Quality Management	Document Type:	Document Number: F-CS-SIS-en-B-0006	Date: 2020-11-23
FAIR	Common Specification	Template Number: Q-FO-QM-0005	Page 1 of 23

Document Title:	**Modular Open Common Middle-Ware Library for Equipment- and Beam-Based Control Systems of the FAIR Accelerators**
Description:	Detailed functional specification for a modular micro-service middle-ware that shall facilitate the implementation of high-level monitoring (data aggregation), signal-processing and superordinate control loops that cannot be suitably implemented in equipment-based controllers.
Division/Organization:	FAIR
Field of application:	FAIR Project, existing GSI accelerator facility
Version	V 1.0

Abstract

This document describes the functional requirements on a modular event-driven middle-tier micro-service middle-ware library that provides common communication protocols and numerical data visualisation tools that shall aid accelerator engineers and physicist to write high-level monitoring and (semi-) automated feedback applications that use existing equipment- and beam-based information to drive new accelerator device settings and that cannot be suitably implemented in a single equipment-based controller or any other local services.

Prepared by:	**Checked by:**	**Approved by:**
A. Krimm	F. Gressier (QA)	D. Ondreka (SYS)
R. Steinhagen	H. Bräuning (BEA)	P. Spiller (SPL, SIS100/SIS18)
	D. Day (ACO)	R. Steinhagen
	T. Milosic (BEA)	(FAIR Comm. & Control PL)
	D. Ondreka (SYS)	
	A. Schwinn (ACO)	

Document History:				
Version	Prepared by:	Checked, Date	Released, Date	Comment
V 0.1	A. Krimm R. Steinhagen			Initial version based on preparatory work and discussions in the FAIR Commissioning and Control Working Group, ACO and BEA.
V 0.2	R. Steinhagen	2020-10-02		initial internal review
V 0.2	R. Steinhagen	2020-11-11		updated source-code references and schemas
V 1.0	R. Steinhagen	2020-11-16		prepare for release

Table of Contents

1 Purpose and Classification of the Document

The purpose of this document is to summarise the functional requirements on a modular event-driven middle-tier micro-service library middle-ware, communication protocols and numerical data visualisation tools that shall aid accelerator engineers and physicist (further referred to as 'users') to write high-level monitoring and (semi-) automated feedback applications that use existing equipment- and beam-based information to drive new accelerator device settings and that cannot be suitably implemented in a single equipment-based controller or any other local services.

Leveraging the users domain-level-expertise, the micro-service framework shall abstract common functionality, minimise user-level boiler-plate code, lower the required software-design and coding expertise for writing monitoring or feedback applications, and to promote modern open-source architectures and C++ coding standards that can be easily maintained, upgraded and also reused in other fields.

2 Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [1].

OPTIONAL references explaining common non-accelerator specific terminology are in-lined using hyperlinks. References that may be necessary for the implementation and that SHOULD be consulted are listed in the reference section in addition to the hyperlinks.

3 Scope of the technical System

The following functionalities are required:

I. a modular micro-service middle-ware library based on C++ aiding in the development of network-based, semi- to fully automated real-time feedback applications that MUST
   a) aggregate, synchronises and provide functional interfaces to sanitise numerical data received from multiple devices,
   b) allow to inject user-supplied (custom) and domain-specific user-code to numerically post-process or to derive control signals from the received data, and
   c) forward these result to other services using a common communication library.

II. a communication library to be used in the micro-service providing an abstraction, that SHALL be compatible with existing wire-formats and domain objects [2], and implement a (de-)serialisation based directly on the user-supplied domain-object data definitions (C++ class or struct) that are exchanged between micro-services and clients.

III. a scientific charting library focused on performance optimised real-time data visualisation of these pre- and post-processed signals, based on modern C++ standards and that SHOULD be functionally compatible with the existing Java-based Chart-fx implementation [2].

3.1 Overall Micro-Service Architecture

Figure 1 provides a schematic overview for the targeted micro-service architecture.

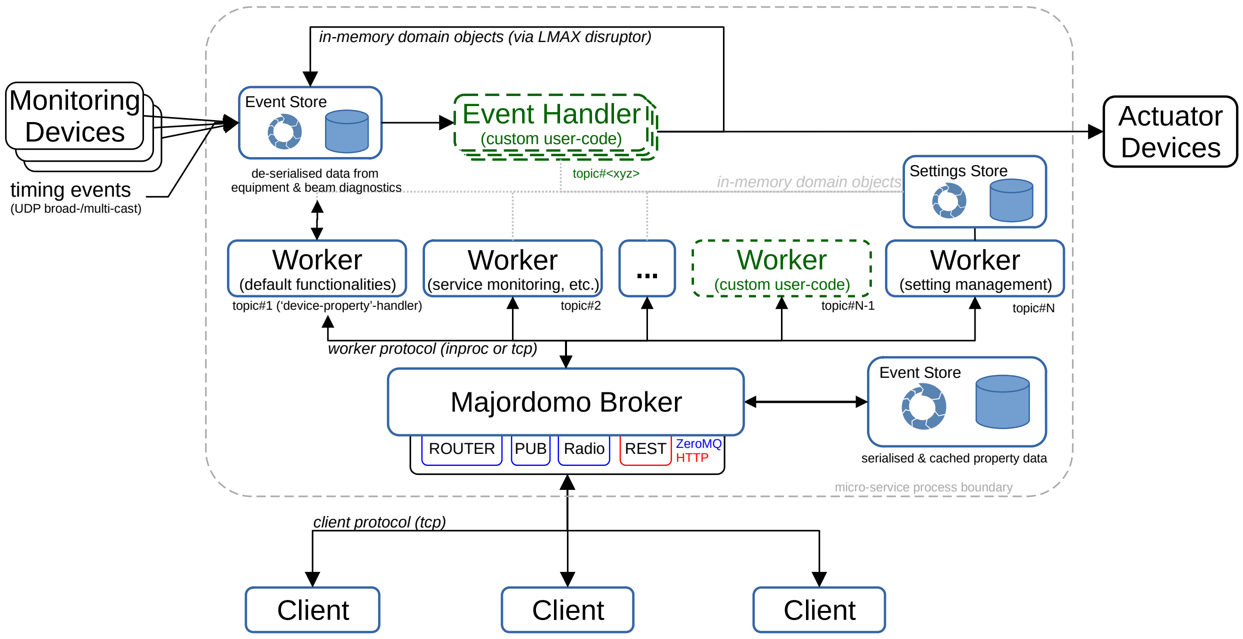


Figure 1: targeted micro-service system architecture which combines the following design and communication patterns: event-sourcing, majordomo (MDP), a Role-Based-Access-Control (RBAC) that may be enabled for critical workers, and publish-subscribe, radio-dish and HTTP-based REST communication channels [ref1,3].

The displayed blocks have the following interpretation:

- 'client': UI applications and/or other micro-services
- 'monitoring devices': sensor devices, hardware equipments and/or other micro-services
- 'actuator devices': the device settings-management-system and/or hardware device actuators
- 'workers': implementations of the (often signal-processing, filtering, or copy) call-back functions for a given service (i.e. 'property' or 'topic')

Workers cover two components:

- a C++-class-based domain-object definitions for the input parameter and return value (optional, in case this is different to the input parameter), and
- two event-handler classes that are registered either with the Majordomo broker or Event Store, and that implement the call-back functions further described below.

For some workers the micro-service middle-ware is expected to provide default implementations. Those workers can reside either inside the same process or in external services that are spawned by the Majordomo broker.

External communication interfaces rely either on one of the ZeroMQ message pattern (e.g. 'ROUTER', 'PUB/SUB', 'RADIO/DISH') or HTTP-based REST interface.

3.2 Multiplexing Modelling

FAIR consists of a cascade of individual particle accelerators, each optimised for a given function or energy reach, and that consist of a large number of distributed devices that perform specific sub-functionalities that are necessary to receive and form particle beams, manipulate, accelerate or decelerate these beams, before transferring them to subsequent accelerators or experiments that utilise or study the particular particles' properties. To guarantee the very strict timing requirements, the control of each accelerator device is governed by the 'time multiplexing' and 'device/property' modelling principles that are important for the internal data structure modelling within the micro-services[1] outlined in the following sections.

3.3 Timing Multiplexing Modelling

The term 'multiplexing' is used to describe the orchestration of sequence of individual actions, data acquisition, or reference settings. It organises the facility operation into 'Patterns' containing further nested substructures (aka. 'timing contexts'): 'Beam-Production-Chains' (BPCs) describing the overall transfer-process for a given particle beam from its source to its experiment destination, and 'Sequences' that group sequences 'Beam-Processes' (BPs).

Beam-Processes are used to model and describe low-level atomic accelerator operations that are being executed, for example, particle injection, acceleration or deceleration, special particle beam manipulations, storage, or fast/slow extraction of these beams to the respective experiments. The nested timing context structure and its nomenclatures is illustrated in Figure 2.

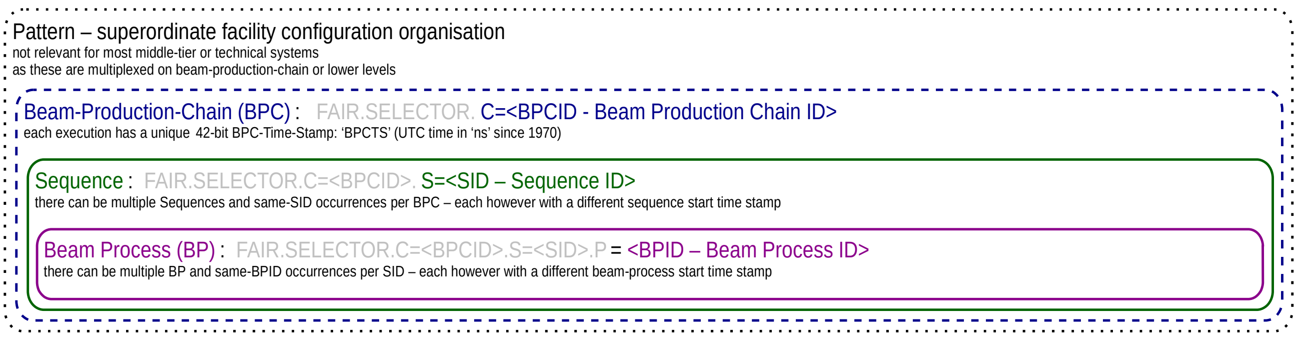

Pattern – superordinate facility configuration organisation
not relevant for most middle-tier or technical systems
as these are multiplexed on beam-production-chain or lower levels
Beam-Production-Chain (BPC) : FAIR.SELECTOR. C=<BPCID - Beam Production Chain ID>
each execution has a unique 42-bit BPC-Time-Stamp: 'BPCTS' (UTC time in 'ns' since 1970)
Sequence : FAIR.SELECTOR.C=<BPCID>. S=<SID – Sequence ID>
there can be multiple Sequences and same-SID occurrences per BPC – each however with a different sequence start time stamp
Beam Process (BP) : FAIR.SELECTOR.C=<BPCID>.S=<SID>.P = <BPID – Beam Process ID>
there can be multiple BP and same-BPID occurrences per SID – each however with a different beam-process start time stamp

Figure 2: facility-level nested multiplexing schematic: this also SHALL guide the multiplexing contexts for actions, settings and grouping of individual (continuous) measurement samples into discrete data chunks (↔ data storage & domain-objects definitions).

1 One may relate these principles largely with directing a symphony orchestra or large music band. In this analogy, the specific music instrument corresponds to a 'device', the instrument's pitch, dynamic range, type-of-stroke etc. corresponding to the 'device-property' (or setting), and the temporal execution (periods, phrases, etc.) corresponding to the 'timing multiplexing'.

The micro-services SHALL facilitate multiplexing based on BPCs, Sequences, or BPs with one of the important tasks being to combine or to split data chunks to the given superordinate- or sub-context. For the micro-services – for practical purposes – the low-level multiplexing timing event structure[2] is described by a simple ASCII-based and case-insensitive filter referred to as 'timing context selector', or 'context selector' for short. It MAY be either empty or MUST follow the form:

'FAIR.SELECTOR.C=<BPCID>:T=<GID>:S=<SID>:P=<BPID>'

The context selector has the following filter sub-components:

- 'FAIR.SELECTOR.' being the static selector prefix for each context,
- 'C=<BPCID>' being a 22bit-integer value carrying the ID of the BPC,
- 'T=<GID>' being a 12bit-integer value carrying the ID of the timing group,
- 'S=<SID>' being a 12bit-integer value carrying the ID of the Sequence, and
- 'P=<BPID>' being a 12bit-value carrying the ID of the BP.

The ordering of the optional filters 'C=', 'T=', 'S=' and 'P=' after the initial selector prefix is arbitrary and separated by a colon (':'). Valid wildcard sub-strings are 'ALL' or a zero-length string.

For example:

- '' (empty string, or 'Null-Selector').
- '*FAIR.SELECTOR.ALL*' or 'FAIR.SELECTOR', both referring to all BPCs,
- '*FAIR.SELECTOR.C=0:S=ALL*' or '*FAIR.SELECTOR.C=0*', both referring a BPC with ID 0,
- 'FAIR.SELECTOR.C=0:S=1', referring to the Sequence with ID 1 within the BPC with ID 0

The full payload of a raw timing message is described by a 256-bits representation. The IDs described above MAY, however, be also represented by simple 32-bit integers within the micro-services if this simplifies the filtering or improves performance. The full raw message SHALL nevertheless be carried along in case other pertinent information details need to be extracted further down the processing chain that might not be strictly necessary for the use within the micro-service.

The timing group filter ('T=') Identifies a group of equipment, such as an accelerator or transfer line. As a simplification, handling of a filter 'T=<GID>' MAY be considered optional or assumed constant during run-time as nearly all source devices belong to only one timing group and thus publish data for one 'T=' filter setting only.

The full timing context selector may be shortened in diagrams or referenced as '<ctx>' for brevity and readability when necessary. An exemplary timing sequence containing multiple BPCs is illustrated in Figure 3.

2 also outlined in optional internal reference 'White-Rabbit timing message payload'

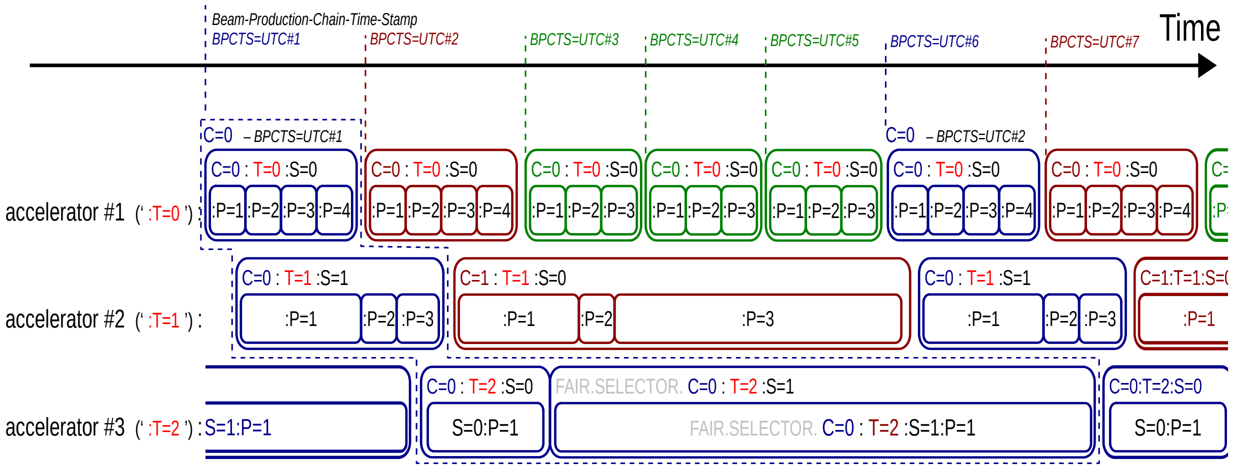


Figure 3: exemplary sequence of Beam-Production-Chains (BPCs): Note that individual BPCs may execute in parallel (ie. partially overlap in time) and are only distinguished by their given BPC-(Start)-Time-Stamp (BPCTS, UTC nanoseconds since 1.1.1970).

Two basic types of input data SHALL be aggregated by the micro-services:

A) continuous time updates of sample-by-sample or chunked data not aligned to a <ctx>.

B) chunked data that is already (partially) aligned to a <ctx>, that needs to be combined across multiple input sources, or that needs to be combined to a larger context (e.g. several BP-multiplexed data to a Sequence) or split into smaller sub-contexts (e.g. BPC-organised data into its BP-constituents).

The aggregation SHALL cover both single data sources as well as grouping of multiple sources into a combined domain-object structure.

IMPORTANT: reference settings are defined and supplied for each BP but for finite lengths only. Thus, for time-continuous data-acquisition there may be 'gaps' between BPs, Sequences or BPCs for which no context definition is formally defined. For example, one BP with fixed length may have finished, while the next BP may have not started yet due to a timing synchronisation pause. While a zero-order-holder algorithm is technically possible (ie. allotting the additional data samples to the previously executed context), it is recommended to store this data in domain-objects that are multiplexed by the next-larger timing context definition (ie. grouping BP → Sequence, or better grouping BP → BPC) and to split and extract the BP-multiplexed data on request by the client (if necessary at all).

3.4 Device/Property Modelling

The ‘device/property’ control paradigm defines how recurring reference settings, data acquisitions, and actions are being modelled within the settings-supply system, accelerator devices, and in extension also within the micro-services. The setting are modelled by a flat hierarchy of ‘<device>/<property>’ endpoints that are described by a simple ASCII-based topics descriptions, that include filters and that are loosely follow the commonly-known ‘path’ definition for the http-scheme described in RFC 1808 (Web-URLs, ZeroMQ or RESTful path ‘topic’ descriptions):

‘<device>/<property>?ctx=<ctx>;<filter$_i$=value$_j$>; … ;<filter$_n$=value$_m$>’

‘<device>/<property>?ctx=<ctx>&<filter$_i$=value$_j$>& … &<filter$_n$=value$_m$>’

The topic description has the following sub-components:

- <device>: a globally unique name of a device or micro-service and which is being resolved by the external name-services to the host’s ‘<ip>:<port>’ the service is running on;
- <property>: a device-specific endpoint (e.g. ‘Setting’, ‘Acquisition’, ‘Status’, etc.) with its field layout being controlled by the underlying domain-object (class or structure) member variable definitions that may contain fundamental-, character-, or floating-point-types as well as vectors, lists or maps thereof (i.e. ‘property-name’==’C++ class-name’ and ‘property-field-name’ == ‘C++ class-member-variable-name’ ↔ see also (de-)serialisation),
- <ctx>: context selector (described above) filtering the returned property notifications only for matching contexts;
- <filter$_i$=value$_j$>: arbitrary user-defined name-value pairs that facilitate the filtering of information which is being exposed to the client or specific user-action modifier, for example, ‘channelNameFilter=signalA, signalC’ instructing the micro-service that the returned serialised property data should contain only data for ‘signalA’ and ‘signalC’.

The ordering of the optional context selector and the optional filters after the initial ‘<device>/<property>’ is arbitrary and separated by a semi-colon (‘;’) or ampersand (‘&’). The domain-objects that are exchanged with external clients SHOULD prefer and maintain a flat hierarchy. Internal domain-objects MAY use tree-like structures if this simplifies the data management, notably with respect to handling timing multiplexing. Each domain-object and field within the domain-object may be adorned with additional optional static compile-time meta-information indicating, for example:

- the field unit,
- a brief field (or property) usage description,
- ‘SET group’, whether the field/property is read-write or read-only,
- whether the property provides subscription via the default (X)PUB and/or DISH socket, or
- whether the property is protected by a Role-Based-Access (RBAC) control mechanism.

These compile-time meta-information SHALL be used as part of the compile-time (de-)serialisation and incorporated into the wire-format that is transmitted to external clients.

External clients can retrieve, monitor or manipulate these domain-objects through the following commands:

- GET: an (a-)synchronous re read command returning either the requested property object based on the provided topic string as defined above, or forwarding an exception in case of a communication or internal micro-service failure.
- SET: an (a-)synchronous write command to the micro-service that sends a complete property domain-object corresponding to the targeted topic and replacing the present domain-object completely (i.e. containing all fields) with the new object, and that is either acknowledged with a simple 'OK' or an exception similar to the 'GET' command.
- PARTIAL-SET: a command similar to 'SET' but allowing to specify only a sub-set of domain-object fields that shall be modified in the corresponding domain-object held by the micro-service. The micro-service SHALL verify whether the partial information is complete (i.e. all fields of a mandatory sub-set being present, as defined by the domain-object's meta-data) and numerically consistent (i.e. by execution of a user-supplied validation function) prior to applying the new values to the domain-object internal to the micro-service.
- SUBSCRIBE (& UNSUBSCRIBE) to start (stop) monitoring of notified changes or updates of a given topic following ZeroMQ's RFC 29/PUB/SUB and RFC 48/RADIO/DISH[3] socket patterns. Publications of the device/property domain-objects via a (X)PUB sockets are triggered in response to either an active 'notify' event (e.g. after a background process finished the post-processing step) or after a '(PARTIAL-)SET' event has been successfully completed. The 'notify' event SHALL trigger the existing 'GET' callback function that may have a basic default (e.g. streaming the unmodified domain-object associated to the property and given context filter) or user-supplied implementation.

In addition to the public 'GET' and '(PARTIAL-)SET' handler, a second type of worker SHALL implement

- an internal 'EVT-callback' handler that is invoked in response to trigger events by other workers or modules via the event store, and executes the user-provided post-processing function. In order to facilitate FIR- and IIR-type filtering: the handler SHALL receive the issuing event – including the topic, <ctx> information, new input domain-object payload, lists of past input- and output-domain-objects (for the same topic/<ctx>), and return in response a new domain-object that in turn may trigger a notify event to the broker. Input- and output- domain-objects may not necessarily share the same definition/structure. Based on the initial proof-of-concept prototypes and performance evaluation tests available at [2], the event-source implementation is suitably RECOMMENDED to be based on the 'LMAX disruptor' pattern or one of its equivalent existing open-source C++ ports (alternate).

Internal to the micro-service, the Majordomo broker MAY notify and publish data only for active subscription topics, i.e. properties for which either external clients have explicitly registered their interest[4] or for properties where the user has specified this explicitly regardless of registered clients. The latter possibility facilitates that reconnecting subscription clients or services may retrieve the last notified data samples in the pre-defined default wire-format to build up their own internal history. The micro-service and Majordomo broker in particular SHALL thus keep an event store (as indicated in Figure 4) that implements the history of all internally notified, forwarded and

3 The RADIO/DISH pattern is used for transmitting small low-latency domain-objects via UDP, for example, timing telegrams, to trigger event-based aggregation routines described further below.

4 N.B. ZeroMQ's XPUB socket allows the monitoring of (un-)subscribe actions for a given topic.

serialised domain-objects (typ. 10-30 minutes, based on the available memory and domain-object size). The history SHALL be user-configurable and provide limits for the maximum number of events (domain-objects), time, or memory used by the domain objects. The memory of older domain-objects MAY be deallocated and returned to a byte-buffer-type pool.

Due to the various possible filter combinations, the matching between internal notified topics and registered subscription-client topics for which the domain-objects shall be serialised and forwarded MAY become a numerical bottleneck if there are a large number of subscribers and different topics. For the time being, this is not expected to be critical (yet), since the targeted number of subscribers per micro-service is in the few tens up to low-hundreds for which the resulting latencies using a simplified topic-by-topic matching should be small (compared to other industrial applications with tens of thousands of subscribers and topics). However, the delay for matching notified topics to subscriber-topics SHOULD be monitored. If this delays exceeds more than a millisecond on a modern hardware platform, the matching of notified topic to subscribed topic MUST be optimised. The inclined reader is referred to the OPTIONAL pertinent references on this topic: 'ØMQ High-speed message matching' and 'BRAVE NEW GEEK blog: Fast Topic Matching'.

The modification of settings via 'SET' and 'PARTIAL-SET' MUST be protected for given properties by a simplified Role-Based-Access-Control (RBAC) which implements a basic public-private-key authentication and control consistency validation mechanism (i.e. data content is not encrypted). In this scheme, the client SHALL compute and sign the hash-code of the to-be-sent serialised domain-object together with the user-role using its private key, and append the signed hash-code and role in an additional frame after the serialised domain-object. The micro-service receives the serialised data structure and – depending on whether the property is being protected – recomputes the hash of the sent serialised domain-object and verifies the result with the client's signed hash-code using the micro-service's public key. An exception SHALL be thrown in case of a mismatch.

The authentication and retrieval of the user-to-role mapping and public-private-key pairs is outside the scope of this specification and MAY – for practical purposes – be simplified and assumed to be provided through static configuration files that are available to the client and server. A schematic tentative protocol stack may look like:

- Frame 0: client or worker source ID (ZeroMQ-specific)
- Frame 1: protocol version (six bytes, e.g. "MDPW<xy>" or "MDPC<xy>" see 18/MDP)
- Frame 2: command (one byte, e.g. 0x01 GET request, 0x02: PARTIAL, 0x03: FINAL, ...)
- Frame 3: service or client source ID
- Frame 4: request or reply topic (i.e. topic "<device>/<property>?ctx=<ctx>;<filter$_i$=value$_j$>")
- Frame ≥5: request or reply (in YaS binary, CmwLight binary, or JSON wire-formats)
- Last Frame: RBAC role, token, signed hash (optional frame)

3.4.1 Some notes on Worker Modelling

The 'worker' SHALL encapsulate the user-provided code that is registered with and called by the Majordomo broker in case the registered topic[5] and other constraints are met. For each worker there MUST be at least one registered domain-object definition (i.e. C++ class or struct) that is used as part of the input-parameters and return-value of the worker's call-back functions.

5 which comprises the '<device>/<property>', <ctx>, and filters as described above

Multiple single-threaded worker implementations may (according to MDP) be registered to the Majordomo broker to allow for a certain degree of parallelism and re-using ZeroMQs internal message queues However, the default worker implementation SHALL primarily reuse one of the global Majordomo's or external process' thread-pools that can be shared across multiple workers and SHALL implement RBAC-based priority queues in order to control the trade-off between sequential execution (i.e. queues) and parallelism (i.e. multiple threads). There SHALL be several groups of thread-pools per Majordomo process and the user may chose which group the given worker belongs and should be executed within, for example, sharing a thread-pool for a given group of topics or based on criticality of the given topic (e.g. admin or security properties), An overview of the proposed worker threading schematic is shown in Figure 5.

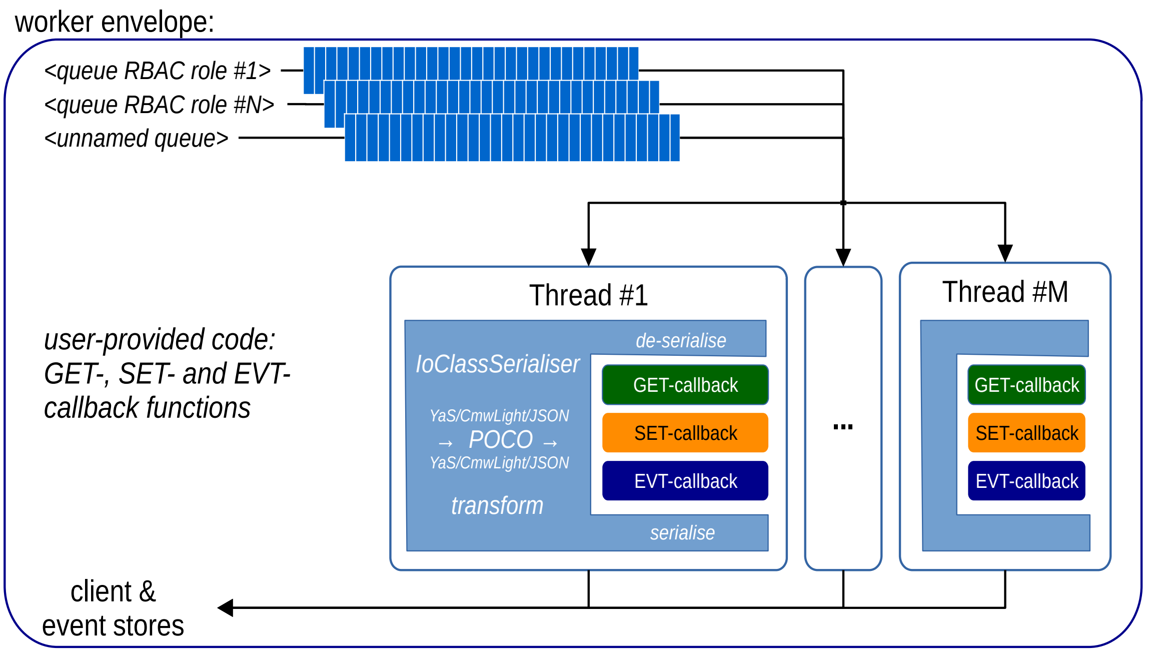


Figure 5: tentative worker threading schematic: each worker is assigned to a given thread-pool group during start-up that defines 'M' available threads and 'N+1' priority queues that are filled and served based on the provided RBAC role and its priority – if any – or otherwise default to an 'unnamed queue'. Each worker's user-code input or output is (internally) wrapped by the 'IoClassSerialiser' that detects the request or reply payload's wire-format and transforms the binary stream to and from a C++ domain object (POCO). The command frame determines whether the GET-, SET-, or EVT-callback function shall executed.

The user usually implements only the worker interface that groups the mandatory GET, SET and internal EVT-callback functions. For most cases, the worker should be derived (overwritable) from an abstract default parent class that implements the default GET/SET behaviour for for simple properties (i.e. where these are fully determined by the input-/output domain object).

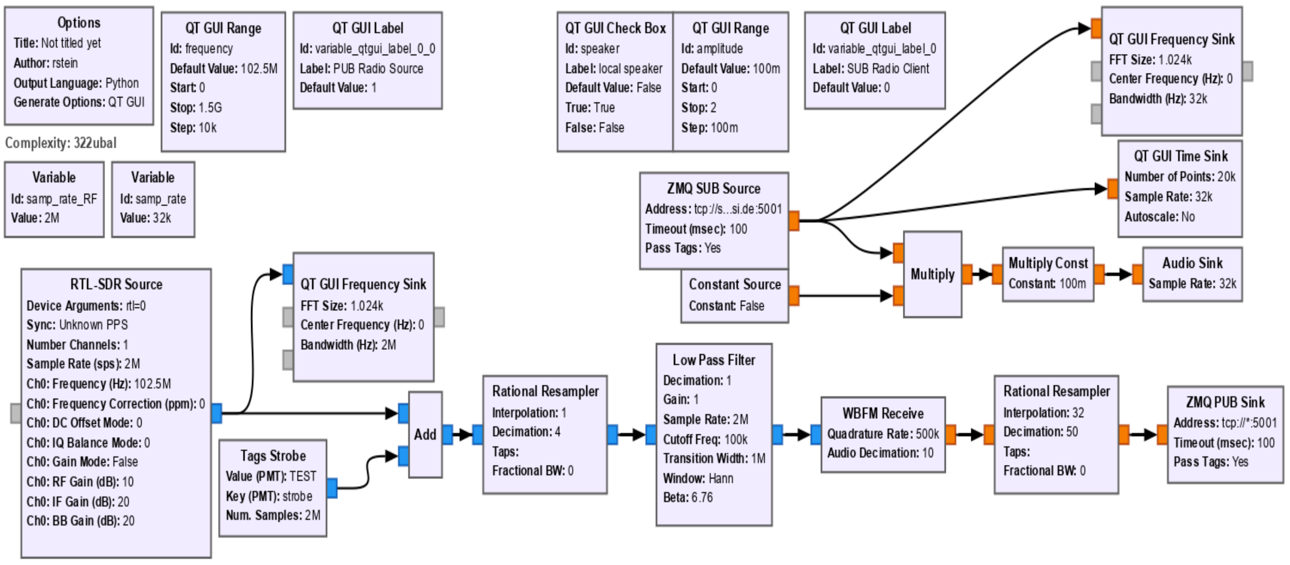
Several workers may be cascaded via the event store. For example, the return value of an internal call-back function triggering ('notify') the execution of another (pre-defined) worker implementation with the output of the first worker being the input of the second worker. The user may choose to register the internal call-back function only for the event-cascade but not necessarily as dedicated property. This is useful in case the intermediate worker's domain-objects are kept internally inside one of the event stores for post-processing in other workers, while there is no need to expose them as public topic. This way, the external client-side property interface can be simplified.

Nevertheless, domain-objects SHOULD be accessible through the generic event store property interface for expert-level diagnostics.

3.5 Aggregation of Continuous Data

The micro-service SHALL aggregate and tag continuous time updates of sample-by-sample or chunked data that are not yet aligned with a timing context <ctx> and provide multiplexed domain-objects for other workers within the same micro-service or external clients. Since this requires libraries with substantial external dependencies, this SHALL be implemented as an extension to the micro-service core functionality (i.e. plugin) and the users MUST be able to opt out of these functionalities. The specific implementation SHALL be compatible with the FAIR Digitizers Data Acquisition framework and low-level interfaces specified and described in [3].

For the aggregation and post-processing of continuous data, the micro-service SHALL reuse open-source signal processing, data analysis and fitting libraries, notably the GnuRadio (GR) and ROOT libraries [4, 5]. GR facilitates the implementation of software-defined-radio and provides powerful and versatile flow-graph-based signal post-processing facilities that can be configured programmatically, via 'GR's '*.grc' flow-graph files, or using the graphical user-interface (generating '*.grc' files) as illustrated in Figure 6.



Options
Title: Not titled yet
Author: rstein
Output Language: Python
Generate Options: QT GUI

Complexity: 322ubal

QT GUI Range
Id: frequency
Default Value: 102.5M
Start: 0
Stop: 1.5G
Step: 10k

QT GUI Label
Id: variable_qtgui_label_0_0
Label: PUB Radio Source
Default Value: 1

QT GUI Check Box
Id: speaker
Label: local speaker
Default Value: False
True: True
False: False

QT GUI Range
Id: amplitude
Default Value: 100m
Start: 0
Stop: 2
Step: 100m

QT GUI Label
Id: variable_qtgui_label_0
Label: SUB Radio Client
Default Value: 0

QT GUI Frequency Sink
FFT Size: 1.024k
Center Frequency (Hz): 0
Bandwidth (Hz): 32k

Variable
Id: samp_rate_RF
Value: 2M

Variable
Id: samp_rate
Value: 32k

ZMQ SUB Source
Address: tcp://s...si.de:5001
Timeout (msec): 100
Pass Tags: Yes

Constant Source
Constant: False

QT GUI Time Sink
Number of Points: 20k
Sample Rate: 32k
Autoscale: No

Multiply

Multiply Const
Constant: 100m

Audio Sink
Sample Rate: 32k

RTL-SDR Source
Device Arguments: rtl=0
Sync: Unknown PPS
Number Channels: 1
Sample Rate (sps): 2M
Ch0: Frequency (Hz): 102.5M
Ch0: Frequency Correction (ppm): 0
Ch0: DC Offset Mode: 0
Ch0: IQ Balance Mode: 0
Ch0: Gain Mode: False
Ch0: RF Gain (dB): 10
Ch0: IF Gain (dB): 20
Ch0: BB Gain (dB): 20

QT GUI Frequency Sink
FFT Size: 1.024k
Center Frequency (Hz): 0
Bandwidth (Hz): 2M

Tags Strobe
Value (PMT): TEST
Key (PMT): strobe
Num. Samples: 2M

Add

Rational Resampler
Interpolation: 1
Decimation: 4
Taps:
Fractional BW: 0

Low Pass Filter
Decimation: 1
Gain: 1
Sample Rate: 2M
Cutoff Freq: 100k
Transition Width: 1M
Window: Hann
Beta: 6.76

WBFM Receive
Quadrature Rate: 500k
Audio Decimation: 10

Rational Resampler
Interpolation: 32
Decimation: 50
Taps:
Fractional BW: 0

ZMQ PUB Sink
Address: tcp://*:5001
Timeout (msec): 100
Pass Tags: Yes

Figure 6: GR-based FM radio-receiver and PUB/SUB flow-graph example: the flow-graph contains both the ZeroMQ-based producer (bottom, ending with PUB- Sink) as well as consumer (top-right). Streaming tag (<ctx>) information is mocked by the 'Tags Strobe' block. source: 🗋

In the given example, isosynchronous 'stream tags' are used to tag and propagate the specific timing event and <ctx> information that is applicable during the data acquisition alongside each GR data stream.

Each stream is written via a custom GR sink into lock-free circular buffers that pertains the timing event tag information and that SHALL subsequently be used by the micro-service worker to construct the client-requested domain-object for each acquisition mode as outlined in Section 4.1 in reference [3] and illustrated in Figure 7.

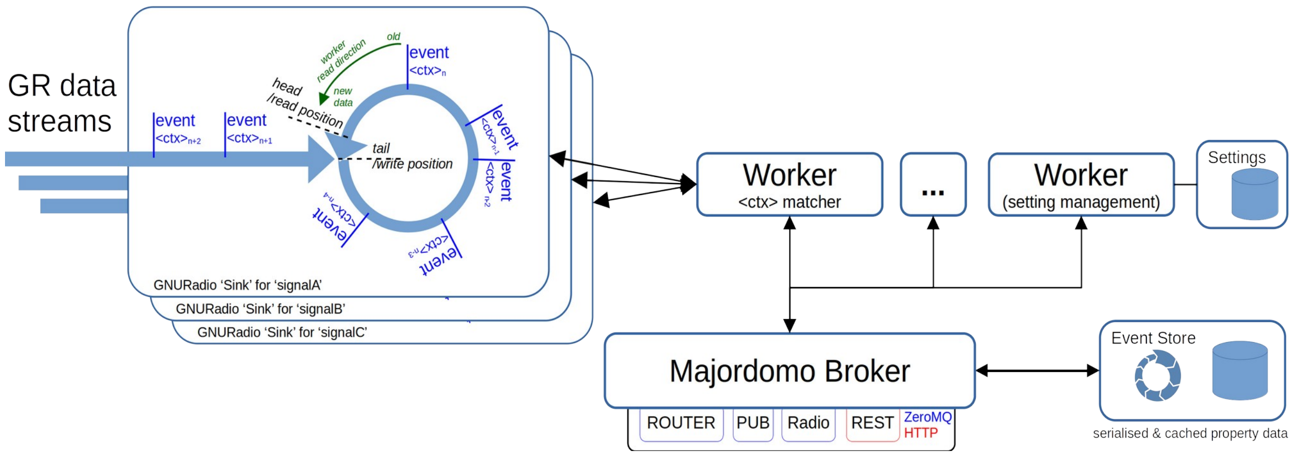


Figure 7: **schematic GR Sink as used by the FAIR digitizer.**

The micro-service SHALL provide facilities to integrate, modify the parameter of individual processing blocks, and allow remote updates to the flow-graph definitions themselves similar to as implemented for the FAIR digitizer framework. For the implementation of the micro-services, the timing multiplexing and stream tags may be simulated and based, for example, on the timing diagram schematically outlined in Figure 8. A GR source SHALL be provided as a client interface to acquire continuous data from other similar micro-service implementations. As an OPTIONAL extension, the acquisition MAY be extended to allow aggregation and synchronisation from multiple continuous external sources.

Rationale: Graphical flow-graph processing definitions provide a low-entry-point for users with little to no programming experience, allowing them to define custom signal-post-processing chains easily. At the same time, they profit from the integration of the digitizer and micro-service middle-wares with other software infrastructure of the control system (e.g. data archiving and post-mortem system, generic data visualisation tools, etc.). For more C++-affine users, GR provides the convenience of reusing common highly-performance-optimised signal-processing blocks (e.g. using VOLK) while focusing either on the high-level data analysis, extension of existing, or new custom C++ processing blocks.

3.6 Aggregation of Timing-Context-Multiplexed Chunked Data

The micro-service SHALL aggregate chunked data from different external sources that are already (partially) aligned to a timing contexts <ctx>, that need to be combined across multiple input sources, or that need to be combined into domain-objects covering a larger context (e.g. several pieces of BP-multiplexed data into Sequence- or BPC-multiplexed data) or split into smaller sub-contexts (e.g. BPC-organised data into its BP-constituents). The micro-service SHALL implement and follow an event sourcing pattern as schematically illustrated in Figure 9.

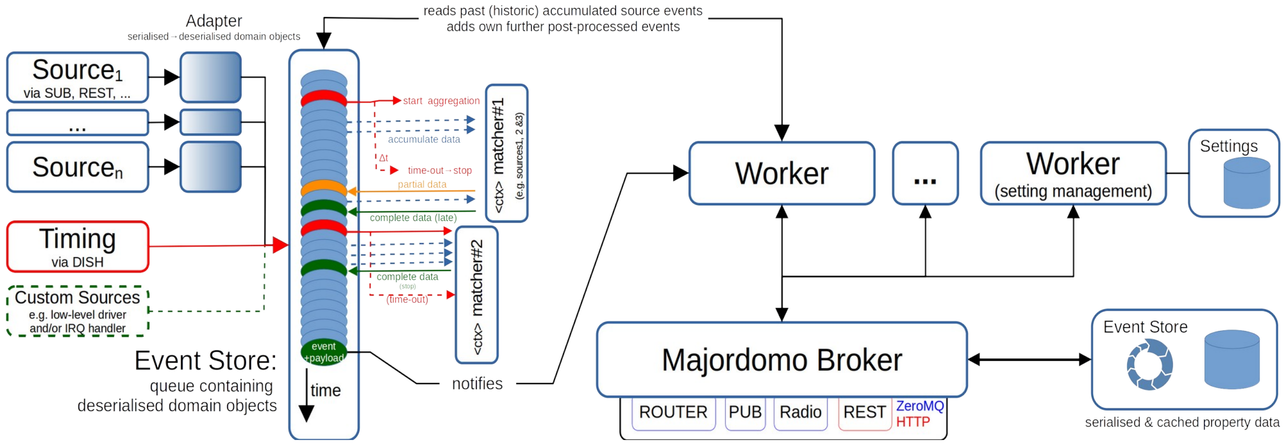


Figure 9: **schematic <ctx> multiplexed-based data aggregation:** based on the event GR Sink as used by the FAIR digitizer.

The aggregation scheme relies on three types of workers:

- the 'Adapter' tasked with receiving and de-serialising the specific (possibly foreign) wire-format from the external source, and storing the resulting domain-objects alongside their timing context into the primary event store's event stream[6]. Users MUST be able to define custom sources which MAY use the same interface than the predefined input sources. E.g. required for low-level drivers (e.g. IRQ handlers) that wish to write (part of) their data into one of the event store streams.

- the '<ctx>-matcher' that aggregates multiple source domain-objects for one given <ctx> and BPCTS into one combined aggregate domain-object. The aggregation is triggered by either a timing event or data arrival with a new <ctx> and/or BPCTS, accumulates the source specific domain-objects into a new aggregate. The new aggregate is stored inside the same (or optionally another) event stream within the same event store, either once all required data arrived for a given <ctx> and BPCTS or once a configurable time-out w.r.t. the aggregation start is reached.

- post-processing workers that perform the actual post-processing based on user-supplied worker handler-callback code. The handler-callback is triggered whenever a new aggregate is written to the event store. In order to facilitate FIR- and IIR-type filtering, the handler SHALL receive the issuing event – including the topic, <ctx> information, new input domain-object payload, lists of past input- and output-domain-objects (for the same topic/<ctx>), and – return a new domain-object that in turn may trigger a notify event to the broker. Input- and output- domain-objects may not necessarily share the same definition/structure.

6 The 'event stream' being one of possibly several memory- and time-limited queues within the event store that individual workers can iterate over. Each event stored in the stream should have a unique increasing event ID so that worker can detect whether new events arrived since the last time it checked the queue.

Default implementations for ‘GET’, ‘SET’, ‘PARTIAL-SET’ may be provided, but the post-processing handler-callback-function SHALL be kept as a mandatory method to be implemented by the user.

Additional worker services are required to monitor and debug the event store:

- a dedicated worker implementing the property that configures and controls the aggregation settings (e.g. time-outs, list of to-be-aggregated <ctx> sub-sets, load-balancing/number of ‘<ctx>-matcher’, etc.).
- a dedicated worker implementing the topic (property) that re-publishes the source- and aggregate events and payloads (domain-object, etc.) stored in the event store (e.g. via a topic ‘<micro-service>/InputEventStore/<signal source name>?<filters>#timing’).

The aggregation SHOULD be instrumented to monitor performance, detect and isolate failures.

4 Specific Functional and Implementation Requirements

The following section outlines specific technical requirements in addition to the conceptual requirements outlined in the previous section.

REQ-1. The design SHALL follow the concepts and technical details outlined in Section 3.

REQ-2. Deviations from the specification MAY be permitted, provided approval from the contracting body and the contractor (briefly) documents, for example:

a) the merit of the alternate implementation,

b) the potential design or performance improvements, or

c) the reprioritisation in order to achieve critical functionalities in earlier prototypes and when implementation of the other less critical functionalities are easy to implement at a later stage (ie. extreme programming paradigm).

REQ-3. Libraries SHALL be written in C++ following modern design paradigms.

REQ-4. Libraries MAY utilise C++ language features up to and including C++20 provided this aids minimisation of boiler-plate-type code, improves the readability or is pertinent for the performance. Reference boundary conditions (incl. max versions) are:

a) C++ compilers: gcc 10.2.1, clang 10.0.1

b) operating systems:

1. CentOS (8+Appstream and/or Stream),
2. OpenSUSE (LEAP 15.2 and/or Tumbleweed),
3. or other equivalent Linux distributions
4. OPTIONAL: MS Windows

c) hardware platform:

1. Linux x86 64-bit (Kernel 5.x)
2. OPTIONAL: Raspberry Pi (version ≥ 3) and other platform for which common gcc or clang cross-compilers exist.

REQ-5. Newer or experimental compiler features MAY be considered provided both parties agree and only if it is foreseeable that these will become part of the upcoming C++23 standard, are likely to receive a wider adoption and are available for at least one of the reference compiler and platforms.

REQ-6. The development SHALL follow modern CI/CD principles notably the rapid prototype and extreme programming pattern[7]. The source-code SHALL

a) be published in a public repository under the organisation's account[8],

7 I.e. testing critical items and their full vertical stack implementing first before adding further functionalities.

8 source code shall be hosted either at GitHub under the organisational account for generic library aspects, or at the organisation's own public GIT infrastructure in case there are site-specific configurations or implementations necessary (e.g. configuration scripts for local test environments, etc.). The need for the secondary should be minimised.

b) be published under the LGPL 3.0 license with the organisation being the primary custodian and copyright owner,

c) development snapshots MUST be buildable and unit-testable on at least one of the publicly available CI/CD platforms (in order of preference): TravisCI, Github, CircleCI, and the openSUSE Build Service

d) builds in the 'master' branch MUST pass the following minimum QA test requirements:

1. build structure, compiler warnings and best practises listed in the GitHub hosted cpp_starter_project reference projects,
2. unit-test code-coverage of at least 75% (N.B. the publicly available Codecov.io service may be used to automate the metric),
3. LGTM static code analysis, executed for each pull-request from a GIT development to master branch,
4. Coverity Scan static code analysis, regularly executed within less than a week.
5. further additional analysis tools[9] MAY be added if deemed useful.

REQ-7. The final site-acceptance test SHALL be able to generate stand-alone container (pods) that contain the executable micro-service's or test application's binary as well as all dynamic library components that are required for the execution.

REQ-8. The CI/CD build systems SHOULD use

a) Modern CMake (3.18.0)

b) Conan

c)

REQ-9. The implementations SHOULD minimise external library dependencies beyond those that are already part of the C++ language standard. Additionally required external dependencies SHALL be agreed upon by both parties. Intrinsically necessary libraries that are known during the specification phase and that may be used are:

a) {fmt} (v.7.0.3) globally,

b) LMAX disruptor globally,

c) ZeroMQ (v4.3.2) for the micro-service and (de-)serialiser libraries,

d) log4cpp (> v1.1.3) for the micro-service and (de-)serialiser libraries,

e) GNURadio (v3.8) for the specific continuous-time micro-service implementation,

f) ROOT (v6.20/06) for the event-builder micro-service implementation example, or

g) Qt (5.15, OPTIONAL 6.0, LGPLv3-compatible components only) for the scientific charting library and its test applications.

9 e.g. cppcheck, clang-tidy, Valgrind

4.1 Micro-Service Middle-Ware Library

MS-1. The overall design SHALL follow the concepts outlined in Section 3.

MS-2. The micro-service SHOULD follow modern standards and design principles, and favour a light-weight unopinionated library character that can be “mixed-and-matched” – rather than a monolithic opinionated feature-complete framework that locks or guides users into the framework’s way of doing things.

MS-3. SHOULD implement a modular system that provides basic functions with minimal external library dependencies that can be further extended by plugins that may depend on additional library dependencies.

MS-4. SHOULD favour ‘composition’ over ‘inheritance’ in order to be easily integrated or re-used by other user-applications.

MS-5. SHALL run in a single process, use internal threads as needed, and spawn external micro-service processes where reliability or user-requirements demands them[10]. The threads’ scheduling priority and CPU affinity SHALL be statically configurable.

MS-6. SHALL provide configurable thread pools for groups of workers and clients that MAY grow or shrink dynamically, for example, based on the average micro-service load.

MS-7. SHOULD encapsulate spawning of external workers to be extendible.

MS-8. SHALL provide a setting store that contains domain-objects that are directly accessible from any internal worker and externally accessible through the Majordomo broker and its standard communication library API.

MS-9. SHALL recover its initial setting store configuration during startup (in increasing priority order) from either:

1. the default class constructors of the corresponding domain-objects,
2. deserialised from local configuration files, using one of the known wire-formats [2.b)], or
3. from another micro-service that specifically stores the initial specific micro-service device configuration or retrieves this from, for example, a to-be-defined local GIT-based versioning system or database.

MS-10. SHOULD document the user-side API of the Majordomo broker properties, domain-object and properties according to the Majordomo management interface 8/MMI, accessible via a default property, and OPTIONALLY also using the OpenAPI definitions.

MS-11. SHOULD instrument and provide a diagnostic property that exports statistics on input- and output connections, for example: list of active subscriptions and input/output bandwidths for a given topic, missing required input subscriptions (streams), statuses of circuit-breakers, heart-beat analysis, access frequency of properties (hourly rate), process utilisation etc.

MS-12. SHOULD export statistic data to an external logging framework (e.g. Prometheus)

MS-13. The micro-service SHALL be able to export for each property basic HTML-based views and that allow simple form-based data interaction with the property data via its REST interface[11]. The following functionality SHOULD be provided with increasing level of complexity and that may require additional external libraries:

10 e.g. ‘flaky’ user-code or non-C++ worker implementations that shall not crash the more robust broker

1. default table-based property views including form-based data interactions using a basic HTML template engine,
2. extended views and interactions based on user-supplied custom HTML templates,
3. as above but complemented with graphics content (e.g. rendering data using the visualisation library and embedding the resulting graphics into the template), and
4. optionally export user-provided Web-Assembly WASM-code of custom applications.

4.2 Communication Library

CL-1. The Majordomo MDP/0.2 protocol pattern SHALL be implemented, extended by specific functionalities outlined below, and that can be integrated into other C++ programs [6].

CL-2. SHALL be compatible with existing wire-formats and domain objects [2].

CL-3. SHOULD provide open interfaces for new wire-formats or transport protocols.

CL-4. SHALL derive the (de-)serialisation scheme directly from the user-defined domain-object data definitions (i.e. annotated C++ classes or structures) that are exchanged between micro-services and clients. The implementation SHOULD evaluate the use of compile-time (static) reflection, in particular existing prototype concepts or those that are expected to be become part of the official C++23 standard [7]. The expressed aim of the compile-time reflection SHOULD be to minimise the necessity of external, proprietary and error-prone IDL-definitions or command-line-based two-phase-parser-code-generator constructs. Ideally, the class serialiser is a header-only library that directly generates the required information for the (de-)serialisation of the domain-object.

CL-5. SHALL forward exceptions in worker or broker to the respective client or event sources.

CL-6. The Majordomo broker SHALL abstract the underlying transport (i.e. requests via ZeroMQ vs. REST) and wire-format protocols and keep them transparent from the worker's post-processing call-back function. i.e. the worker's result is received, processed and returned to each client in the client's wire-format of choice without the worker needing to know the specific details.

CL-7. The low-level interface SHALL be kept open and worker call-back functions MAY be given (albeit deprecated/expert) access to low-level library functions to add custom code or provide workarounds for missing functionalities in the underlying transport protocol layers.

CL-8. The Majordomo broker SHALL keep a history of domain-objects and corresponding serialised property data in one or more event stores with configurable depth (i.e. limited number of events, time, or available memory).

CL-9. The Majordomo broker MAY keep a configurable history of all client GET/SET requests for debugging purposes only. Access to this information SHALL be limited, enabled or disabled (default) using RBAC.

CL-10. The PUB/SUB equivalent push implementation for the REST interface to clients SHALL support both 'server-sent-events' (SSE) as well as the 'long-polling'-based technique[12].

11 N.B. Since REST via HTTP allows multiple wire-formats, the choice between HTML, JSON, or binary-based formats shall be determined primarily by the client's mime-type or overwritten by a provided filter (e.g. '?mime-type=JSON') in the requested topic.

CL-11. The Majordomo broker SHALL provide extendible functions to register each of its bound ROUTER, PUB, RADIO, and REST ports to a given nameserver. For the time being, the type of nameserver and required specific protocol to register these ports remains to be defined (out-of-scope for this implementation).

CL-12. The library SHOULD be primarily split into:

1. a server-side component (e.g. Majordomo broker, worker handling, basic property implementations),
2. a worker-side component for external C++-based[13] worker (user-code), and
3. multiple client-side components that can be used on their own, for example, one each for dealing with the DEALER/ROUTER, PUB/SUB, RADIO/DISH, and REST interfaces.

CL-13. The micro-service library SHALL provide utilities and helper functions that implement a circuit-breaker, retry and fall-back pattern to be used in worker or client tasks that access external resources and that may (temporarily) be unavailable. As a RECOMMENDED conceptual reference, the popular (albeit Java-based) Resilience4J implementation may be consulted and used to derive an optimised C++-based version. This may be OPTIONALLY extended by rate-limiter, time-limiter, or cache that share similar implementations.

CL-14. The micro-service SHALL deploy (RBAC) role-based priority queues for client requests.

4.3 Scientific Charting Library

SC-1. SHALL provide a Qt-based functional equivalent 'Chart-Qt' implementation that is compatible to the Chart-Fx Java-based library [2].

SC-2. SHALL be written in C++ and MAY utilise language features up to C++20 if this aids minimisation of boiler-plate-type code, improves readability, or is pertinent for the performance, standards follow modern design principles.

SC-3. SHOULD follow a MVC-pattern similar to Chart-Fx and – where possible – re-use similar API and domain-object interface definitions.

SC-4. Interfaces MAY be adapted using templates and other C++ design patterns to aid performance, usability, and if this provides a cleaner more maintainable C++ code.

SC-5. MUST achieve a performance equivalent or better than its Java-based counter-part for equivalent benchmark cases.

SC-6. SHALL provide a similar plugin, renderer infrastructure and extendable axis and DataSet domain-object functionalities. The minimal system SHOULD provide implementations for the following specific components (N.B. each category sorted according to priority/simplicity):

1. 'XYChart' and its 'Chart' parent class,
2. 'Axis'-interface: 'DefaultNumericAxis', 'OscilloscopeAxis', and axes' synchronisation,
3. 'ChartPlugin's: 'Zoomer', 'EditAxis', range and value indicators[14],

12 N.B. the 'long-polling' facilitates easier transparent caching proxies across network domains. Proxy-implementation is out-of-scope for this specification.

13 N.B. implementations for non-C++ implementations, for example, Java- and Python-based are foreseen but out-of-scope for this specification.

14 may benefit from a code de-deduplication.

4. ‘Renderer’: ‘ReducingLineRenderer’, ‘ErrorDataSetRenderer’ (incl. reducer[9]), and ‘ContourChartRenderer’,
5. ‘DataSet’-interface: ‘[Double,Float][Error]DataSet’[10,15] and ‘MathDataSet’,
6. demos equivalent to: ‘SimpleChartSample‘, ‘RollingBufferSample‘ (as a quick performance benchmark), ‘ErrorDataSetRendererSample‘, ‘ErrorDataSetRendererStylingSample‘, ‘ContourChartSample‘, and ‘ChartPerformanceBenchmark‘.

SC-7. SHALL be able to render charts directly into PNGs and OPTIONALLY SVGs objects.

SC-8. SHOULD explore the compatibility to be transpilable to Web-Assembly (WASM).

5 Quality Assurance, Tests, and Acceptance

The following subsections briefly describe the necessary quality assurance (QA) measures in addition to the specific technical QA requirements listed above.

5.1 Quality Assurance system of the contractor

Due to the complexity of the accelerator control system and therefore the need for early integration tests, an incremental and iterative development approach is advised. Based on the main requirements an overall solid system architecture is setup and in a first iteration a vertical slice through all layers is built realising the most crucial functionalities. Then, in well-defined iteration cycles, components are replaced by newer versions, implementing more functionality.

Additionally the long lifetime of the FAIR facility and in this time the occurring computer hardware and software technology changes generate the necessity for continuous adaptation and extension. All delivered components must therefore be developed with the focus on maintainability and exchangeability.

The code and project documentation should follow best-practices for free and open-source projects and include a basic primer and examples for new and/or less skilled C++ developer.

5.2 Factory and Site Acceptance Test (FAT & SAT)

The Factory Acceptance Test (FAT) is used to ensure that the component meets the functional and non-functional requirements defined by the contracting body and is ready to be delivered. The Site Acceptance Test (SAT) ensures that the component fulfils all user requirements and that it is ready to use within the users environment. Each iteration cycle must repeat tests from previous iterations. The development SHALL follow a Continuous-Integration/Continuous-Development (CI/CD) paradigm with the FAT and SAT being continuously evaluated on the public build server. After the last iteration step and successful testing of all requirements a review meeting is to be done to get the acceptance by the contracting body.

Due to the software nature of this specification, only the SAT is used as criterion of fulfilment of the specification. However, it is highly recommended that the implementer reuses the same SAT procedure (and associated module Unit testing) for their internal FAT prior to delivering or deploying the component.

In order to simplify testing, the delivered component shall primarily implement:

- the unit tests associated with each individual post-processing modules,

15 the use of templates is strongly encouraged

- the reference flow-graph, post-processing chains, (simulated) reference test signals outlined in Sections 3.5 and 3.6,
- web- and basic graphical C++-based user-interface which make use of all specified post-processing and data visualisation modules and which are representative for all other possible signal flow-graph and post-processing combinations.
- intermediate results and concepts SHALL be uploaded to the repository in mutually agreed upon intervals and/or based on amount of work (i.e. every 5-10 effective work days).

Exemplary module parameters and GNU Radio and reference implementation will be provided where applicable or necessary during the development process.

The primary SAT acceptance criteria are:

- fulfilment of the functionalities outlined in section 3 and of all named individual items in section 4 (i.e. ‘REQ-[1-9], MS-[1-14], CL-[1-14], and SC-[1-8]),
- adequate functional unit-test code line-coverage of at least 75% with respect to the library code-base,
- adequate inline code documentation and coding standards, and
- functionality and performance equivalent to the Java-based reference for the specific class implementations outlined above in the specific technical requirements.

6 References

1. RFC 2119 “Key words for use in RFCs to Indicate Requirement Levels”, https://tools.ietf.org/html/rfc2119
2. Chart-fx charting library: https://github.com/GSI-CS-CO/chart-fx
   a) domain-objects: https://github.com/GSI-CS-CO/chart-fx/tree/11.2.2/chartfx-dataset/src/main/java/de/gsi/dataset
   b) wire-format: https://github.com/GSI-CS-CO/chart-fx/tree/11.2.2/microservice
3. Common specification, “On the Digitization of Analog Signals in the FAIR Accelerator Complex” (F-CS-C-0002e), https://edms.cern.ch/document/1823376/1
   a) GNURadio Integration of the FAIR digitizer (3000-, 4000- and 6000-series PicoScopes from Pico-Technology): https://gitlab.com/al.schwinn/gr-digitizers
   b) GNURadio Integration of the FAIR digitizer flow-graph modules: https://gitlab.com/al.schwinn/gr-flowgraph
   c) A. Schwinn, “Lock-Free Circular Buffer Implementation” Prototype at: https://gitlab.com/al.schwinn/lockfree-custom-fesa-cyclebuffer
4. GNU-Radio, https://www.gnuradio.org/ further useful references:
   a) flow-graph concepts: https://wiki.gnuradio.org/index.php/Guided_Tutorial_GRC
   b) writing custom GNURadio C++ modules: https://wiki.gnuradio.org/index.php/Guided_Tutorial_GNU_Radio_in_C%2B%2B
5. ROOT Data Analysis Framework, https://root.cern/ & https://github.com/root-project/root
6. Majordomo Protocol MDP/0.2: https://rfc.zeromq.org/spec/18/
7. Known C++ compile-time reflections concepts:
   a) refl-cpp (C++17): https://github.com/veselink1/refl-cpp
   b) RTTR C++ Reflection Library (C++13): https://github.com/rttrorg/rttr
   c) http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0592r4.html
      1. David Sankel, “C++ Extensions for Reflection”, ISO/IEC CD TS 23619, N4856 http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/n4856.pdf