store's event stream;
- '<ctx>-matcher': collecting multiple source domain-objects into one combined new 'aggregate' which is usually triggered by timing events or arrival of a new <ctx>. The new aggregate is stored inside the same (or optionally another) event store event/stream, either once all required data arrived or once a configurable time-out w.r.t. the aggregation start is reached;
- post-processing workers: performing the actual post-processing and control actions based on user-supplied worker handler-callback code;

The handler-callbacks are triggered whenever a new aggregate is written to the event store, receive the actual and past issuing events, data to facilitate FIR- and IIR-type filtering, and may notify the MDP to publish the intermediate results.

DESIGN CONSIDERATIONS

There have been many expensive software failures, accidents, and inefficiencies in the past, most famously the Ariane flight 501 and Mars Climate Orbiter that failed due to avoidable type- and physical-unit conversion errors [23, 24]. Our accelerator domain is not immune to these type of failures, and struggles in addition with organically grown code-bases that often – as a consequence of Conway's law[1], time- and resource constraints – cannot be adequately pruned, refactored or optimised. OpenCMW thus aims at lean, modern, and long-term sustainable architecture that avoids the above mistakes by design.

[1] "[Developers will] produce a design whose structure is a copy of the organization's communication structure", [25]

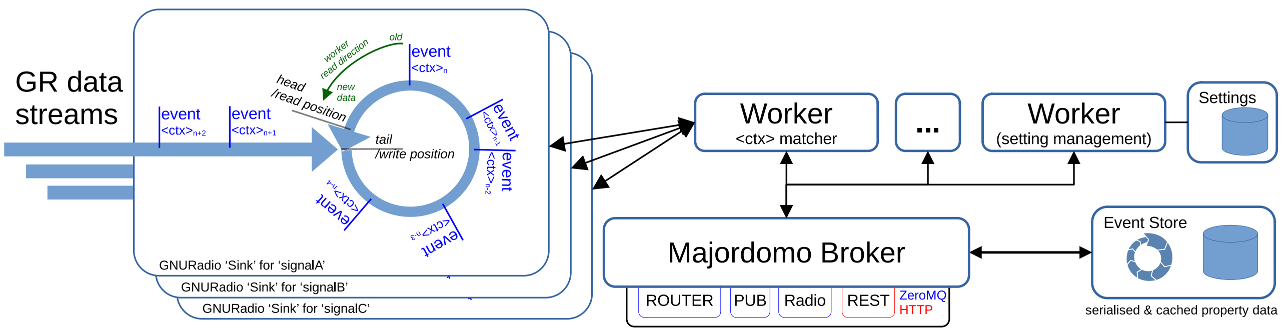

Figure 2: Schematic GNU-Radio-based (GR) processing of continuous data as used by the FAIR Digitizers [17–20]. The data processed by the flow-graph is stored in non-blocking circular buffers (sinks) similar to tagged b-trees that allow to efficiently chunk the data according to the required timing <ctx> scope.

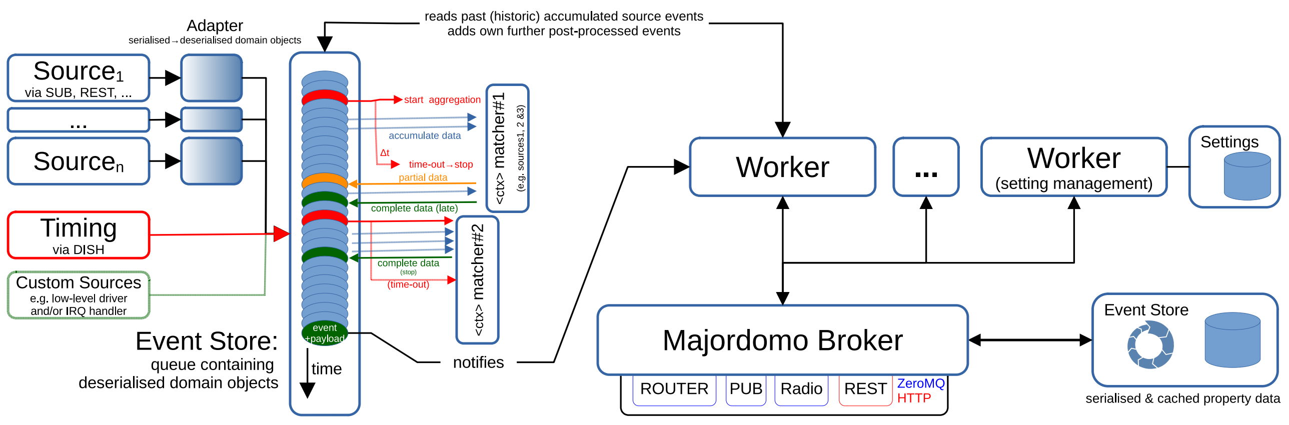

Figure 3: Schematic <ctx> multiplexed-based data aggregation based on the event-sourcing design pattern [11].

The transport layer and data serialisation are basic core functionalities of OpenCMW or any middle-ware for transmitting, storing and later retrieving information by (often quite) different subsystems. With a multitude to choose from, a non-negligible subset of frameworks claim to be the fastest, most efficient, easiest-to-use, etc. A fair comparison of their performance is rather complex, highly non-trivial, and thus subjective because the underlying assumptions of 'what counts as important' is quite different between specific domains.

Instead, we want to document the rationale as to why we decided to implement a new middle-ware, its considerations, constraints, technical core choices, and goals that went into OpenCMW's design in the hope that it might be interesting, perhaps be adopted, inspire new ideas, or any other form of improvement. OpenCMW's core paradigms, loosely ordered according to their importance, are:

1. undogmatic lean and agile software design principles.
2. performance: end-to-end real-time latency minimisation between the service's data object being ready, serialised, and sent, until it is received, fully de-serialised, and ready for further processing by the client[2].
3. loose coupling between server- and corresponding client-side-definitions to provide soft sustainable upgrade paths.
4. strict separation of concern between the communication transport layer, serialiser protocol, and user supplied custom-code: OpenCMW provides extendable interfaces to transparently exchange or add new transport and serialisation protocols simplifying the custom user-code that does not need to be aware of the actual protocols that the client may request. This enables more modular implementation, easier maintenance, and efficient code-sharing of code blocks between different projects.
5. static compile-time reflection[3][26, 27]: derive interfaces directly from C++ or Java domain-objects:
   - less error-prone and more efficient compared to other IDL-based solutions that need to rely on external code-generation tools.
   - greatly reduces the cognitive complexity, see List. 1 and the appendix for code examples.
   - improves productivity of new, occasional, or less-experienced developers who need to be only "vaguely" familiar with C++ or Java.
   - supports UTF8, all primitive/fundamental data types, nested objects, common data containers.
   - efficient (first-class) support of large collections of numeric (floating-point) data.
6. strong type- and physical-units safety [28–30].
7. OpenAPI [31]: self-documented data-structures with optional `constexpr` data field annotations to communicate the service's data-exchange-API intent to the client (e.g. physical unit, access constraints, human-understandable description, etc.). This meta-information can be used to re-generate the domain-objects on the client-side.
8. Modern C++20 [32] and Java 11 standards:
   - more concise with respect to expressing developer's intent and constraints.
   - easier, safer, and more intuitive syntax for new or occasional developers.
   - better performance, due to larger proliferation of compile-time `constexpr` and `consteval` expressions and library functionalities.
9. minimise external library dependencies: especially rely on those already or targeted to become official part of the C++ ISO standard [26, 28, 32].
10. minimise code-base and code-bloat while providing hard-to-implement core-functionalities: small code fits more likely into CPUs's cache and thus results in faster execution. Also from a maintenance perspective, more code requires more time to read and understand, and is harder to modify, test, or fix.

[2] Network-IO bandwidth limit driven trade-offs between size and en-/decoding speed are not of primary concern for our use-case

[3] N.B. Java is presently limited to run-time reflection.

```cpp
struct CppDomainObjectExample {
  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:string                                                                                 notAnnotated    = "Hello World!";
  // [..]
};
// refl-cpp-based: targeted and becomes obsolete with the next C++ standard
ENABLE_REFLECTION_FOR(CppDomainObjectExample, temperature, current, injectionEnergy, notAnnotated)
```

```java
public class JavaDomainObjectExample {
  @MetaInfo(description = "device specific temperature", unit = "K")
  public float   temperature     = 23.2f;
  @MetaInfo(description = "this is the current from ...", unit = "A")
  public float   current         = 42.f;
  @MetaInfo(description = "SIS18 energy at injection before being captured", unit = "eV")
  public float   injectionEnergy = 8.44e6f;
  public String  notAnnotated    = "Hello World!";
  // [..]
}
```

Listing 1: C++ POCO and equivalent Java POJO domain-object reflection example.

11. unit-test and CI/CD driven development to minimise errors and detect potential (also quantitative performance) regressions.
12. free- and open-source LGPLv3-licensed code-base.

It is important to us that this code can be re-used, built- and improved-upon by anybody and thus we hope that the above paradigms lower the entry-level and improve the likelihood of OpenCMW to be adopted also by external users that wish to understand, upgrade, or bug-fix 'what is under the hood' or what is of specific interest to them.

PERFORMANCE

In real-time control and notably for feedback systems, performance depends not only on correct results but also when it is applied to the accelerator or presented to the operators. The required maximum latency that is to be minimised, corresponding group-delay and phase-lag depends on the specific application. OpenCMW's primary use-case are high-level software-based monitoring and (semi-)automated feedback applications at FAIR with bandwidths of up to 25 Hz and latencies well below 10 ms. The framework itself should contribute only with a small proportion to these latencies.

Since the OpenCMW architecture by design strongly decouples, and since there are a wide range of possible combinations, we independently benchmarked the transport-layer only using small dummy-payloads and serialisation without the transport layer.

Majordomo Broker

The performance results for our Java-based Majordomo implementation for both MDP in-process (InProc) and external worker (TCP, via *localhost*) is shown in Table 1. The performance is largely determined by the specific ØMQ implementation (*JeroMQ*) in the above case [33], polling loop optimisation, and by itself contributes very little to the overall latency. This is deemed sufficient for our few Java-based applications with lower latency requirements. The C++-based Majordomo implementation, based on *libzmq* and loosely on the MDP reference implementation [34, 35], aims at latencies below 1 ms to be compatible with FAIR's SDR applications [17–20]. This is presently being optimised and will be released soon.

Table 1: OpenCMW-Java Majordomo Performance (Test-System: AMD Ryzen 9 5900X)

ØMQ Transport Type	msgs/s
REQ/REP synchronous via TCP	12594
REQ/REP synchronous via InProc	27247
REQ/REP asynchronous via TCP	285714
REQ/REP asynchronous via InProc	400000
SUBSCRIBE via SUB-Socket	1538461
SUBSCRIBE via DEALER-socket	666666

Serialiser

OpenCMW aims to provide interfaces for service-client communication where: a) high-performance is important – usually relying on binary-type (de-)serialisation wire-formats – and b) where ease of access, notably, REST and Web-based compatibility is important, often relying on string-based wire-formats such as JSON, YAML, XML, etc. Figure 4 shows the round-trip performance results for OpenCMW and various other common implementations for both C++ and Java [1, 2, 36–42].

While some implementations provided only map-style interfaces, each serialiser has been extended and optimised to (de-)serialise the same equivalent domain-object back-to-back to have more comparable results with the compile-time reflection-based approach. A large spread within but also between binary- and string-based wire-format encoding libraries is visible. The 'CmwLight' serialiser is OpenCMW's functional open-source re-implementation of the existing proprietary CMW-Data protocol that is presently used internally at GSI[21]. The benchmark is available at [43]. Pull-Requests to improve the OpenCMW and other serialisers are welcome. Tentative results indicate that many of the serialiser supporting binary wire-formats seem to be optimised for simple data structures that are typically much smaller than 1k Bytes rather than large numeric ar-

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUPV009

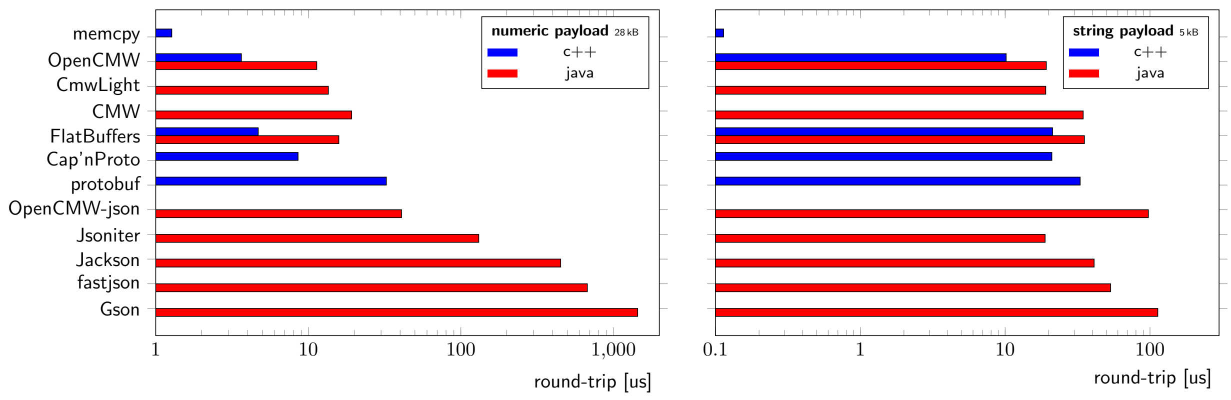

Figure 4: Round-trip performance for converting the input- to output-domain-object via the given wire-format for numeric- (left) and `std::string`-data (right) dominated domain-objects (N.B. log-scale, smaller is better).

rays that were either slow to encode and/or required custom serialiser extensions. *Jsoniter* showed the best results for parsing JSON in Java [39]. We thus decided to use it as the parsing back-end for the *OpenCMW-JSON* deserialiser also to avoid common mistakes and error-handling when processing incomplete or erroneous JSON streams that the deserialiser must be able to handle. Since these error types are non-issues for the serialiser side, we adapted a custom JSON serialiser implementation to the compile-time reflection approach which proved to be more performant. This is also the reason for the visible performance differences between *OpenCMW-JSON* and *jsoniter* in Fig. 4. At the time of the test, the C++ *simdjson* library was not yet evaluated but appears promising and will thus be included in future benchmarks [44].

DOMAIN API EVOLUTION

An important consideration for every distributed system is how to be able to update the API defining domain-objects without having to simultaneously update every other component of the system. To allow for a reliable update strategy OpenCMW uses the transferred metadata to validate the data during deserialisation.

The MDP broker (or any other caller of the serialiser) can control if metadata should be serialised, the type of checks that should be performed on deserialisation, and how errors should be treated. The behaviour of the error handling can be controlled with domain-object annotations.

To optimise the performance impact for high update rates, OpenCMW assumes domain objects stay identical during one connection and only performs checks on the first update. The MDP client calls the verbose deserialiser on the first update and the fast deserialiser on subsequent updates. If there are differences in the first update it can:

- ignore compatible changes like new optional fields or missing optional fields. The client code can still check for the presence of optional fields through the return value of the serialiser.
- request a fallback property if available and warn the user of the legacy API.
- throw an exception.

Worker implementations should keep future extensions in mind when defining their domain objects, but not at the cost of a clean design. When changes are necessary, new (non-mandatory) fields can be added in a backward compatible way if they don't change the meaning of the other data. If non-compatible changes like changing data types are performed, a fallback property providing the old domain object should be offered in the 'two on production' pattern[45, 46]. It is encouraged to also make use of the 'aggressive obsolescence' and 'experimental preview' patterns, to keep the burden of API maintenance manageable. There is no explicit versioning of the domain objects, mismatches in domain objects are determined by the deserialiser.

CONCLUSION

OpenCMW is a new open-source modular event-driven micro-service and middle-ware library for equipment- and beam-based monitoring as well as feedback control systems for the FAIR Accelerator Facility.

Based on modern C++20 and Java concepts, together with the benefits and chances of an agile open-source development, we believe that the above lean software development principles are key ingredients and will help experienced as well as entry-level engineers and physicists to write, share, and improve functional and long-term maintainable code for high-level monitoring and (semi-)automated feedback applications both at FAIR, other laboratories as well as used by other interested developers elsewhere. For detailed information, code examples, or to participate in this project, please visit and interact with the authors via [1–3].

ACKNOWLEDGEMENTS

The implementation heavily relies upon and re-uses time-tried and well-established concepts from ØMQ (notably the Majordomo communication pattern [8]), LMAX's lock-free ring-buffer disruptor [11], GNU-Radio real-time signal processing framework [17], Javalin for the RESTful interface [47], as well as previous implementations and experiences gained at GSI, FAIR and CERN [21, 48–57].

APPENDIX

```java
package io.opencmw.server.rest.samples;

import static io.opencmw.OpenCmwProtocol.EMPTY_URI;

import java.io.IOException;
import java.net.URI;
import java.util.Objects;
import java.util.Timer;
import java.util.TimerTask;
import java.util.concurrent.TimeUnit;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.zeromq.ZContext;

import io.opencmw.domain.NoData;
import io.opencmw.rbac.BasicRbacRole;
import io.opencmw.rbac.RbacRole;
import io.opencmw.serialiser.annotations.MetaInfo;
import io.opencmw.server.MajordomoBroker;
import io.opencmw.server.MajordomoWorker;
import io.opencmw.server.rest.MajordomoRestPlugin;

import zmq.util.Utils;

public class BasicSample {
    private static final Logger LOGGER = LoggerFactory.getLogger(BasicSample.class);

    public static void main(String[] argv) throws IOException {
        final MajordomoBroker broker = new MajordomoBroker("PrimaryBroker", EMPTY_URI, BasicRbacRole.values());
        final URI brokerRouterAddress = broker.bind(URI.create("mdp://*:" + Utils.findOpenPort()));
        final URI brokerSubscriptionAddress = broker.bind(URI.create("mds://*:" + Utils.findOpenPort()));
        broker.start();
        new MajordomoRestPlugin(broker.getContext(), "Test HTTP/REST Server", "*:8080").start();
        // instantiating and starting custom user-service
        new HelloWorldWorker(broker.getContext(), "helloWorld", BasicRbacRole.ANYONE).start();
    }

    @MetaInfo(description = "My first 'Hello World!' Service")
    public static class HelloWorldWorker extends MajordomoWorker<BasicRequestCtx, NoData, ReplyData> {
        public HelloWorldWorker(final ZContext ctx, final String serviceName, final RbacRole<?>... rbacRoles) {
            super(ctx, serviceName, BasicRequestCtx.class, NoData.class, ReplyData.class, rbacRoles);

            // the custom used code:
            this.setHandler((rawCtx, requestContext, requestData, replyContext, replyData) -> {
                final String name = Objects.requireNonNullElse(requestContext.name, "");
                LOGGER.atInfo().addArgument(rawCtx.req.command).addArgument(rawCtx.req.topic)
                    .log("{} request for worker - requested topic '{}'");
                replyData.returnValue = name.isBlank() ? "Hello World" : "Hello, " + name + "!";
                replyContext.name = name.isBlank() ? "At" : (name + ", at") + " your service!";
            });

            // simple asynchronous notify example - (real-world use-cases would use another updater than Timer)
            new Timer(true).scheduleAtFixedRate(new TimerTask() {
                private final BasicRequestCtx notifyContext = new BasicRequestCtx(); // re-use to avoid gc
                private final ReplyData notifyData = new ReplyData(); // re-use to avoid gc
                private int i;
                @Override
                public void run() {
                    notifyContext.name = "update context #" + i;
                    notifyData.returnValue = "arbitrary data - update iteration #" + i++;
                    try {
                        HelloWorldWorker.this.notify(notifyContext, notifyData);
                    } catch (Exception e) {
                        // further handle exception if necessary
                    }
                }
            }, TimeUnit.SECONDS.toMillis(1), TimeUnit.SECONDS.toMillis(2));
        }
    }

    @MetaInfo(description = "arbitrary request domain context object", direction = "IN")
    public static class BasicRequestCtx {
        @MetaInfo(description = " optional 'name' OpenAPI documentation")
        public String name;
    }

    @MetaInfo(description = "arbitrary reply domain object", direction = "OUT")
    public static class ReplyData {
        @MetaInfo(description = " optional 'returnValue' OpenAPI documentation", unit = "a string")
        public String returnValue;
    }
}
```

REFERENCES

[1] FAIR. 'OpenCMW C++ Project'. (2021), https://github.com/fair-acc/opencmw-cpp
[2] FAIR. 'OpenCMW Java Project'. (2020), https://github.com/fair-acc/opencmw-java
[3] FAIR. 'OpenCMW Gitter Forum'. (2021), https://gitter.im/fair-acc/opencmw
[4] A. Krimm and R. Steinhagen, 'FAIR Common Specification - Modular Open Common Middle-Ware Library for Equipment- and Beam-Based Control Systems of the FAIR Accelerators', FAIR, Tech. Rep., 2020. https://edms.cern.ch/document/2444348
[5] P. Hintjens, *ZeroMQ*. O'Reilly Media, Inc., 2013, ISBN: 9781449334062. https://zguide.zeromq.org
[6] ZeroMQ, 'ZeroMQ home-page', The ZeroMQ Project, Tech. Rep., 2020. https://zeromq.org/
[7] ZeroMQ, 'ZeroMQ Project Respository', The ZeroMQ Project, Tech. Rep., 2013-2020. https://github.com/zeromq
[8] P. Hintjens, 'Majordomo Protocol RFC', The ZeroMQ Project, Tech. Rep., 2012. https://rfc.zeromq.org/spec/18/
[9] P. Hintjens, 'ZeroMQ Publish-Subscribe RFC', The ZeroMQ Project, Tech. Rep., 2014. https://rfc.zeromq.org/spec/29/
[10] D. Somech, 'ZeroMQ Radio-Dish RFC', The ZeroMQ Project, Tech. Rep., 2020. https://rfc.zeromq.org/spec/48/
[11] M. Thompson, D. Farley, M. Barker, P. Gee and A. Stewart. 'LMAX Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads'. (2011), https://lmax-exchange.github.io/disruptor/
[12] National Institute of Standards and Technology, 'Role Based Access Control - RBAC', NIST, Tech. Rep., 2020. https://csrc.nist.gov/projects/role-based-access-control/faqs
[13] D. F. Ferraiolo, D. R. Kuhn and R. Chandramouli, *Role-Based Access Control*, 2nd. USA: Artech House, Inc., 2007, ISBN: 1596931132.
[14] R. T. Fielding, 'Architectural Styles and the Design of Network-based Software Architectures - Chapter 5: Representational State Transfer (REST)', Ph.D. dissertation, University of California, Irvine, 2000. https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf
[15] T. Berners-Lee and D. Connolly, *Hypertext markup language — 2.0*, Internet RFC 1866, Nov. 1995.
[16] R. Fielding *et al.*, 'RFC 2616, Hypertext Transfer Protocol – HTTP/1.1', RFC Editor, Tech. Rep., 1999. http://www.rfc.net/rfc2616.html
[17] GNU Radio Website. (accessed May 2020), http://www.gnuradio.org
[18] R. J. Steinhagen *et al.*, 'Generic Digitization of Analog Signals at FAIR – First Prototype Results at GSI', in *Proc. 10th International Particle Accelerator Conference (IPAC'19), Melbourne, Australia, 19-24 May 2019*, (Melbourne, Australia), ser. International Particle Accelerator Conference, Geneva, Switzerland: JACoW Publishing, Jun. 2019, pp. 2514–2517, ISBN: 978-3-95450-208-0. DOI: 10.18429/JACoW-IPAC2019-WEPGW021. http://jacow.org/ipac2019/papers/wepgw021.pdf
[19] FAIR. 'gr-digitizers repository'. (2021), https://github.com/fair-acc/gr-digitizers
[20] FAIR. 'gr-flowgraph repository'. (2021), https://github.com/fair-acc/gr-flowgraph
[21] J. Lauener and W. Sliwinski, 'How to design & implement a modern communication middleware based on ZeroMQ', in *Proceedings of ICALEPCS'2017*, (Barcelona, Spain), JACoW, Ed., ser. International Conference on Accelerator and Large Experimental Control Systems, 2017, MOBPL05. 7 p. DOI: 10.18429/JACoW-ICALEPCS2017-MOBPL05. https://cds.cern.ch/record/2305650
[22] R. T. Fielding, 'Relative Uniform Resource Locators', RFC Editor, Tech. Rep. 1808, Jun. 1995, 16 pp. DOI: 10.17487/RFC1808. https://rfc-editor.org/rfc/rfc1808.txt
[23] J.-L. L. et al, *ARIANE 5 Flight 501 Failure: Report by the Enquiry Board*. European Space Agency — ESA, Jul. 1996. https://esamultimedia.esa.int/docs/esa-x-1819eng.pdf
[24] A. G. Stephenson *et al.*, *Mars Climate Orbiter Mishap Investigation Board: Phase I Report*. Jet Propulsion Laboratory (U.S.), United States. National Aeronautics and Space Administration, Nov. 1999. https://llis.nasa.gov/llis_lib/pdf/1009464main1_0641-mr.pdf
[25] M. E. Conway, 'How do committees invent?', *Datamation*, Apr. 1968. https://www.melconway.com/Home/pdf/committees.pdf
[26] D. Sankel. 'N4856 – C++ Extensions for Reflection'. (Mar. 2020), http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/n4856.pdf
[27] V. Karaganev. 'refl-cpp — a compile-time reflection library for C++'. (2020), https://github.com/veselink1/refl-cpp
[28] M. Pusz. 'P1935R2 – A C++ Approach to Physical Units'. (Jan. 2020), http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1935r2.html
[29] M. Pusz. 'A Physical Units Library For the Next C++'. (Sep. 2020), https://github.com/CppCon/CppCon2020/blob/main/Presentations/a_physical_units_library_for_the_next_cpp
[30] M. Pusz. 'mp-units - A Units Library for C++'. (2021), https://github.com/mpusz/units
[31] OpenAPI Initiative. 'The OpenAPI Specification'. (2016-2021), https://github.com/OAI/OpenAPI-Specification
[32] ISO, *ISO/IEC 14882:2020: Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization — ISO, 2020, p. 1853. https://www.iso.org/standard/79358.html
[33] ZeroMQ. 'JeroMQ – Pure Java implementation of libzmq'. (2021), https://github.com/zeromq/jeromq
[34] ZeroMQ. 'libzmq – ZeroMQ core engine in C++, implements ZMTP/3.1'. (2021), https://github.com/zeromq/libzmq
[35] ZeroMQ. 'Majordomo Reference Implementation'. (2021), https://github.com/zeromq/majordomo
[36] Renshaw, David and Hancock, Harris and Varda, Kenton. 'Cap'n Proto Library'. (2021), https://github.com/capnproto/capnproto
[37] W. van Oortmerssen et al. 'FlatBuffers'. (2021), https://github.com/google/flatbuffers

[38] S. Ghemawat, J. Dean, D. Dulitz, C. Silverstein, P. Haahr and C. Anderson. 'Protocol Buffers - Google's data interchange format'. (2021), https://github.com/protocolbuffers/protobuf
[39] Tao Wen. 'Json Iterator - Java'. (2021), https://github.com/json-iterator/java
[40] Brown, Paul and Saloranta, Tatu. 'Jackson'. (2021), https://github.com/FasterXML/jackson
[41] Alibaba. 'FastJson Library'. (2021), https://github.com/alibaba/fastjson
[42] 'Gson'. (2021), https://github.com/google/gson
[43] FAIR. 'OpenCMW C++ Serialiser Benchmarks'. (2021), https://github.com/fair-acc/opencmw-benchmarks-cpp
[44] e. a. Daniel Lemire John Keiser. 'simdjson : Parsing gigabytes of JSON per second'. (2021), https://github.com/simdjson/simdjson
[45] D. Lübke, O. Zimmermann, C. Pautasso, U. Zdun and M. Stocker, 'Interface evolution patterns: Balancing compatibility and extensibility across service life cycles', in *Proceedings of the 24th European Conference on Pattern Languages of Programs*, ser. EuroPLop '19, Irsee, Germany: Association for Computing Machinery, 2019. DOI: 10.1145/3361149.3361164.
[46] A. Jesse. 'API evolution the right way'. (2019), https://opensource.com/article/19/5/api-evolution-right-way
[47] D. Aas. 'Javalin - A simple web framework for Java and Kotlin'. (2021), https://github.com/tipsy/javalin
[48] R. J. Steinhagen and R. Bär, 'FAIR Common Specification: Accelerator and Beam Modes', GSI & FAIR, Tech. Rep., 2017. https://edms.cern.ch/document/1823352/
[49] H. Hüther, J. Fitzek, R. Müller and A. Schaller, 'Realization of a Concept for Scheduling Parallel Beams in the Settings Management System for FAIR', in *Proceedings of ICALEPCS'15*, Melbourne, Australia, 2015.
[50] R. J. Steinhagen *et al.*, 'FAIR Common Specification: Digitization of Analog Signals at FAIR', GSI & FAIR, Tech. Rep., 2017. https://edms.cern.ch/document/1823376/
[51] J. Serrano *et al.*, 'White Rabbit: Sub-nanosecond Timing Distribution over Ethernet', in *Proc. of 2009 International Symposium on Precision Clock Synchronization for Measurement, Control and Communication*, Brescia, Italy: IEEE, Oct. 2009, pp. 1–4. DOI: 10.1109/ISPCS.2009.5340196.
[52] S. Deghaye, M. Lamont, L. Mestre, M. Misiowiec, W. Sliwinski and G. Kruk, 'LHC Software Architecture (LSA) – Evolution toward LHC Beam Commissioning', in *Proc. 11th Int. Conf. on Accelerator and Large Experimental Control Systems (ICALEPCS'07)*, Oak Ridge, TN, USA, Oct. 2007, pp. 307–309.
[53] R. Müller, J. Fitzek and D. Ondreka, 'Evaluating the LHC Software Architecture for Data Supply and Setting Management within the FAIR Control System', in *Proc. 12th Int. Conf. on Accelerator and Large Experimental Control Systems (ICALEPCS'09)*, GSI Helmholtzzentrum, Darmstadt, Germany, Kobe, Japan, Oct. 2009, pp. 697–699.
[54] D. Ondreka, J. Fitzek, H. Liebermann and R. Müller, 'Settings Generation for FAIR', in *Proc. of International Particle Accelerator Conference (IPAC'12)*, GSI Helmholtzzentrum, Darmstadt, Germany, New Orleans, Louisiana, USA, May 2012, pp. 3963–3965.
[55] M. Arruat *et al.*, 'Front-End Software Architecture (FESA)', in *Proc. 11th Int. Conf. on Accelerator and Large Experimental Control Systems (ICALEPCS'07)*, Oak Ridge, TN, USA, Oct. 2007, pp. 310–312.
[56] S. Matthies, H. Bräuning, A. Schwinn and S. Deghaye, 'FESA3 Integration in GSI for FAIR', in *Proc. 10th Int. Workshop on Personal Computers and Particle Accelerator Controls (PCaPAC'14)*, Karlsruhe, Germany, Oct. 2014, pp. 43–45.
[57] V. Rapp and W. Sliwinski, 'Controls Middleware for FAIR', in *Proc. 10th Int. Workshop on Personal Computers and Particle Accelerator Controls (PCaPAC'14)*, Karlsruhe, Germany, Oct. 2014, pp. 4–6.