D3.3
Full Core Middleware release

Work package	WP3 Core Memory and Data Aware Middleware	
Author(s)	Christopher Haine Utz-Uwe Haus	HPE HPE
Reviewer #1	Domokos Sarmany	ECMWF
Reviewer #2	Jayesh Badwaik	JUELICH
Reviewer #3	Marc Perache	CEA
Dissemination Level	public	
Nature	other	

Date	Author	Comments	Version	Status
31.07.2020	Utz-Uwe Haus	Initial version for first review	V0.1	Draft
05.11.2020	Christopher Haine	Revised version for second review	V0.9	
30.11.2020	Christopher Haine	Revised version after second review	V1.0	Final

DATA ORCHESTRATION IN HIGH PERFORMANCE COMPUTING
This project has received funding from the European Union's Horizon 2020 research and innovation program through grant agreement 801101.

Executive Summary

Maestro is a FETHPC-2018 funded project that will design a data- and memory-aware middleware framework for HPC applications and workflows. Work Package 3 (WP3) of the Maestro project develops the core middleware framework.

Contents

Glossary

Maestro Core	...	The internal software layer that constitutes the Maestro framework
CDO	...	Core Data Object
API	...	Application Programming Interface
GPU	...	Graphics Processing Unit
MPI	...	Message Passing Interface
UPC	...	Unified Parallel C
GFS	...	Global File System
MIO	...	Maestro IO Interface
WIP	...	Work In Progress
PEG	...	Parsing Expression Grammar

1 Introduction

1.1 Architecture Reminder

The Maestro middleware is built around the concept that applications should be empowered to delegate the movement of and/or access to the data they provide or require to an intelligent middleware that can reason about the system characteristics, data-movement cost, and workflow-level scheduling of data placement. At the same time, data should not be required to be allocated in Maestro-defined data structures. Rather, low-overhead annotations of existing data should be sufficient to inform the middleware of and permit it to handle such application-defined and application-managed data.

The Maestro middleware can be understood to provide its features at three different levels:

- As a data-management layer for all memory tiers of a system inside a single process, e.g. across multiple threads, to use the abstraction of core data objects (CDOs) to better structure data exchanges between program parts, across devices (e.g. GPUs), and to use convenient data transformations provided by the core library.
- As a data-management layer for all memory tiers of a system across multiple execution domains (compute nodes, processes, workload-manager jobs and/or allocations), including coupling of applications by their CDO dependencies.
- As an enabler of a workflow-management solution, in that it enables the workflow manager to observe and influence data availability, demand, locality, and transfer without the applications knowing about the workflow they are embedded in.

In all three cases, the basic usage pattern is the one that is described in D3.1 [2].

- Data objects (CDOs) are declared using a workflow-level unique name.
- Attributes (Maestro-specific or user-specific) can be added to each CDO; existing data allocations are simply attributes as are layout information and access patterns. Intended usage, lifetime, or required permanence or redundancy are other attributes.
- Data objects are offered to other participants of a workflow or requested from other participants via a conceptual pool.
- Participants withdraw the object they contributed to the pool eventually.
- During the time an object is pooled, Maestro take full control over it; it may move, re-layout, redistribute, or copy the data as it sees fit across the entire range of resources available to the workflow, including the resources contributed by participants in the form of CDOs.

- A consistent memory access API is integrated with the CDO interface, allowing seamless access to various memory layers, and independent of whether the CDO is user-allocated or Maestro-allocated.

The Maestro design is intentionally flexible about implementation of the components, and we are striving to avoid monolithic design wherever possible. We believe that this makes it easy to exchange certain components later on, or extend functionality, while keeping compatibility not only at the API level, but also with the implementation currently built. At the same time, the design was guided by the principle of the Core being 'just a library', usable from C/C++/Fortran, and not dictating a programming model, such as MPI, UPC, etc.

2 Additional APIs

The Maestro core design consists of mainly two APIs as proposed in D3.1 [2]. One is the CDO management API which was largely discussed in D3.1 [2] and released in D3.2 [3], and the other is the CDO transformation API which will for the most part be the subject of the next deliverable (D3.4). In this section we describe additional APIs, as part of the full release, on the topic of metadata, events and groups. In particular we address classes of requirements from Maestro applications that have been made explicit by applications partners in D2.1 [6] and D2.3 [5], and sum them up in Appendix 6.2 indicating their current status.

2.1 Events

Event subscription allows for closer inspection and control on the maestro-enabled workflow operations. For example, it makes it possible to know if a certain CDO is OFFERed, or if a given application has joined the workflow. It is also possible to receive acknowledgment when a given operation was performed by an application, which would possibly arbitrarily stall said application in its pool operations, to finally let it proceed normally.

To define a set of CDOs that matches certain desired criteria, such as values or ranges of values for given attributes – which can be user-defined (see Section 2.2) –, including wildcards and other regular expressions, users need to create a CDO selector. The syntax for selector construction is given in Fig. 1.

The `mstro_subscribe` operation declares that a notification – of an event – is to be sent upon changes to a CDO whose attributes satisfy the conditions of the specified CDO selector. In particular, wildcards on the name or certain attributes make it possible to trigger event messages for CDOs whose name (or ID) is not known to the recipient previously. This can be used to implement custom handling of CDOs which are not known by (or have dynamically changing) names. In order to make it possible for the receiver of an event to immediately post a REQUIRE, that is to be able to have a chance to inspect the CDO before it is taken off the pool too quickly, the `mstro_subscribe` operation needs a flag (`require_ack` flag)

```
selector <- attribute-selector
         | union-selector
         | intersection-selector

attribute-selector <- attribute-key kv-op value
                    |  '(' 'has' attribute-key ')'
                    |  '(' attribute-selector ')'
                    |  '(' 'not' attribute-key ')'

kv-op <- '=' | '!='              /* type-based (non-)equality */
      | '<' | '<=' | '>' | '>='  /* type-based comparison
                                    (numerical types) */
      | '~='                     /* regex-match on string
                                    representation of value */

union-selector <- '(' 'or' selector-list ')'

intersection-selector <- '(' 'and' selector-list ')'

selector-list <- selector
              | selector ' ' selector-list
```

Figure 1: CDO selector syntax (pseudo-PEG format). The authoritative syntax is defined in `attributes/maestro/cdo_sel_parse.peg`.

to indicate that the CDO triggering an event should be forbidden from being completely withdrawn from the pool before an acknowledgment has been received from the notified party. In case the user requires an acknowledgment for completion of a given operation, `mstro_subscription_ack` needs to be called to allow the pool manager to proceed with the completion of the operation handling, thereby possibly releasing a stalled application. Example code is given in section 5.2 for such scenario, in particular in `check_subscribe.sh`. In many cases however, this flag will not be set, and events do not need to be acknowledged, and will not stall the operation progress.

Watching for events is done via the two traditional routes, that is either by a poll (`mstro_subscription_poll`) or a wait operation (`mstro_subscription_wait` or `mstro_subscription_timedwait`). Figure 2 shows the event subscription API.

2.2 Metadata

Metadata – otherwise referred to as “CDO attributes” in Maestro core language – is encapsulated by CDOs as previously discussed in D3.1 [2], and is of great relevance for applications as shown by the numerous related requirements (see Appendix 6.2 for status, and D2.3 [5] for detailed description). Core attributes – such as local size or name, for the full list see `./attributes/maestro-core.yaml` – are accessible by default through the `.maestro.core` namespace, via `mstro_attribute_get` and `mstro_attribute_set` functions; example code is given in Section 5.2, in particular under `check_declaration_seal`.

```
mstro_cdo_selector_create(mstro_schema schema,
                          const char *namespace,
                          const char *query,
                          mstro_cdo_selector *result);
mstro_subscribe(mstro_cdo_selector cdos,
                mstro_pool_event_kinds events,
                bool require_ack,
                mstro_subscription *res);
mstro_subscription_wait(mstro_subscription s,
                        mstro_pool_event *event);
mstro_subscription_timedwait(mstro_subscription s,
                             const struct timespec *deadline,
                             mstro_pool_event *event);
mstro_subscription_poll(mstro_subscription s,
                        mstro_pool_event *events);
mstro_subscription_ack(mstro_subscription s,
                       mstro_pool_event events);
```

Figure 2: Event subscription API

An extra API is needed to allow users to specify their own. In order to incorporate user-defined attributes within CDOs, the Maestro core expects the user to provide a YAML schema that user-defined attribute operations will have to be compliant with, essentially consisting a key-value list of possibly optional keys. To ensure attribute consistency, the aforementioned schema itself needs to be validated against a top-level internal schema (content given in Appendix B). The meta-schema serves as a reference for all attribute schema, including the core attributes schema. Validation process is detailed in `./attributes/attributes.md`. The meta-schema accepts native types such as string, int, bool, enum and map, the latter allowing composition.[1]

In the application code, the user needs to make a couple of calls to enable the user-defined schema in Maestro core. First, `mstro_schema_parse` must be called to load the schema. Then, once parsed, the user must call `mstro_schema_merge`) in order for the schema to be merged to the existing full attribute key-value list recognized by the Maestro core. After that, the user may specify attribute values dynamically by calling `mstro_attribute_get` and `mstro_attribute_set`, or else by way of a YAML string to `mstro_attributes_set_yaml`.

The user-defined attributes API is summarized in Figure 3.

See `./attributes/attributes.md` for complementary information.

2.3 Groups

Groups of CDOs, as first defined in D3.1 [2], is a way of conveniently handling multiple related CDOs. Groups have a few basic attributes such as a name, and the list of CDOs it consists of.

[1] Since not a lot of experience has been gathered with the requirements arising from user-defined schemata, the meta-schema may require extensions in future versions of Maestro.

```
mstro_schema_parse(const uint8_t *yaml_data, size_t data_len,
                   mstro_schema *result);
mstro_schema_parse_from_file(const char *fname, mstro_schema *result);
mstro_schema_merge(mstro_schema main,
                   mstro_schema consumed);
mstro_schema_free(mstro_schema sch);
```

Figure 3: User-defined attribute API.

There are a few different ways to assemble a Group:

- add a CDO to the Group using its handle;
- add a CDO to the Group using its name;
- add a set of CDOs using a selector.

A group can be constituted of both names and handles. This allows forward-referencing: the OFFERing component does not have to be offering all of its CDOs.

The Groups API is consistent with CDO management API in the sense that, similarly to CDOs themselves, Groups can be `DECLARE`d, `OFFER`ed, `DEMAND`ed, and so on. However, in the context of Groups the semantics are slightly different from those of CDO management.

`SEAL`

- Seals the CDOs that are not sealed yet
- Resolution of CDO selectors occurs at this point

`OFFER`

- Group definition at a global scope, that is for all workflow participants. At this point producers can't contribute any more CDOs to the Group: the group member list is final.
- Offers the CDO that are not offered yet

`DEMAND`

- Results in the user obtaining a group handle, with a list of CDOs containing minimal information, and access to the CDO data itself.
- Will block until the pool manager can guarantee availability of all member CDOs
- The group handle returned by REQUIRE/DEMAND is provided with an iterator, GROUP_NEXT, that performs the individual CDO DEMAND operations in a Maestro-defined order

`WITHDRAW`

- Similar to CDOs will block until all recipients have disposed of the group

- CDOs that are in an OFFERed group cannot be withdrawn until the groups that contain them have been withdrawn

`DISPOSE`

- Does not dispose the constituting individual CDOs, users must dispose of CDOs individually even if they did not declare them explicitly (e.g., because they received them after a name-only group membership declaration, or received a handle by GROUP_NEXT iteration).

A test implementing these semantics is available in the repository in `./tests/simple_group_client.c` and can serve as an example.

Behaviour described above might be too rigid for certain use cases, typically if the user does not want to wait until all Group members are available at DEMAND. Using the Events API instead is the way to express such semantics, where a conceptual group – *i.e.* a set of CDOs, but not a CDO group in the maestro core sense – is formed by a CDO selector, and where individual availability can be probed using events.

3 Documentation

The code is documented using `doxygen`, and automatic documentation generation occurs at build time. The entry to the html-based documentation is at `docs/html/index.html`, while rudimentary manual-page style documentation can be found at `docs/man/man3`.

4 Building

The Maestro core repository is available here: `https://gitlab.version.fz-juelich.de/maestro/maestro-core`

The Maestro core repository uses `gitlab.com` continuous integration, allowing testing on Docker containers with a few linux systems such as Debian, Ubuntu, openSUSE and CentOS.

Steps to build the code have been previously described in [3] and are also available in the README and INSTALL files.

5 Testing

This document describes the code of the upcoming Version 0.2.0 release of the Maestro core. It has been tagged as `d3.3` after review of this document, with patches included from the `d3.3-review` revision that was basis of the initial review of this document.

The Version 0.2.0 release is expected shortly after completion of this document, subject to updates of the included `mamba` and `mio` libraries.

5.1 Demonstrators

Maestro core repository features three out of its four planned MVPs, as defined in D6.1 [4] for WP6, with the fourth one being the subject of a future deliverable. MVP1 is a multithreaded local demonstrator, uses a proxy for the workflow component, producers, consumers and archivers. MVP2&3 are a multiprocess demonstrator, where two clients exchange a CDO through the Pool Manager, using GFS and MIO transport. MVPs are located in the `./tests` folder.

Name	Run	Further reading
MVP1	run_demo.sh	See [3]
MVP2&3	check_pm_interlock.sh	See by ascending amount of detail D6.1 [4] on MVP definition, the `INSTALL.md` file, and D5.5 on Adaptive Transport [1].

5.2 Examples

Maestro core repository also features a number of unit tests (in `./tests`), which altogether account for nearly 50% of code coverage. Some of these tests can serve as reasonable example code for Maestro users, as detailed below.

Name	Purpose
check_version	get version string
	Basic API
check_init	call toplevel API init/finalize
check_declare	local declare
check_declaration_seal	attribute set/get, regular and YAML, and SEAL example
check_pool_mamba	basic Mamba use through Maestro
check_pool_retract	basic retract example
	Local Pool
check_pool_local	local test using basic CDO movement API
check_pool_local_rawptr	basic raw pointer attachment to a CDO and local movement
check_pool_local_stress	same with many calls
check_pool_local_putget	local multiple producer/consumers OFFER/DEMAND
	Metadata
check_schema_parse	user attributes (metadata) schema parse example
check_type_parser	check schema type parsing
	Transformation
check_layout	automatic rowmajor to colmajor transformation; automatic buffer allocation
	Multiprocess
check_pool_manager	pool manager startup
check_pm_declare.sh	basic one pool manager plus two clients setup
check_pm_declare_group.sh	one pool manager plus two clients exercising the CDO GROUP interface
check_subscribe.sh	archiver component subscribes to CDO OFFERs from an injector component

The multiprocess tests contain a `simple_pool_manager` component that can be used directly as a default pool manager component in a multi-application workflow.

6 Requirements Status

6.1 Core

Requirements for the Maestro core, as per D3.1 [2]:

Proposal Number	Proposal Text	Status

PP.3.1.1.a	Maestro should introduce a solution to general ubiquitous problems of data-movement, without being constrained by the limited scope of existing solutions.	✓
PP.3.1.1.b	Maestro should generate new data- and memory-aware abstractions for data handling, management, movement and transformation.	✓
PP.3.1.1.c	Maestro should develop a middleware layer, upon which other software components spanning systems software and programming environments (amongst others) can be developed.	✓
PP.3.1.1.d	Maestro should enable runtime analysis of data objects.	→ WP4
PP.3.1.1.e	Maestro should enable compile-time analysis of data-objects, meaning (together with PP.3.1.1.d) that Maestro should combine static and runtime analyses.	→ WP4
PP.3.1.1.f	In addition to an explicit interface, Maestro should present an implicit, "temporary behavioural modification" model, providing e.g. POSIX library overloading during/under certain timeframes/conditions. When Maestro is disabled, the application proceeds as it would before Maestro modifications, and with no performance loss.	✗
PP.3.1.1.{g,h}	Maestro core abstractions should be able to detect the specific case of layout (in)sensitive transformations, to adapt to this situation and to store sufficient scope to inform decision making in this area.	detection→ WP4, storage→ metadata, transformations→ D3.4
PP.3.1.1.i	Although Maestro core abstractions should adapt to the situations of functionality described in PP.3.1.1.{g,h}, it should also adapt to the situations where qualities of both classes are relevant.	→ WP4
PP.3.1.2.a	Maestro should present an API that is useful at any layer of the software stack, including at the very least various components in the programming environments, systems software and runtimes spaces.	✓
PP.3.1.2.b	Maestro should be aware of the nuanced variety of system-specific performance characteristics and take these into consideration when performing data accesses and movements.	→ WP4

PP.3.1.2.c	Maestro should recognise the variety of means by which data can be stored in a certain media. However for internal transformation purposes it will choose the most appropriate or performant method.	→ WP4
PP.3.1.2.d	Maestro's application-level interface should allow the user's intentions to be translated to Maestro, which will seek to satisfy the user's intentions while delaying or modifying data movement according to the Middleware's own decision making framework.	✓
PP.3.1.2.e	Context and meaningful data movement semantics should be communicable in the Maestro infrastructure, requiring core abstractions and API to allow for this	→ WP4
PP.3.1.2.f	Since Maestro should self-manage data during certain epochs, a common interface to all memory levels should be developed. This interface should also be extended to provide a common explicit data movement interface.	✓
PP.3.1.2.g	Maestro should explore the state-of-the-art in memory system modelling and either adopt or develop an appropriate modelling framework to support coarse-grained data movement optimisation decisions and finer-grained data access optimisation decisions.	→ WP4
PP.3.1.3.a	Maestro should survey the available abstractions that provide desired functionality and develop the missing abstractions where existing solutions do not yet exist.	✓
PP.3.1.3.b	Maestro should build memory performance tools and queries that are higher-level than existing tools like hwloc.	→ WP4
PP.3.1.3.c	Maestro abstractions should store additional metadata describing structured context where it exists, e.g. the iteration domain that a structured dataset was derived from.	storage→ metadata, acquisition. → WP4
PP.3.1.3.d	Maestro abstractions should allow opaque data to be represented, as well as grades of opaqueness/structure.	✓
PP.3.1.3.e	Maestro abstractions should allow relations between data objects to be resolved, especially data, task and workflow dependencies.	→ WP4

PP.3.1.3.f	Maestro will provide a *data object blocking* transformation `CREATE_CDO_GROUP_FROM_CDO` which creates a *group* (set) of data objects with specified qualities from a single data object. After blocking transformation, each data object will reference a subset (i.e. below a certain capacity threshold) of the original data object's data.	Groups, WIP
PP.3.1.4.a	A Maestro library API and library-only mode will be available that provides a subset of the middleware capabilities, while excluding any client-server operations, resilience features, namespace and inter-application communications. Mixing of library-only and middleware modes so that for example, transformed data is later used in a workflow, should be fully supported but with restrictions.	✓
PP.3.1.4.b	Maestro will allow objects to be requested using a given layout scheme. That layout scheme will be resolved by Maestro at the moment that objects are given to the Consumer.	✓
PP.3.1.4.c	Maestro will abstract the movement of data between distinct memories in the system, for example between CPU and accelerator memory. A memory model (see PP.3.1.2.g) is a prerequisite for this feature to be enabled.	✓
PP.3.1.4.d	Maestro will provide a tiling abstraction, including static code analysis, programming abstraction and runtime support used to tile data and promote silently to faster memory. This feature will be developed in collaboration with the EPiGRAM-HS project as some parts of its implementation lay outside the scope of Maestro.	Mamba, WIP
PP.3.1.4.e	Maestro will capture the original ordering of a loop-nest as a *schedule* and will allow a set of transformations to be applied on that schedule, effectively returning a new schedule (ordering).	→ WP4
PP.3.2.2.a	Maestro should allow indexing of lists of data object names; nesting of data objects is conceivable but not planned.	✓
PP.3.2.2.b	Maestro should allow CDO naming (or name matching) based on existing application or middleware language naming conventions.	✓

PP.3.2.2.c	Maestro should detect potentially redundant loads/stores within defined epochs of temporal locality; Maestro should manage storage resources to enable communication (hot caching) in the event of redundant loads/stores.	→ WP4
PP.3.2.2.d	Maestro's data-object declarations should incur very low overhead; Communications of data objects should use the fastest interconnects and transfer API available (e.g. avoid the PFS and use the interconnect network over MPI).	→ WP4
PP.3.2.2.e	A distribution attribute should be possessed by Maestro's core data abstractions, allowing a node to understand, reference and translate data elements owned by any other node; Maestro should use the distribution specifier to perform automatic data object redistribution; Maestro should optimise data redistribution.	→ WP4
PP.3.2.2.f	Maestro should allow array/tensor-based transformations in the form of i) layout transformation upon data object retrieval ii) explicit data object transformations.	D3.4
PP.3.2.2.g	Data-object dependence should be combined with a Maestro-controlled CDO lifetime, and temporal locality epochs.	→ WP4

6.2 Applications

Further requirements for the Maestro core from applications, as per D2.3 [5], sorted by core categories instead of applications:

6.2.1 Basics

Number	Description	Desirability	Status
R1.7a	Once data objects handed over to Maestro, transport and access must no longer be the application's concern.	Must have	✓
R1.7b	Maestro should take care of maintaining unique references to the data objects.	Must have	✓
R1.15b	The Maestro middleware must be able to cope with each producer running in parallel on multiple nodes and without explicit synchronisation between the producers. The producers mostly run independently from each other.	Must have	✓
R1.15d	The Maestro middleware must be able to cope with multiple consumers running in parallel on multiple nodes without need for synchronisation. The consumers will also run independently from each other.	Must have	✓
R1.16b	The Maestro middleware must be able to support multiple workflows running and producing objects simultaneously.	Must have	✓[2]
R1.24	Once data objects are handed over to Maestro, they must be immutable.	Must have	✓
R3.1	Grant ownership of data to Maestro	Must have	✓
R3.3	Require data owned by Maestro	Must have	✓

6.2.2 Configuration

Number	Description	Desirability	Status
R2.6	Maestro should provide a configuration system to describe properties of the workflow/pipelines for which it will manage the data exchange, where these properties are known in advance.	Should have	→ WP4
R2.7	In "streaming" mode, a consumer should have the opportunity to declare in advance (in configuration) the data arrays it will consume.	Should have	→ WP4

[2] Maestro Core Pool Manager (PM) sense of workflow separation is only a name (`MSTRO_WORKFLOW_NAME`). Using the same given name, clients – regardless of their originating user workflow and regardless of WP4 workflow management – can connect to the PM and operate Maestro normally.

6.2.3 Telemetry

Number	Description	Desirability	Status
R1.14	Maestro to record and communicate usage statistics – type of storage used and their load, write (production) rate, read (consumption) rate, etc. – to human operators or system monitoring/logging tools.	Should have	→ WP5

6.2.4 Failure Resilience

Number	Description	Desirability	Status
R1.10	When an error occurs in the Maestro software/system that impacts the ability of the Maestro system to correctly satisfy requests, it should be treated as a hard failure and be propagated to all other Maestro-dependent instances.	Must have	✓
R1.11	Maestro may have further 'soft' failure modes, if these do not impact the ability to access data according to the specification. As an example, if a duplicate copy of an object is lost, this may impact performance and may violate resiliency guarantees, but it is not a hard error from the perspective of the workflow.	Optional	✓
R1.12	Errors propagate from Maestro to dependent workflow components when they next access the Maestro API.	Should have	→ WP4
R1.13	Maestro may have a mechanism to enable an out-of-band workflow component to inject knowledge and data into it outside of normal process.	Should have	→ WP4
R1.17	Objects within Maestro are handled transactionally. No partial state must exist.	Must have	✓
R1.18	The way Maestro handles objects in case of errors within Maestro should be well-defined and consistent.	Should have	✓
R1.19	Objects within Maestro are handled consistently. No risk of undefined behaviour or data corruption.	Must have	✓

6.2.5 Memory Management

Number	Description	Desirability	Status
R3.2	Indicate Maestro to move or allocate data to a particular tier of memory	Must have	✓
R3.4	Maestro is able to manage memory resources (including allocation and deallocation) across multiple layers of the memory hierarchy.	Must have	✓
R3.5	Data can be moved synchronously or asynchronously	Should have	✓
R3.6	Maestro can move data from one memory space to another	Must have	✓
R3.7	Maestro can manage different memory spaces within a node	Must have	→ WP4
R3.9	Maestro should manage (re)use of the memory space with minimal overhead	Should have	→ WP4

6.2.6 Transformations

Number	Description	Desirability	Status
R4.1	Redistribution of parallel 2D fields	Must have	→ D3.4
R4.2	Row and column-major layout transformations of distributed arrays	Should have	→ D3.4

6.2.7 Stress Conditions

Number	Description	Desirability	Status
R1.15a	The Maestro middleware must be able to cope with around four hundred producers declaring and writing a total of at least 20,000 objects per second. The Maestro middleware must also be able to sustain that rate for the duration of a time-critical operational run, which is around one hour.	Must have	→ WP6
R1.15b	The Maestro middleware must be able to cope with consumers making time-critical read operations for up to 70% of the written time-critical data.	Must have	→ WP6
R1.16	Maestro to be able to cope with the creation of at least 150TiB data in an hour per workflow.	Must have	→ WP6
R1.16a	The Maestro middleware must be able to support non-time-critical consumption of around 80% of the data produced by time-critical operations.	Must have	→ WP6
R1.28	Maestro must be able to cope with multiple producers and multiple consumers accessing objects simultaneously and in a time-critical fashion.	Must have	→ WP6

6.2.8 Metadata

Number	Description	Desirability	Status
R1.1	Objects handled within Maestro can be described and handled according to user-defined domain-specific metadata.	Must have	✓
R1.2	User-defined metadata should take the form of a dictionary of key-value pairs, or equivalently as a set of arbitrarily named attributes.	Should have	✓
R1.3	Object retrieval should support metadata queries that describe lists of values.	Must have	WIP
R1.3a	Object retrieval must support retrieving all objects of a list of metadata values as a single operation.	Must have	WIP
R1.3b	Object retrieval must support retrieving all existing objects of a list of metadata values even if not all requested objects are available.	Must have	WIP
R1.3c	Object retrieval must support querying which of a list of metadata values are currently available for retrieval.	Must have	WIP

R1.3d	Object retrieval must support querying which of a list of metadata values the Maestro middleware expects to be available in the future, even if they are not currently available.	Must have	WIP
R1.5	Object retrieval may support wildcards, such as 'all' and 'every'. This may be interpreted as all available data matching the request.	Optional	WIP
R1.7c	Maestro should facilitate locating objects according to their metadata.	Must have	WIP
R1.26	Consumer applications should not be required to know the size of the data prior to requesting it.	Should have	✓
R1.27	Objects handled within the Maestro middleware can be described by three sets of attributes: • one specifying the properties of the run (e.g. operational or research, starting date/time), • one describing the data-collocation policy (typically the horizontal and vertical representation of the field), • and one defining the specific properties of the field (i.e. which parameter at what output step and vertical level). These three together could be made equivalent to describing objects according to user-defined metadata, although it is a much less desirable solution than R1.2.	Must have	✓(user-defined attributes)

R2.1	Maestro must allow a conceptual data organization based on the following data container concepts: • array: a multi-dimensional typed array • record: a grouping of related datasets published at the same time (e.g. simulation output fields at a specific time step) • collection: a grouping of related records (e.g. checkpoint data collection, post-processing data collection)	Must have	✓(user-defined attributes)

R2.2	Maestro must provide the ability to add (and query) named attributes (metadata) to data arrays, records and collections	Must have	✓(user-defined attributes)
R2.3	Maestro should provide a way to hierarchically group related data arrays within a record (and query this group structure), similar to HDF5 groups.	Should have	✓(user-defined attributes)
R2.4	A data collection must accommodate records produced by multiple independent producers. Hence, labelling of data records is at least a 2-tuple made of a sequence integer (the sequence number of a record with the list of records) and a producer ID integer. Maestro must provide an indexing or attribute system to be able to support this.	Must have	✓(user-defined attributes)

6.2.9 CDO Groups

Number	Description	Desirability	Status
R1.21	A single consumer must be able to iterate over a set of objects, even when this is too large to fit in memory at once.	Should have	→ WP4
R1.23	Once consumers have obtained a data set and started iterating over the elements, Maestro must manage its resources so that it will not be out-of-memory-killed at least until the iteration completes.	Should have	→ WP4
R1.25	Maestro may support iteration granularity larger than a single data object.	Optional	WIP

6.2.10 Lifetime

Number	Description	Desirability	Status
R1.8	Objects may have a minimum lifetime (possibly zero).	Must have	✓
R1.9	Objects may have a maximum lifetime (possibly infinite).	Optional	✓
R2.8	Maestro must provide the ability to persist data beyond the lifetime of the workflow that generated it, and make it available to appropriate consumers.	Must have	WIP; `simple_archiver` example component to be extended
R2.9	When used for persisting data produced by an application, Maestro should provide the ability to configure which data collection is persisted.	Should have	WIP; depends on resource description (WP4)
R2.10	When used for persisting data produced by an application, Maestro should provide the ability to specify a subsampling rate of persistence or filter.	Should have	WIP; `simple_archiver` component extension

6.2.11 Misc.

Number	Description	Desirability	Status
R1.6	Support describing a set of fields as 'complete'.	Optional	CDO group for the set can be `OFFER`ed
R1.29	The Maestro middleware should be able to distribute/place objects among nodes according to hints supplied by third-party components.	Should have	Discussions started
R2.5	In "streaming" mode, a data collection should accommodate multiple consumers organized into groups, similar to Kafka groups (or Redis Stream consumer groups). Records are broadcasted to all consumer groups. Consumers within a consumer group retrieve records from a queue.	Should have	`SUBSCRIBE` to CDO `OFFER` permits implementing this, needs more discussion in WP4

A Internal Tests

Some unit tests focus on internal API testing, and are therefore less attractive entry points to Maestro users.

Name	Purpose
check_symtab	check symbol table operations
check_protobuf_c	protobuf messages serialization
check_transport_gfs check_transport_mio	GFS transfer MIO transfer

B Metadata Schema

As defined in `./attributes/maestro-schema-schema.yaml`:

```
# Every schema needs to have a descriptive name
schema_name: str()
# ... and a version number
schema_version: int(min=0)

# It may define a namespace prefix to permit using shorter (relative)
# attribute names in the definitions later
schema_namespace: regex('(^$)|(^\.$)|(^(\.[^\s]+)+\.$)',
                        required=False,
                        none=False, multiline=False,
                        name='fully_qualified attribute namestring')

# If non_elementary types are to be used in the attribute definition
# these must be declared
schema_types: list(include('maestro_user_type_def'),
                   required=False)
schema_type_values: list(include('maestro_user_typespec'),
                    required=False)

# A Maestro schema needs to be a valid yaml document
maestro_attributes: list(include('maestro_attribute_def'), required=False)
---
maestro_attribute_def:
  key: regex('(^(\.[^\s\.]+)+$)|(^[^\s\.]+(\.[^\s\.]+)*$)',
             required=False,
             none=False, multiline=False,
             name='valid attribute namestring')
  type: include('maestro_user_typespec')
  required: bool()
  default: Any(include('maestro_builtin_typeval'),
               include('maestro_user_typeval'),
               required=False, none=False)
  documentation: str(multiline=True)
```

```
maestro-builtin-typespec: any(regex('^str\(.*\)$'),
                              regex('^bool\(.*\)$'),
                              regex('^int\(.*\)$'),
                              regex('^map\(.*\)$'),
                              regex('^enum\(.*\)$'),
                              regex('^map\(.*\)$')
                              )
---
maestro-builtin-typeval:  Any(bool(),
                              int(),
                              enum('None'),
                              str())
---
maestro-user-typespec: any(include('maestro-builtin-typespec'),
                           map(),
                           list(),
                           str())
---
# things that can be checked directly
# plus
# things that can be run-time checked
maestro-user-typeval: any(include('maestro-builtin-typeval'),
                          include('maestro-builtin-typespec')
                          )
---
maestro-user-type-def:
  typename: str()
  typespec: include('maestro-user-typespec')
  documentation: str()
---
```

References

[1] Christopher Haine and Utz-Uwe Haus. *Maestro D5.5*, 2020.

[2] Christopher Haine, Utz-Uwe Haus, and Adrian Tate. *Maestro D3.1*, 2019.

[3] Christopher Haine, Utz-Uwe Haus, and Adrian Tate. *Maestro D3.2*, 2019.

[4] Christopher Haine, Utz-Uwe Haus, and Adrian Tate. *Maestro D6.1*, 2019.

[5] Domokos Sarmany, Simon Smart, Julien Capul, Francois Tessier, and Salem El Sayed. *Maestro D2.3*, 2020.

[6] Domokos Sarmany, Simon Smart, Teodor Nikolov, Julien Capul, Sebastien Morais, and Francois Tessier. *Maestro D2.1*, 2019.