HORIZON 2020
TOPIC FETHPC-02-2017
Transition to Exascale Computing

Maestro
801101

D3.1
Initial Core Middleware Specification, API Document

WP 3: Core Memory and Data Aware Middleware

Christopher Haine
Utz-Uwe Haus
Adrian Tate

Date of preparation (latest version): February 2019

DOCUMENT INFORMATION

Deliverable Number	D3.1
Deliverable Name	Initial Core Middleware Specification, API Document
Due Date	February 2019 (PM 6)
Deliverable lead	CRAY
Authors	Christopher Haine Utz-Uwe Haus Adrian Tate
Responsible Author	Utz-Uwe Haus (CRAY) e-mail: uhaus@cray.com
Keywords	[Memory Systems, Data Middleware, HPC]
WP/Task	WP 3/Task(s) 3.1
Nature	R
Dissemination Level	PU
Planned Date	February 2019
Final Version Date	February 2019
Reviewed by	Dirk Pleiter (JUELICH)

DOCUMENT HISTORY

Partner	Date	Comment	Version
CRAY	18/2/2019	Final version before review	0.1
CRAY	27/2/2019	Final version after internal review	1.0

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.

The following requirements (amongst many others) are discussed. Maestro should:

- solve general ubiquitous problems of data-movement
- develop a middleware layer, upon which other software components can be developed.
- present an API that is useful at any layer of the software stack
- either adopt or develop a modelling framework for i) coarse-grained data movement and ii) finer-grained data access optimisation
- survey the available abstractions that provide desired functionality and develop the missing abstractions where existing solutions do not yet exist
- allow relations between data objects to be captured and resolved
- enable explicit movement of data between distinct memories (e.g. between CPU and GPU memory)
- abstract the movement of data between distinct memories in the system, for example between (e.g. between CPU and GPU memory)
- enable the promotion of data objects to available memory locations
- 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
- provide a tiling abstraction and support library, used to tile data and promote silently to faster memory.
- provide a distribution attribute within the core data abstractions, allowing a node to understand, reference and translate data elements owned by any other node
- enable implicit usage, providing e.g. POSIX library overriding during/ under certain timeframes/conditions.
- allow array/tensor-based transformations in the form of i) layout transformation upon data object retrieval ii) explicit data object transformations.

The full list of more detailed requirements is in Section 3.
The full Initial Specification of Maestro is described in Section 4. Maestro will present:

- three operating modes (middleware (with management infrastructure), library-only (no management infrastructure and library API) and hybrid (full set of features)

- a set of core abstractions providing memory awareness and data awareness.
- an interface to every memory level, enabling core functionality to that memory.
- expression of data dependencies that can be expressed at the data-object level.
- a high-level interface that allows the user to express dependencies between CDOs, and informs Maestro of the location of CDOs across a workflow.
- self-allocation of resources for user data where needed, fully configurable by the user.

The custom abstractions presented by Maestro Core are:

- a Memory Object that represents a set of locations that can be used for persistent and/or non-persistent storage of data, an allocation and an interface.
- a System object, which is a set of memory objects represented as a graph, and which can be used to query the transport rates between two memories.
- a Context object, that captures higher-level classifiers telling Maestro what types of Scope objects and CDOs to expect.
- a Scope object that captures any available information pertaining to the scope, size, access relations (indexing), schedules (ordering) of the data and semantic information pertaining to the application's usage of data-structures.
- a Core Data Object (CDO), typically formed from a Scope Object and a Memory Object, representing real data and their physical location (if known). CDOs that do not represent real data or represent unspecified data are also legal. It combines all available information known and is how an application and Maestro communicate data intentions.

Two major interfaces are presented by Maestro:

1. the Management API: A 4-step protocol ('Give-Take' semantics) for the giving of data objects to Maestro and the taking of data objects back from Maestro:

   1) `DECLARE` (CDO declaration): allows an application to describe a CDO to Maestro.
   2) `OFFER / REQUIRE` (CDO pool injection): Producer/Consumer (respectively) gives CDOs to management pool
   3) `WITHDRAW/DEMAND` (CDO pool retraction): Producer/Consumer (respectively) takes CDOs from the Management pool
   4) `DISPOSE` (CDO disposal): Delete CDO

   The CDO management API can perform CDO-level silent memory promotion at the data object level via *memory wildcards*.

2. a Transformation API that is concerned with layout-sensitive transformations, is applied to mutable data and does not return new CDOs as outputs. Some transformations (e.g. Permute, Split, Fuse) operate directly on a Scope object while others (e.g. Align, Reshape) are applied to CDOs.

   A tiling abstraction is presented that allows the user to program to an explicit tile data-structure, the location of which will be modified by Maestro rather than the user.

3. a Hybrid API, for those users with a clear interest in both layout-insensitive management, and layout-sensitive transformation, to e.g. allow layout 'transformations on `REQUEST`'.

A set of CDO qualities are described:

- a set of CDO Accessors.
- a CDO Convenience API.
- CDO attributes (name, allocation, state, lifetime, redundance).
- CDO levels (0 (name only), 1 (moveable), 2 (detailed).
- CDO lifetime.
- one of three lifetime handling modes: Hard, Soft or Ignored deadlines.
- CDO Groups (sets) are supported by Maestro Core.
- CDO namespaces including wildcards.
- a unique identifier (CDOID), which can be queried but not influenced by the application.

Contents

1 Introduction

1.1 About This Document

This document forms the initial presentation of a fundamental technical deliverable of the Maestro project, the *Data- and Memory-aware Middleware*, referred to herein as *Maestro Core*. In Maestro, the middleware is co-designed based on the application requirements, derived during the project from Maestro's set of key target applications.

The sequence of project developments for middleware definition, design and development is

D3.1 Initial Middleware Specification (M6)

MS2.1 Initial workflow requirements communicated to other work packages (M6)

D2.2 Workload Characterisation and Middleware Requirements (M9)

D3.2 Full Middleware Specification and Reference Implementation (M12)

D4.1 Data Access Semantics (M12)

D3.3 Full Middleware Release (M24)

Technical requirements of Maestro applications have been partially ascertained at the time of writing this technical report. Since Maestro is a co-design project that evolves according to application requirements, the technical specification of Maestro Core remains subject to change until D3.2 in M12. This document does however serve as a general design agreement between partners allowing work in related work packages to proceed. Major modifications to Maestro Core (defined as those that may modify the behaviour in a way that may affect another work package) will therefore only be possible with an interim design document being distributed. Minor modifications (those that will not affect behaviour outside of WP3) will continue to be made until D3.2.

Section 3 of this report describes the Maestro project technical problem in a general sense and from the perspectives of the target applications. Section 4 provides the initial Maestro Core specification of the proposed solution to the problem specified in the preceding section. The subsequent deliverable D3.2 will finalise this specification and provide usage examples of the proposed middleware. Occasionally, this document shall describe a technical item for which insufficient information is available for full definition. Such items are labelled 'deferred to D3.2' to show that such items will be fully developed in the full middleware specification.

1.2 Maestro Project

Maestro is a FETHPC-2018 funded project that is seeking to design a data- and memory-aware middleware framework for HPC applications and workflows. The abstract for the Maestro project reads

> Maestro will build a data-aware and memory-aware middleware framework that addresses ubiquitous problems of data movement in complex memory hierarchies and at many levels of the HPC software stack. Though HPC and HPDA applications pose a broad variety of efficiency challenges, it would be fair to say that the performance of both has become dominated by data movement through the memory and storage systems, as opposed to floating point computational capability. Despite this shift, current software technologies remain severely limited in their ability to optimise data movement. The Maestro project addresses what it sees as the two major impediments of modern HPC software: 1. Moving data through memory was not always the bottleneck. The software stack that HPC relies upon was built through decades of a different situation – when the cost of performing floating point operations (FLOPS) was paramount. Several decades of technical evolution built a software stack and programming models highly fit for optimising floating point operations but lacking in basic data handling functionality. We characterise the set of technical issues at missing data-awareness. 2. Software rightfully insulates users from hardware details, especially as we move higher up the software stack. But HPC applications, programming environments and systems software cannot make key data movement decisions without some understanding of the hardware, especially the increasingly complex memory hierarchy. With the exception of runtimes, which treat memory in a domain-specific manner, software typically must make hardware-neutral decisions which can often leave performance on the table. We characterise this issue as missing memory-awareness. Maestro proposes a middleware framework that enables memory- and data-awareness.

Work Package 3 (WP3) of the Maestro project is concerned with developing the core middleware framework that underpins the proposed technical solution. Since Maestro is a co-design project, the initial period of WP3 has been spent developing technical requirements from the Maestro consortium including the application owners. Since Maestro is a very ambitious and multi-disciplinary project, its technical specification is complex, requiring technical clarifications and advancements to be made at multiple levels of abstraction almost simultaneously. To mitigate the obvious risks in this process, the Maestro consortium has already provided successive iterations of feedback into the emerging technical specification, and will continue to do so until the final specification in deliverable D3.2.

Maestro tackles many data movement problems across the entire memory-storage hierarchy. A wide variety of technical concerns span this problem scope, as illustrated in Figure 1. While most technical concerns can be separated into one of two distinct groups that we shall call “transformations” on the left hand side and “data object management” on the right hand side, a small number of technical considerations lay in the intersection. Maestro’s ability to tackle the wide variety of technical issues lays in responding meaningfully to these two groups, as well as to items that lay in the intersection. Maestro will introduce a set of underlying abstractions that enable it to detect and respond to such varieties in context, in particular the Scope Object and the Core Data Object, which are described in Section 4.2. Although Maestro needs to respond to such variety of technical scenarios, we wish that a level of symmetry allowing identical or similar API and usage models are retained in either case. As such, the Maestro Technical Specification will describe two key APIs, the transformation API described in Section 4.3.2 and the Data Object Management API described in Section 4.3.1. As illustrated in Figure 1 these two APIs address broadly separable domains, though a small number of requirement lay in the intersection of the two. This document describes the required scope and functionality of both APIs, but does not contain a detailed API specification which will be provided in deliverable D3.2 and D3.3.

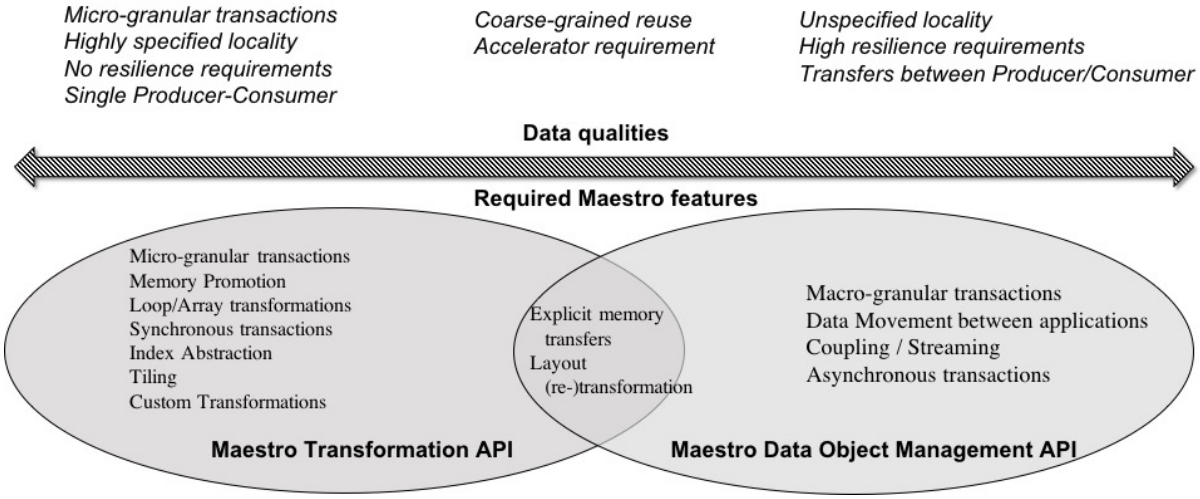

Figure 1: Scope of data transformations and management

2 Terminology/Glossary

Term	Meaning
Core Middleware / Maestro Core	The internal software layer that constitutes the Maestro framework
Core Data Object (CDO)	Fundamental data unit understood by Maestro and communicated between Maestro and Application
Core Data Model	Maestro's model of data and data movement
CRUD	Create, Read, Update and Delete
HBM	High Bandwidth Memory
HDD	Hard Disk Drive
MPI	Message Passing Interface
PFS	Parallel File System
SSD	Solid State Drive
WLM	Workload Manager
Workflow	A set of computational tasks that require scheduling due to task or data dependencies
[Workflow, Task, Data] Dependency	A relation that one workflow component, task or computational element, cannot be safely completed until another component has been computed.

3 Problem Specification and Technical Requirements

This section describes the technical problem that the Maestro Project and thus the middleware attempts to solve. The problem specification covers the general technical problem of data movement (Section 3.1) and the technical requirements arising from the Maestro Applications (Section 3.2.2). We present a taxonomy of the problem space in the following subsections, with each specific problem accompanied by a technical proposal.

We introduce here as a placeholder definition an abstract *data object* which the reader should view as a general abstract representation of user data encompassing *yet-to-be-defined features*. This abstract data object is a placeholder for the fundamental *Core Data Object (CDO)* which is defined in Section 4.2.5 and which the reader may instead prefer to read in advance.

Throughout the remainder of this document, these individual proposals will be referenced using, for example, PP.3.1.1.a to refer to the first proposal in Section 3.1.1.

3.1 General Problem of Data Movement

The so called "data-movement problem" is an umbrella term for a multitude of related technical problems of varying scope, complexity and level of abstraction, the full range of which is explored in the following subsections.

3.1.1 Inadequacy of the Software Stack for Data Movement Optimisation

(a) HPC software is increasingly required to perform data movement and access optimisations in order to address the emergence of memory and storage bottlenecks in applications.
Proposal: Maestro should introduce a solution to general ubiquitous problems of data-movement, without being constrained by the limited scope of existing solutions.

(b) The modern bottlenecks lay usually in the memory and storage systems while the software stack was primarily designed in an era when this was not the case.
Proposal: Maestro should generate new data- and memory-aware abstractions for data handling, management, movement and transformation

(c) The solution to data-movement problems may lay outside the scope of one single canonical software stack component (e.g. Compiler, Workload Manager).
Proposal: Maestro should develop a middleware layer, upon which other software components spanning systems software and programming environments (amongst others) can be developed.

(d) Many data movement optimisations cannot be performed entirely at compile time, i.e. based on static analysis, since some important parameters are not instantiated until runtime (e.g. symbolic constants for array leading dimensions) and cannot be predicted in a general manner.
Proposal: Maestro should enable runtime analysis of data objects.

(e) Many data movement optimisations cannot be performed entirely at runtime since many basic access relations cannot be easily detected without significant overhead

(e.g. array access functions, data dependencies).
Proposal: Maestro should enable compile-time analysis of data objects. In combination with PP.3.1.1.d Maestro should combine static and runtime analysis.

(f) Within HPC, there is a need for optimisation to occur inside legacy frameworks that are using standard interfaces such as POSIX.
Proposal: In addition to an explicit interface, Maestro should enable implicit usage, providing e.g. POSIX library overriding during/ under certain timeframes/conditions. When Maestro is disabled, the application proceeds as it would have done before Maestro modifications, and with no performance loss compared to the unmodified version.

(g) A class of problems exist that are specific to the layout and access patterns of application data. The technical considerations may be quite distinct to the problems addressed by other parts of the Maestro middleware, e.g. tiling and loop ordering while other technical requirements may play little to no role, e.g. resilience.
Proposal: Maestro core abstractions should be able to detect the specific case of layout sensitive transformations, to adapt to this situation and to store sufficient scope to inform decision making in this area.

(h) A class of problems exist that are mostly layout independent. The technical considerations relating to these problems may be quite distinct, e.g. strong resilience requirements, while other technical requirements will play little to no role in this area, e.g. layout transformations
Proposal: Maestro core abstractions should be able to detect the specific case of layout insensitive transformations, to adapt to this situation and to store sufficient scope to inform decision making in this area.

(i) The intersection of the classes of problems described in PP.3.1.1.g and PP.3.1.1.h may be non-empty.
Proposal: Although Maestro core abstractions should adapt to the situations of functionality described in PP.3.1.1.g and PP.3.1.1.h, they should also adapt to the situations where technical qualities of either class may be required.

3.1.2 Lack of useful Data Access Interfaces and Memory Interfaces

(a) Different software components in the stack (e.g. OS, compiler, WLM, runtime) have very different ways of accessing data, different data-structures are required, and different APIs are needed.
Proposal: Maestro should present an API that is useful at any relevant layer of the software stack, including at the very least multiple levels in the programming environments, systems software and runtimes areas.

(b) Different levels of the memory-storage system (e.g. DRAM, HBM, SSD) store data with different access performance characteristics and possibly in different layouts and/or different chunk/page sizes.
Proposal: Maestro should be aware of the nuanced variety of system-specific performance characteristics as well as caching behaviour and take these into consideration when performing data accesses and movements.

(c) An application may possess different ways of accessing stored data at a specific level, e.g. data stored in DRAM can be accessed through a C-array or through Ramdisk.
Proposal: Maestro should recognise the variety of means by which data can be stored in a certain medium, however for internal transformation purposes it will choose the most appropriate or performant method.

(d) User/application intentions dictate the likely choice of access method, which may be required to change, e.g. computation on data requires arrays in DRAM, but required persistence of the computed results typically requires POSIX file-based storage on HDD.
Proposal: Maestro's application-level interface should allow the user's intentions to be translated and communicated direcly to Maestro, which will then seek to satisfy those intentions while delaying or modifying data movement according to the middleware's own reasoning framework.

(e) No common data-structures (e.g. C-array, POSIX file) possess sufficient metadata to communicate meaningful semantics to another software components.
Proposal: Context and meaningful data movement semantics should be communicable in the Maestro infrastructure, requiring core abstractions and API to allow for this.

(f) No common memory interfaces exist that allow (for example) allocation of arrays at any level in the memory hierarchy
Proposal: Since Maestro should self-manage data during certain times/under certain conditions, a common interface to all memory levels should be developed. This interface should also be extended to provide a common explicit data movement interface.

(g) Robust and general models of the memory hierarchy are not available
Proposal: Maestro should explore the state-of-the-art in memory system modelling and either adopt or develop an appropriate modelling framework to support i) coarse-grained data movement optimisation decisions and ii) finer-grained data access optimisation decisions. It is possible (but not certain) that one modelling framework cannot cover both scenarios.

3.1.3 Lack of Data- and Memory Aware Abstractions

Data- and memory-aware abstractions are not in widespread usage. Memory-awareness and data-awareness are clearly distinct, compound sets of issues that are unpacked in the following set of problems/proposals.

(a) Data- and memory-aware abstractions are not in widespread usage.
Proposal: Maestro should survey the available abstractions that provide desired functionality and develop the missing abstractions where existing solutions do not yet exist.

(b) Abstractions capturing basic memory usage and performance information at a level high-enough for analysis purposes do not exist

Proposal: Maestro should build memory performance tools and queries that are higher-level than existing tools like hwloc, which are too node-centric, are missing node type classes, are missing performance characteristics, or make simplistic data movement cost assumptions.

(c) Typical simulation applications, which may be considered being "classical HPC", often allow analytic understanding of data placement. In this case, additional semantic context would be useful to be captured
Proposal: Maestro abstractions should store additional metadata describing structured context where it exists, e.g. the iteration domain that a structured dataset was derived from.

(d) Data Analytics and other forms of data-intensive computing may produce data with little or no structure
Proposal: Maestro abstractions should allow opaque data to be represented, as well as grades of opaqueness/structure.

(e) Complex Workflows cannot be resolved through static dependency analysis and may require structured queries on metadata (attributes).
Proposal: Maestro abstractions should allow relations between data objects to be captured and resolved, especially data/task/workflow dependencies.

(f) Maestro users may wish to receive data objects that exceed the amount of physical memory available
Proposal: Maestro will provide a *data object blocking* transformation which creates a *group* (set) of data objects with specified qualities from a single data object. After the blocking transformation, each data object will reference a subset (i.e. below a certain capacity threshold) of the original data object's data.

3.1.4 Lack of Data-based Transformation Capabilities

As detailed in PP.3.1.1.g, a class of problems exists that is sensitive to data layout, and the converse situation is also true (as detailed in PP.3.1.1.h). Maestro must adapt to either scenario. As such, Maestro users must be able to access the appropriate parts of the functionality and to not be necessarily encumbered by the full functionality and restriction of the whole middleware. Explicit usage of different memories, including accelerator memory is considered a transformation and so included here.

(a) Users interested only in layout-sensitive functionality within Maestro should be unencumbered by the full usage model and setup/configuration obligations of the Maestro middleware
Proposal: 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 such that for example, transformed data is later used in a workflow, should be fully supported but with restrictions.

(b) A Consumer may require data to be organised differently to how it was provided by a Producer

Proposal: 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.

(c) Users may explicitly move data between distinct memories (e.g. between CPU and GPU memory)
Proposal: 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.

(d) Maestro users may wish for data objects to be promoted to available memory locations, exploiting the memory- and data-awareness that is only available through Maestro
Proposal: 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.

(e) Many transformations are applied at the level of a schedule/ordering and do not require movement of explicit data
Proposal: 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).

3.2 Prologue to Maestro Applications Co-Design

3.2.1 Introduction to Maestro Applications

3.2.1.1 The Integrated Forecasting System (IFS) We focus on the product generation step of the ECMWF workflow: the IFS code is performing the simulations for a 10-day forecast (240 time steps) in about an hour, generating about 20TB of data. At each iteration around 20 fields are written out to the Field Database Server (FDB), effectively an intermediate disk staging area. From there an archiving process takes care of permanently storaging data, and all individual requests for data subsets (one field's value on a subset of the special grid) are produced by product generation tasks. During one forecast run about 10-12 Million products are generated. The fields which are distributed across nodes inside the IFS simulation are already being aggregated by separate IO-nodes, so that one field is written from exactly one IO-node (assignment is round-robin, not fully fixed). A number of technical requirements are derived from this workflow: Data objects need to be generated and addressable at very high frequency: the smallest addressable unit should be one field (time step, height level, ... fixed; all 2-d grid points), leading to about 14000 such objects being created per second. At the same time, objects should be referenceable in all the natural groupings that arise in the application: same quantity at same time at different heights, or all quantities at the same time step and height, etc. A Middleware should be able to actively perform data colocation or disaggregation to permit such 'sliced' access to related objects. For each product generation run the set of products requested is gathered from customers 'just in time'. A workflow manager exists (ecflow),

which permits chaining of tasks including programmed and/or conditional execution of tasks. ecFlow will schedule the necessary product generation tasks in a master-worker scheme that tries to maximize use of a given input field for multiple customer requests. For the purposes of this project a simpler benchmarking setup (Kronos framework) may be used. The main requirements from this use case are thus: a low-intrusion API to annotate data objects in the IO server code, and workflow coordination of the product generation tasks to directly schedule data movement of the freshly generated fields to the product generation through the interconnect, without requiring reads of the on-disk data in the FDB. The FDB should ideally be populated by the same mechanism transparently.

3.2.1.2 Global Earth Modelling Three simulations (COSMO, an atmosphere model; CLM, a land model ; ParFlow, a hydrologic model for surface and subsurface flow) are run concurrently in an MPMD setup and exchange field data using the OASIS coupling framework[1]. Fields are defined using the MCT framework. All simulations are run in compatible grid resolutions, but on varying numbers of nodes : ParFlow uses about twice the number of compute nodes than each of the other two applications. Thus data needs to be redistributed (2:1) or directly transferred (1:1). All data consists of 2-dimensional spacial grids. Grid resolution is static. Data is produced at a rate of 51 grid variables (140x140 doubles each), amounting to roughly 8M per step. Still, the coupled applications only scale to about 200 processes (20 nodes of 10 cores each), as it becomes communication bound after that. In addition to the TerrySysMP coupled application case there are preprocessing and postprocessing steps which are performed using the 'CDO climate toolkit'. This process is very inefficient and overlap of pre-/postprocessing with the simulation is not currently performed.

3.2.1.3 In-Situ Analysis in Computational Fluid Dynamics In this family of use cases the focus lays on improving the possible communication rate and frequency between simulation and visualization or analysis tasks, or in coupled simulations. All applications in these workflows already use a unified IO-Abstraction library (Hercule). The majority of data consists of spacial grids, and applications already annotate their data with field type attributes from an in-house catalogue. Hercule – triggered IO is currently done in a file-per-rank fashion on disk storage. A primary challenge is the shortage and bandwidth limitation of using the filesystem as intermediate storage: except for a rolling backward horizon of snapshot data for checkpoint/restart, intermediate data does not need permanent storage. Multiple consumer tasks will be reading the same data from disk concurrently or repeatedly. Furthermore, data redistribution happens at read time, even if it was known at write time, because Hercule is optimized for fast write-out. There is also no way to consume subsets of an iteration's data before the entire data set has been written out on all ranks. The major feature request of Maestro would be the buffering of data to avoid stalling the producer application, while controlling lifetime of sub-iteration data, redistributing it as needed, and flushing it to disk or dropping it when no longer needed by the workflow. A secondary challenge is the missing universal workflow manager; workflows are orchestrated ad hoc by scripts by the user. Only with proper workflow management support is it conceivable that the user can specify desired

[1] https://portal.enes.org/oasis

intermediate data lifetime, as well as checkpoint/restart or rollback abilities.

3.2.1.4 SIRIUS Library SIRIUS is a domain specific library for electronic structure codes developed at ETHZ/CSCS. It is a collection of classes that abstracts away the different building blocks of pseudopotential plane-wave and full-potential linearized augmented plane-wave codes. SIRIUS can extend legacy Fortran codes with API calls to a domain-specific library which runs on GPU and other novel architectures. The code is written in C++11 with MPI, OpenMP and CUDA programming models.

The DFT Loop mini-app is part of the SIRIUS package. This mini-app is used to benchmark all components of the DFT self consistency cycle – diagonalization of the Kohn-Sham Hamiltonian, charge density construction, mixing and generation of the effective potential – using a pseudopotential or full-potential method. DFT Loop is an interesting use-case for the Maestro middleware as significant data movements are involved and manually managed to properly use the GPU memory. When the data size cannot fit into the GPU memory, allocation/deallocation is manually done by blocks, meaning that the memory management is fully controlled by the developer.

In this context, the benefit Maestro could provide is two-fold: first, Maestro could optimize the movement of data between host memory (CPU) and HBM of the GPU, based on the communication requirements (targeted memory, timings, transport layer). Secondly, the middleware could insure the reusability of the allocated memory space. All of this would take advantage of the memory abstraction offered by Maestro for better performance-portability and extendibility to emerging memory and storage technologies.

3.2.2 Data Movement Requirements of Maestro Applications

Detailed Maestro application requirements will be described in deliverable D2.2 (M9). Where application-level requirements are already known, they are described in this section, which therefore does not represent a comprehensive set of data object/movement requirements at this time. Requirements stemming from applications are inevitably more specific and lower-level than the general requirements from previous sections. Requirements may be shared among different applications, or may derive from a single application. In either case proposals are labelled with an abbreviation - IFS (Integrated Forecasting System), GEM (Global Earth Modelling), CFD (In-situ Analysis and Computational Fluid Dynamics) or SIRIUS (the SIRIUS library).

(a) An application may require communication of data objects at a very high field-level resolution, for example each field at a given time step (IFS), or certain fields computed early in one iteration marked ‘ready’ before end of iteration (CFD)
Proposal: Maestro should allow indexing of lists of data object names; nesting of data objects is conceivable but not planned

(b) An application may wish to communicate individual fields of a data object at a certain iteration (IFS,CFD)
Proposal: Maestro should allow data object naming (or name matching) based on existing application or middleware language naming conventions.

(c) An application or workflow may read the same data objects from a memory level. This may occur across sequential elements of a workflow or in single-producer multiple-consumer situations (IFS,CFD)
Proposal: 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

(d) Applications and workflows may require high rates of i) data-objects generated per second and ii) data streaming rates. At the time of writing the required rates are estimated to be $\approx$ 14000 data-objects per second (IFS) and 100 GB/s streaming (CFD)
Proposal: 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).

(e) A non-trivial redistribution of data may be required in order to resolve a data-object dependence / data transfer. This may be expressed in the most general sense of data object redistribution from M nodes to N nodes (CFD) or less general situations of e.g. N nodes to N nodes (GEM) and N nodes to 2N nodes (GEM).
Proposal: 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.

(f) Applications may benefit from data layout transformations at the time of data object communication, for example transposition of (linearized) distributed 2d grid data (GEM).
Proposal: Maestro should allow array/tensor-based transformations in the form of i) layout transformation upon data object retrieval ii) explicit data object transformations.

(g) Applications allocate intermediate datasets with no forward-looking knowledge of the potential to reuse that data. A high-level data object dependence API could be exploited to allow more intelligent usage of temporaries, e.g. perform automatic disposal of intermediate data if they are known to be no longer needed in workflow (CFD) or temporary object re-use (SIRIUS)
Proposal: Data-object dependence should be combined with a Maestro-controlled CDO lifetime, and temporal locality epochs.

3.3 Summary of Proposals

The following tables summarise the set of requirements and proposals that have been considered in the middleware definition process, along with forward references to the appropriate technical specification sections.

Proposal Number	**Proposal Text**	**Addressed in Section(s)**

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.	4.1 "Overview and Usage"
PP.3.1.1.b	Maestro should generate new data- and memory-aware abstractions for data handling, management, movement and transformation	4.2 "Abstractions"
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.	4.1 "Overview and Usage"
PP.3.1.1.d	Maestro should enable runtime analysis of data objects	4.1 "Overview and Usage"
PP.3.1.1.e	Maestro should enable compile-time analysis of data-objects, meaning (due to with PP.3.1.1.d) that Maestro should combine static and runtime analysis	4.1 "Overview and Usage"
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.	4.1 "Overview and Usage"
PP.3.1.1.g	Maestro core abstractions should be able to detect the specific case of layout sensitive transformations, to adapt to this situation and to store sufficient scope to inform decision making in this area	4.3.2 "CDO Transformations" and 4.2.4 "Scope Object"
PP.3.1.1.h	Maestro core abstractions should be able to detect the specific case of layout insensitive transformations, to adapt to this situation and to store sufficient scope to inform decision making in this area	4.3.1 "CDO Management API" and 4.2.4 "Scope Object"
PP.3.1.1.i	Although Maestro core abstractions should adapt to the situations of functionality described in PP.3.1.1.g and PP.3.1.1.h, it should also adapt to the situations where qualities of both classes are relevant	4.2.4 "Scope Object" and 4.2.3 "Context Object"
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.	4.1 "Overview", 4.3.1 "CDO Management API" & 4.3.2 "CDO Transformations"
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	4.1.2 "Memory Awareness"

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.	4.1.2 "Memory Awareness" and 4.2.1 "Memory Object"
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	4.4.1 "CDO attributes"
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	4.1.3 "Data Awareness" and 4.2.5 "Core Data Object"
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	4.3.1 "CDO Management API"
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	4.1.2 "Memory Awareness"
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	4.2 "Abstractions"
PP.3.1.3.b	Maestro should build memory performance tools and queries that are higher-level than existing tools like hwloc	4.1.2 "Memory Awareness"
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.	4.2.4 "Scope Object"
PP.3.1.3.d	Maestro abstractions should allow opaque data to be represented, as well as grades of opaqueness/structure	4.2.5 "Core Data Object" & 4.4.1 "CDO Attributes"
PP.3.1.3.e	Maestro abstractions should allow relations between data objects to be resolved, especially data/task/-workflow dependencies	4.1.1 "Core Data Abstraction"

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..	4.4.3 "Groups of CDOs"
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 such that for example, transformed data is later used in a workflow, should be fully supported but with restrictions	4.1 "Overview and Usage"
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	4.3.3 "Hybrid API"
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.	4.3.1.5 "Memory Wildcards" (CDO-level) and 4.3.2.1 "Tiling Abstraction" (tile-level).
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.	4.3.2.1 "Tiling Abstraction"
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).	4.2.4 "Scope Object"
PP.3.2.2.a	Maestro should allow indexing of lists of data object names; nesting of data objects is conceivable but not planned	Section 4.4.4 "CDO Names and Namespace Support"
PP.3.2.2.b	Maestro should allow CDO naming (or name matching) based on existing application or middleware language naming conventions	Section 4.4.4 "CDO Names and Namespace Support", Section 4.4.3 "Groups of CDOs"

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	Agreed in principal but deferred to D3.2
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).	Agreed in principal but deferred to D3.2
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.	Section 4.2.4 "Scope Object"
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.	Section 4.2.4 "Scope Object", Section 4.3.2 "CDO Transformations"
PP.3.2.2.g	Data-object dependence should be combined with a Maestro-controlled CDO lifetime, and temporal locality epochs.	Agreed in principal but deferred to D3.2

4 Maestro Core Technical Specification (Initial)

4.1 Overview and Usage

This section presents the proposed functionality of the Maestro framework at a high-level, with technical details being provided in the subsequent subsections.

Maestro presents a middleware framework (PP.3.1.1.c) providing similar functionality to a diverse set of user applications spanning systems software, programming environments, runtimes and HPC applications (PP.3.1.2.a). Maestro presents a set of features broadly characterised as memory and data-awareness (PP.3.1.1.a), enabled via custom data- and memory-aware abstractions (PP.3.1.1.b).

Maestro data object analysis can be performed at two places/times i) at compile-time after data object annotations are made to a program (e.g. tiling of data) or implicit CDO declarations through directives ii) at runtime through declaration and usage of explicit CDOs. (PP.3.1.1.d and PP.3.1.1.e)

Following the three usage models shown in Figure 1, there are three corresponding operating modes of Maestro, which will also correlate to three values of the Context Object (Section 4.2.3).

1. A middleware service with client-server/manager setup used primarily to support workflow management via the CDO management API (Section 4.3.1). In this mode,

the application will provide data objects to the middleware with the assumption that these data objects will be managed by the middleware. Management will include both copy and transformation of data, as well as overriding of existing operations such as POSIX calls (PP.3.1.1.f). Simple workflow management and coupling of application data can be performed in this model with minimal code change (see Section 4.3.1) and using a high-level dependence API to express inter-application data object dependencies.

2. A library-only mode through which a lightweight set of services is available (PP.3.1.4.a), primarily used to access the transformation API (Section 4.3.2) and allowing layout-sensitive data-centric optimisations to be performed by Maestro. This mode allows applications using only the transformations API to not incur the setup, configuration and overheads of the management framework. A set of program-level abstractions are available in order to access advanced data-centric optimisations such as tiling for loop transformation.

3. In hybrid mode, the full set of transformation and management features are available though there are certain restrictions in place (precise definition of restrictions deferred until D3.2).

4.1.1 Core Data Object Abstraction

Maestro presents a minimal object-like abstraction, the "core data object" (PP.3.1.1.b) that captures and describes diverse data used by applications, and through which Maestro reasons about data movement, transformation and management. The same abstraction supports both the Management API and the Transformation API. By manipulating CDOs, Maestro can perform data transformation (PP.3.1.3.c), allow automated (and explicit) data object transport (PP.3.1.3.d), resolve data dependencies in workflows (PP.3.1.3.e) and manage precise data layout and access patterns.

The precise definition of the CDO along with its surrounding abstraction context is found in Section 4.2.

4.1.2 Memory Awareness

Maestro provides a memory system model (PP.3.1.2.g) through which the cost of moving, transforming or copying a CDO can be queried (PP.3.1.2.b). The memory system model is specific to the hardware implementation of the Maestro installation (PP.3.1.2.f). A user may query the memory system model to guide direct transformations on a CDO. The model may reveal basic latency and bandwidth numbers between nodes, or may refer to an empirical model revealing more nuanced performance information (PP.3.1.2.b). Maestro may query the memory system model before itself modifying or transforming a CDO silently (PP.3.1.2.c). An interface to every memory level is provided, enabling core functionality to that memory, including: `open, close, load, store, reserve` (PP.3.1.3.b). This interface may build on existing technologies if sufficiently robust solutions are known. Deliverable D5.3 will contain a more detailed specification of this functionality, and will also outline API aspects that permit performance analysis tools to inspect the usage (counters, logs) or provide profiling hooks.

4.1.3 Data Awareness

Data access and movement semantics of an application can be captured by combinations of CDOs and higher-level abstractions (Scope and Context objects, see Section 4.2.4) (PP.3.1.2.e).

Data dependencies can be expressed at the CDO level and can be explicitly encoded using a high-level API. Encoding of dependencies can be through named CDOs as well as implicit, wildcarded and virtual CDOs. Maestro will resolve CDO-level dependencies through CDOs placed into a management pool (see Section 4.3.1) (PP.3.1.3.c).

4.1.4 High-level Interface

A high-level interface allows the user to express dependencies between CDOs, and informs Maestro of the location of CDOs across a workflow. The same interface can express task dependencies in terms of CDO dependence. The details of the high-level API are out of scope for this technical specification and will be described in WP4.

4.1.5 Data Transformations

A set of programming level abstractions that are are provided primarily to support the Transformation API enable loop tiling and memory promotion. Use of these abstractions is optional.

Maestro provides layout-sensitive transformation capabilities by combining library-level CDO declarations, explicit usage of advanced framework abstractions (e.g. tiling data-structures) and additional runtime capabilities. Such functionality is available to CDOs that are not inside the management pool.

Given the wide variety of transformations required on CDOs, higher-level abstractions capture the required Scope and Context within which the CDO is operating. Not all features of Maestro are available to all programs, the subset of features depend on the qualities of the declared CDOs, handled by the Scope Object (see Section 4.2.4).

4.1.6 Resource Management

Maestro itself is planned to be usable without allocating significant resources to the Maestro core itself. It can – and in many cases will have to – be assigned resources for its own purposes, like non-volatile storage resources, or dedicated volative memory space.

User programs can make use of Maestro for resource allocation by using the `DECLARE/DISPOSE` mechanism by using the `ALLOCATE_NOW` attribute. Resources may also be allocated transparently for performance reasons or to satisfy lifetime demands requested by the user. The default is that Maestro will not allocate local resources, enabling the zero-copy paradigm, working with existing user data allocations: CDOs declarations should not require data to be copied.

4.2 Abstractions

This section defines the fundamental abstractions (PP.3.1.1.b) that are presented by Maestro. No existing abstractions are suited to the highly varied Maestro requirements, and

so it is expected that suitable abstractions will be developed in the project (PP.3.1.3.a). A limited set of abstractions underpin the Maestro Core, the most fundamental being the Core Data Object. The full list of abstractions is

- Context Object (Ct)
- Scope Object (S)
- Memory Object (M)
- System Object (Sy)
- Core Data Object (CD)

We typically refer to a CD object as a CDO, and the others using their full names. The relationships between these objects in terms of parents, children and siblings is shown in Figure 2.

4.2.1 Memory Object

A Memory Object (M) represents a set of locations that can be used for persistent and/or non-persistent storage of data, an allocation process to reserve a subset of the storage locations, an interface or set of interfaces to load and store data to some set of those locations, and (if included in the memory model) an associated benchmark program to elucidate such loading, storing and allocation.

4.2.2 System Object

A system object (S) is a set of memory objects represented as a graph. Within a system, some nodes/objects may be networked and accessible resulting in an edge between the nodes. A memory system model can query the transport rates between two memories by traversing the edge of the graph in question. Explicit symmetry between sub-trees can express parallelism. Systems can be multi-level as shown in Figure 3 which shows a typical example of distributed memory parallel memory objects, where memory objects representing DRAM, HBM and SSD levels are mirrored across N nodes.

4.2.3 Context Object

A Context object captures higher-level classifiers that tell Maestro what types of Scope Objects and CDOs are likely for this specific Context. The following high-level contexts are currently specified

1. `transformation`: loop and array based data objects as represented by the left hand side of Figure 1 and likely to be used in the Transformation API (Section 4.3.2).
2. `management`: layout insensitive data objects as represented by the right hand side of Figure 1 and likely to be used in the Data Object Management API (Section 4.3.1).

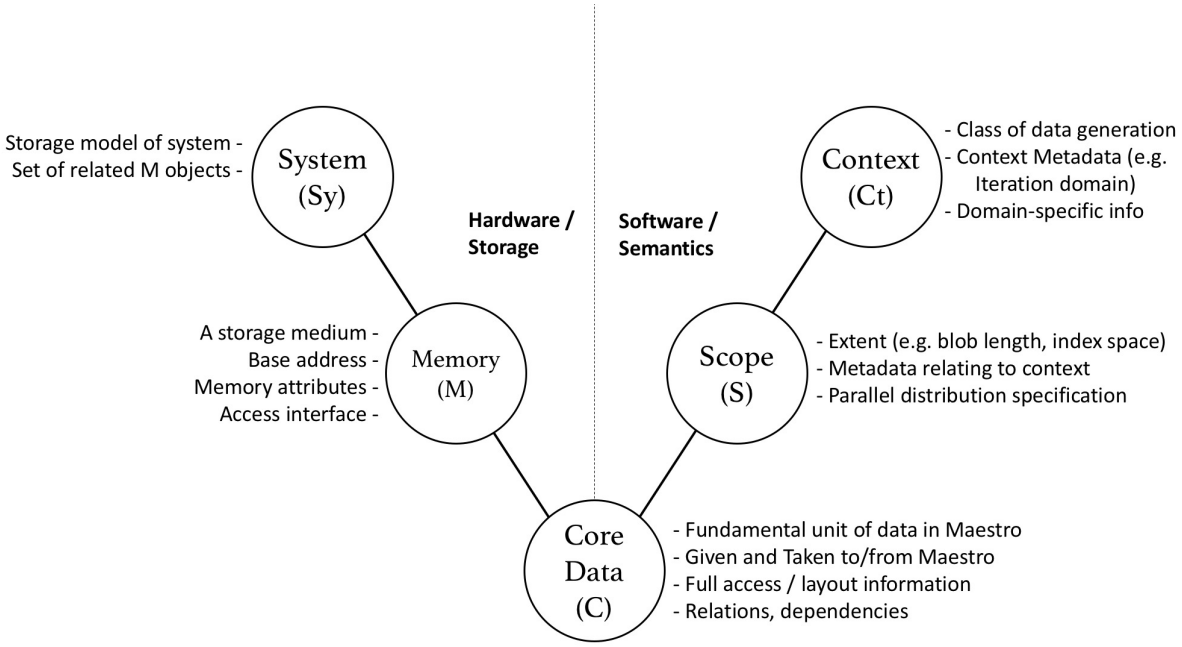

Figure 2: Object hierarchy and relations for the fundamental abstractions enumerated in Section 4.2

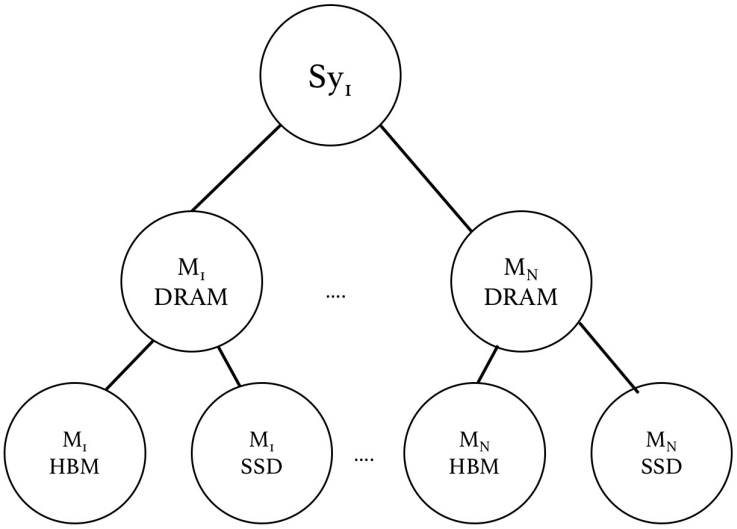

Figure 3: Example Distributed memory system

3. `hybrid`: Enabling the full functionality of the Maestro middleware for applications that may require both the Transformation API and the data object Management API

If `CONTEXT=management` or `CONTEXT=hybrid` domain-specific metadata is passed within the Context object. When `CONTEXT=hybrid` the full variety of Scope Object (see 4.2.4) attributes are available to the application, though this may come with penalties (PP.3.1.1.i).

4.2.4 Scope Object

Maestro tackles many data movement problems across the entire memory-storage hierarchy. As shown in Figure 1 of Section 1 Maestro addresses two almost-distinct sets of technical concerns, "transformations" and "data management". The Scope Object (Sec. 4.2) is where the ability to respond appropriately to the variety of conditions is enabled. Although different technical features will be activated, the usage models in either case should remain similar.

A Scope Object (S) captures any available information pertaining to the scope, size, access relations (indexing), and schedules (ordering) of the data (PP.3.1.3.c). The availability of such information depends heavily on the class of the data object, which is captured along with additional metadata in the Context object (Ct). For an opaque blob (PP.3.1.3.d) the (S) object captures the size of the object in bytes. For a 2-d MxN array the (S) object captures the index space: $\{(i, j) : 0 \leq i < M, 0 \leq j < N\}$.

The Scope Object also captures semantic information pertaining to the application's usage of data-structures such as iteration domains, access functions and a schedule. Layout information such as strides and padding dimensions is a property of the CDO, not the Scope Object. Certain actions within Maestro are enabled and disabled depending on the type of Context (PP.3.1.1.g, PP.3.1.1.h). For hybrid contexts, the full set of attributes is accessible.

Common attributes of all Scope Objects:

1. Handles to CDOs in this scope

Attributes of Scope Object when `CONTEXT=management` or `CONTEXT=hybrid`:

1. Size of the CDO
2. Relations (dependencies) between CDOs

Attributes of Scope Object when `CONTEXT=transformation` or `CONTEXT=hybrid`:

1. Symbolic constants defining the extent and shape of data
2. Index sets: representing the full index space of tensors (PP.3.2.2.f)
3. Mapping between index space dimensions and specific CDOs
4. Iteration Domain: the iteration space of a loop nest
5. Array Access functions: mappings between the iteration domain and a index space
6. Data Dependencies between statement instances
7. Schedule: an effective ordering of the iteration domain (PP.3.1.4.e)

4.2.5 Core Data Object

A core data object (CDO) is typically formed from a Scope Object and a Memory Object, and (in this case) represents real data and their physical location (if known). CDOs that do not represent real data or represent unspecified data are also legal.

The CDO combines all available information from both the hardware/storage side and the software/semantic side (PP.3.1.2.e). As the most complete understanding that the middleware can obtain about a particular data object, the CDO is how applications and Maestro communicate intentions. Two qualities of a CDO are in particular used by applications to influence what Maestro can do with the CDO.

A CDO declaration (see Section 4.3.1.1) by default does not require a copy of data.

CDOs possess two binary states that dictate how Maestro can interact with the CDO

1. `ACCESSIBLE` defines whether the contents of a CDO can be accessed via Maestro accessor (e.g. elemental/tile set/get) functions

2. `POOLED` defines whether a CDO has been given to the Maestro management pool (whereupon it is under temporary, unilateral control by Maestro), or not.

When a CDO is `POOLED`, Maestro may move, copy or transform the CDO, or may temporarily replace explicit operations on that CDO by overriding existing APIs with "Maestro-enabled" replacements, e.g. by overriding POSIX writes to a CDO.

When a CDO is `ACCESSIBLE`, then its content and structure can be queried by an application, and a set of accessor methods allow setting and retrieval of the data. This low-level accessor interface is used by a Transformation API consisting of access ordering, tiling and layout-specific optimisations.

If an application creates a POOLED CDO, the application is relinquishing control of the CDO to the middleware. Maestro will perform transactions, resolve relations and perform operations upon CDOs when they have been given to the Maestro CDO Management pool. When CDOs have not yet been given to the pool, or when the CDOs have been taken out of the pool, then Maestro will not interact with CDOs. Figure 4 shows how CDOs access and pool status are updated after transactions in the Maestro management protocol.

The Transformation API (see Section 4.3.2) cannot be accessed by CDOs for which either `ACCESSIBLE=0` or `POOLED=0`.

A CDO's attributes include pointer(s) to real data, access relations, a unique identifier, relations to other CDOs, domain-specific key-values (see Section 4.4.1).

A CDO includes a distribution specifier (PP.3.2.2.e). A distribution logically maps data elements to distinct memory spaces using a distribution scheme, or (where such a scheme cannot be defined) an explicit map. The distribution specifier can return global or local indices for a specific node given a local or global index (respectively). The distribution specifier can be used to calculate the pairs of indices that must be exchanged between any two nodes in a grid/mesh.

A CDO contains optional metadata related to the context from which is was derived, which it inherits from a Scope Object (see Section 4.2.4). This metadata may be empty. It possesses user-level attributes of a CDO: unique identifier, persistence, distribution, CRUD timestamps.

State	
ACCESSIBLE	POOLED
Chain of events:	
DECLARE	
1	0
OFFER	
0	1
WITHDRAW	
1	0
DISPOSE	

(a) Producer

State	
ACCESSIBLE	POOLED
Chain of events:	
DECLARE	
0	0
REQUIRE	
0	1
DEMAND/RETRACT	
1/0	0/0
DISPOSE	

(b) Consumer

Figure 4: CDO access and pool status during the phases of the CDO management protocol

Maestro provides rudimentary CDO resilience features in an attempt to not lower the level of data resilience experienced by an application compared to when Maestro is not enabled. Inside applications using Maestro for explicit data movement, a CDO declaration also requires a memory object and a scope object declaration. Some of these may be wildcarded to indicate that the application does not possess or does not require very precise information about them.

4.3 Interface to Maestro Core

4.3.1 CDO Management API

Figure 1 from Section 1 showed the variety of data issues addressed by Maestro. This section is concerned with the right-hand extreme of that landscape, largely layout-independent data objects (PP.3.1.1.h). From this perspective, a CDO is simply an abstract object, uniquely identified by a name. In the context of multiple independent applications referring to the same CDO it has read-only characteristics (while it is in the Maestro management pool, see below).

1. A 4-step protocol, shown graphically in Figure 5 specifies the way that an application and Maestro share ownership of data objects. Broadly speaking, 'Give-Take' semantics describe the giving of data objects to Maestro from application and the taking of data objects back from Maestro. To accommodate the necessary differentiation between data producer and data consumer use cases, "give" and "take" have bilateral counterparts.

2. We say that an application is a CDO producer if it injects a CDO into the Maestro system (gives a CDO to the management pool). An application that wishes to access an existing CDO is a CDO consumer (takes from the Management pool).

3. Applications may be both producer and consumer to leverage features of the Maestro core.

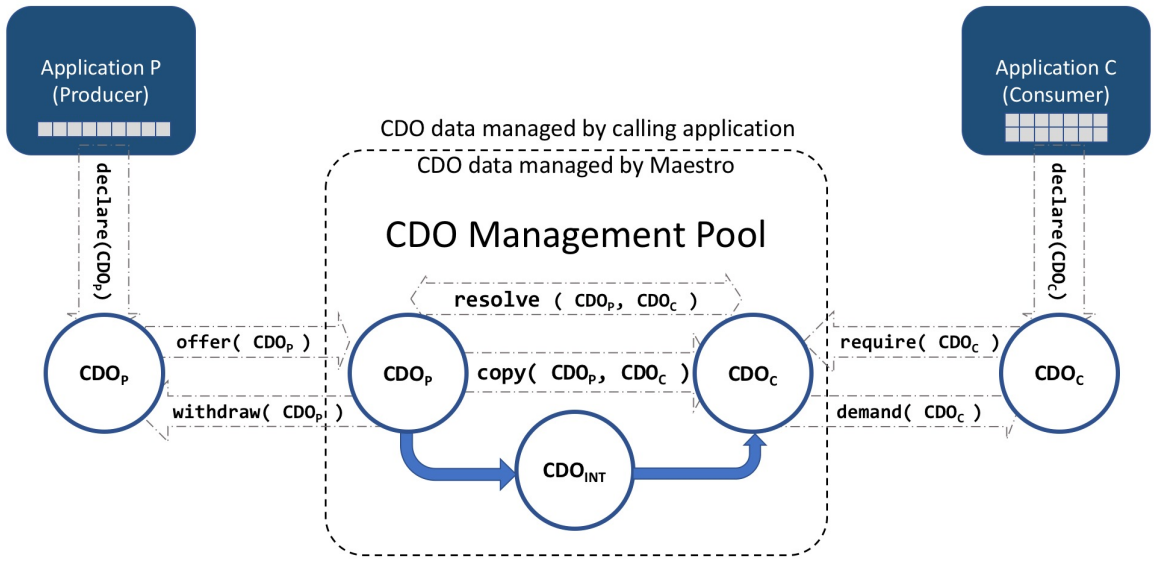

Figure 5: CDO Management API. Detail of `resolve` and `copy` deferred until D3.2

4. The overall give-take API is composed of `OFFER`/`REQUIRE`, `WITHDRAW`/`DEMAND`/`RETRACT`, and `DISPOSE`. They represent 4 distinct steps in the lifetime of a CDO as detailed below, namely

   1) CDO declaration,
   2) CDO pool injection,
   3) CDO pool retraction,
   4) CDO disposal.

5. In all of the operations, CDOs are referred to on the application side by an application-defined name at `DECLARE` time, and then referenced by a *handle* that corresponds to a (Maestro-internal) object identifier. Maestro can match CDO names across applications directly, but typically will be instructed to consider certain names equivalent, e.g., the name of a CDO that is produced by one application "CDO_P" and a (different) name for an input to another application "CDO_C". In such a case we say that the two (different) CDOs are *resolved* inside the Maestro pool.

6. No CDO handle may be used twice in any of the operations of this protocol, in particular

   (a) No CDO handle may occur in an `OFFER`ed twice
   (b) No CDO handle can be used in both `DEMAND` and `OFFER`.

   However, handover of resources from one CDO to the other may be used to, e.g., `OFFER` data that was obtained by `DEMAND`.

4.3.1.1 Declare CDO (`DECLARE`) `DECLARE` allows an application to describe a CDO to Maestro.

1. The application obtains a handle that will be needed for any future communication about this object.
2. Maestro can, depending on the object attributes used in the declaration, allocate resources, prepare and plan for necessary transfers, schedule other workflow components, etc.

Details about CDO naming are given in Section 4.4.4, and attributes are described in Section 4.4.1.

4.3.1.2 Give to management pool (`OFFER`, `REQUIRE`) The two GIVE variants take CDO definitions from Maestro to the CDO Management Pool.

1. CDO producer: `OFFER`
2. CDO consumer: `REQUIRE`
3. After this operation the CDO content may not be accessed by the application. It is entirely up to Maestro where the CDO content resides or even whether its content is consistent with any previous or future state.
4. Maestro can, depending on the object attributes used in the declaration, allocate resources, prepare and plan for necessary transfers, schedule other workflow components, etc. It can even use the CDO's storage allocation for other purposes.

4.3.1.3 Take from management pool (`WITHDRAW`, `DEMAND`/ `RETRACT`) Take is the moment at which CDO ownership is transferred (back) to the application. This operation is blocking: Maestro may choose to delay all operations until this point (permitting lazy CDO handling semantics).

1. CDO producer: `WITHDRAW`
2. CDO consumer: `DEMAND`, `RETRACT`
3. If any part of the workflow requires the CDO that gets `WITHDRAW`n Maestro may either block or decide to create a duplicate instance of the CDO suitable for the expected consumer.
4. In the case of distributed CDOs, an operation to take from management pool do not imply a collective synchronization point across distinct memory spaces.
5. After this operation the CDO content will be in the state required by the CDO declaration and from arguments to `DEMAND`.
6. Maestro will still be aware of the existence of the CDO, but it will no longer access the CDO in any way - it is completely in the hands of the application.

7. If no compatible `OFFER` for a `REQUIRE`d or `DEMAND`ed CDO is visible to the Maestro framework, a `DECLARE`-time attributes determines if the `REQUIRE` will fail, block, or succeed, and whether the `DEMAND` will fail or block.

8. `RETRACT` serves to cancel a previous `REQUIRE`, before the `DEMAND` for the CDO happens. This is is useful for an application that wants to `REQUIRE` multiple CDOs, but after some of them arrive is ready to proceed without `DEMANDING` all of them.

After a the completion of either of these operations, the CDO is `ACCESSIBLE` again.

After a `WITHDRAW` or `DISPOSE/RETRACT` a `DECLARE` for a new CDO can mention the 'old' CDO as resource to potentially reuse memory. This will mark the old CDO as 'dying' by making it no longer `ACCESSIBLE`. In particular it can no longer be `OFFER`ed. The `DISPOSE` will be aware of the transferred resource ownership and potentially does not have to recycle its resources.

4.3.1.4 Delete CDO (`DISPOSE`)

1. This is the inverse of the `DECLARE` operation. It indicates the end of the CDO's lifetime. If resources were provided by Maestro for the CDO they may be deallocated.

4.3.1.5 Using CDO Management for Memory Promotion The CDO management API can perform CDO-level silent memory promotion satisfying (PP.3.1.4.c) at the data object level via *memory wildcards*. A memory wildcard informs Maestro that it may select the appropriate memory location for a CDO. Maestro will choose to move data objects into available memory spaces when CDOs are POOLED.

Note that memory wildcarding requires that Maestro is enabled to allocate its own resources internally (See Section 4.4.1) and only memory levels for which a memory system model is available will be evaluated by Maestro (Section 4.1.2).

4.3.2 Transformation API

In Section 1, Figure 1 showed the scope of the data object functionality required by Maestro. The parts of the Maestro API called the Transformation API are concerned with the left-hand extreme (PP.3.1.1.g).

The Transformation API is only available to CDOs for which `ACCESSIBLE=1` and `POOLED=0` i.e. those which are declared, are not in the pool and for which real data is existent. Attempts to use the Transformation API will return errors if `ACCESSIBLE=0` or `POOLED=1`.

Since the Transformation API is applied to mutable data, it does not return new CDOs as outputs. Transformations operate directly on either a Scope Object or a CDO.

The following transformations can be applied to Scope Objects when `CONTEXT=transformation` and are effectively loop-nest transformations.

1. Permute: change the effective ordering of outer loops.

2. Split: decompose a loop-nest into separate loops

3. Fuse: Combine two or more loops

4. Explicit Tile: Decompose an iteration domain into sub-domains that are traversed interdependently, with an ordering defined within each tiled-domain (PP.3.1.4.d)

5. Reschedule: Modify the schedule of the Scope Object to another (legal) schedule.

The following transformations are applied to CDOs and constitute layout transformations of the data represented by the CDO (PP.3.2.2.f).

1. Transposition: matrix or tensor dense array transposition,

2. Splitting: decomposing the index space,

3. Align: align the specified data to a particular hardware boundary,

4. Pad: append one or more dimensions of data with additional memory for efficient striding,

5. Reshape: apply arbitrary remapping of data to memory locations,

6. Redistribute: modify the distribution of a distributed CDO.

A higher-level Transformation API is expected to be built on top of the above transformations to, e.g., align to cachesize boundaries, pad arrays to a maximum size, etc.

A higher-level User Transformation API, in which the user would need to provide a certain set of callback functions, is expected to be proposed. For the callback registration to be effective, the user provided functions would need to fit guidelines imposed by the mentioned API.

4.3.2.1 Tiling abstraction A tiling abstraction is presented, satisfying PP.3.1.4.d. Such an abstraction allows the user to program to an explicit tile data-structure, the location of which will be modified by Maestro rather than the user. The tiling abstraction is a set of related abstractions, including:

1. A `PARALLEL_FOR` abstraction

2. CDO Indexing abstraction

3. Tile handler

4. A tiling runtime library

Some of the underlying functionality of the tiling abstraction is enabled via the CDO accessors (Section 4.3.4). Since the development of tiling abstractions and associated runtime libraries may lay out-of-scope for the Maestro project, the project will integrate the results of the EPiGRAM-HS project[2]

[2]https://epigram-hs.eu/

4.3.3 Hybrid API

As illustrated in Section 1, Figure 1, the intersection of the spaces encompassed by the CDO Management API and the Transformation API is non-empty. However, certain overheads and limitations accompany both cases - the middleware solution may be too heavyweight for some users and the complications of index spaces and domains are out of scope for Maestro users interested only in the data management aspects. For those users with a clear interest in both layout-insensitive management, and layout-sensitive transformation, the hybrid API is available and in this case the Scope Object is populated with the comprehensive set of all attributes (see Section 4.2.4). The most clear use-case of the hybrid API is to request 'transformations on `REQUEST`'.

When `CONTEXT=hybrid REQUEST` will take a `LAYOUT` specifier which defines a precise mapping between the CDO's index space and the physical addresses of a Memory object, satisyfing PP.3.1.4.b. That layout is independent to the layout of the CDO that was `OFFER`ed to Maestro, and the layout transformation is handled by Maestro.

4.3.4 CDO Accessors and Queries

4.3.4.1 `CDO_[ELEMENT,SLICE,TILE]_GET` When `ACCESSIBLE=1` returns a specific element, slice or tile of a CDO's stored data. The data may be located in DRAM, or any other memory level that is know to Maestro. If `ACCESSIBLE=0`, an error is returned.

4.3.4.2 `CDO_[ELEMENT,SLICE,TILE]_SET` When `ACCESSIBLE=1` stores data to the element, slice or tile locations specified by the indexing arguments. The data may be located in DRAM, or any other memory level that is know to Maestro. If `ACCESSIBLE=0` an error is returned.

4.3.4.3 `CDO_NEXT_TILE` When `ACCESSIBLE=1` returns the next tile indices for a CDO according to a tile scheduling that is defined at tile initialisation time. The data may be located in DRAM, or any other memory level that is know to Maestro. If `ACCESSIBLE=0` an error is returned.

4.3.4.4 CDO Convenience API An explicit CDO handling API is presented for applications that want to use Maestro to apply data layout or data location changes to a CDO.

1. CDO movement from one Memory Space to another: The memory space designation is part of the CDO declaration, hence the following sequence (inside one application) can be used to force Maestro to perform the movement:

```
CDO_HANDLE1=DECLARE(CDO_DESIG, MS1, ATT1)
CDO_HANDLE2=DECLARE(CDO_DESIG, MS2, ATT2)
OFFER(CDO_HANDLE1)
REQUIRE(CDO_HANDLE2)
DEMAND(CDO_HANDLE2)
WITHDRAW(CDO_HANDLE1)
```

This could be captured in a single

```
RELOCATE(CDO_DESIG, ATT1, MS1, ATT2, MS2)
```

function

2. High-level CDO transformation: Maestro declarations will provide attributes beyond Memory Space specifications, capturing some built-in transformations like compression. These can be wrapped just like `RELOCATE` for convenience.

   ```
   TRANSFORM(CDO_DESIG,ATT1,MS1, ATT2, MS2)
   ```

   `RELOCATE` then is simply an identity transformation.

3. Explicit CDO coupling

   (a) App1

   ```
   CDO_HANDLE=DECLARE(CDO_DESIG1)
   OFFER(CDO_HANDLE)
   WITHDRAW(CDO_HANDLE)
   DISPOSE(CDO_HANDLE)
   ```

   (b) App2

   ```
   CDO_HANDLE=DECLARE(CDO_DESIG2)
   REQUIRE(CDO_HANDLE)
   DEMAND(CDO_HANDLE)
   DISPOSE(CDO_HANDLE)
   ```

   where either the `CDO_DESIG1,2` designators are the same, or the Maestro framework is otherwise instructed (e.g., by a workflow specification, compiler aliasing analysis, ...) to treat the two CDOs as equivalent.
   These can be simplified to

   ```
   PROVIDE(CDO_DESIG1, ATT1)
   ```

   and

   ```
   DEMAND(CDO_DESIG2,ATT2)
   ```

   for explicit lock-step synchronization of the applications on this CDO.

4.4 CDO Qualities

We detailed in the previous section fundamental interactions with the Maestro core for data object management and data transformations. The current section focuses on how Maestro handles more semantics. Additional qualities allowing a closer control (attributes, lifetime) on CDOs are described, and also qualities that ease interaction at a higher level (groups, naming). In each of the following sections, the succinct proposal is backed by a longer technical discussion in Appendix B.

4.4.1 CDO Attributes

CDO attributes are properties of a CDO that are attached to the CDO at `DECLARE` time (PP.3.1.3.d). Every CDO can have arbitrary key-value data added to it ("user level attributes") which the Maestro core will not inspect.

The following minimal set of attributes are required.

1. A CDO name. Rules for CDO names are discussed in more detail in Setion 4.4.4.
2. An allocation specification, to distinguish whether the application will provide storage, or whether Maestro resource management is desired.
3. An `ALLOCATE_NOW` flag. This indicates that underlying storage must be available immediately at the end of the `DECLARE` step. It is required if Maestro resource management is desired and the application is not in the consumer position of a workflow (where allocation can be delayed until `DEMAND`) to permit the CDO to be `ACCESSIBLE` after the declaration. Otherwise it is optional.
4. An (optional) lifetime specifier. Can be used to indicate that the CDO must be available for a certain time or until a certain deadline. For details see Section 4.4.2.
5. An (optional) redundance specifier. Can be used to indicate that Maestro should keep duplicate copies *while the CDO is in the management pool*, see Section 4.3.1.2.

Inside applications using Maestro for explicit data movement or using Maestro accessors for automatic memory layer promotion, a declaration also needs to include a memory object and a scope object (PP.3.1.2.d). CDOs operate with different *levels* of available information, which is an internal classification to Maestro and allows different levels of optimisation.

1. A *level-0 CDO* is known by name only, is not suitable for data movement.
2. A *level-1 CDO* contains a memory object and a scope object, allowing reasoning about actual storage location(s) and size(s), and thus permits reasoning about data movement costs inside the memory system.
3. A *level-2 CDO* has extra information useful for efficient operation of Maestro, e.g. relaxed consistency requirements.

For a longer discussion of these topics see Appendix B.1.

4.4.2 CDO Lifetime

CDO lifetime is always bounded by the lifetime of the Maestro workflow in which the CDO appears. Nested Maestro-enabled workflows are conceivable, in which the outer CDOs survive all CDOs appearing only inside the inner workflow.

Three lifetime handling modes can be configured by the user code:

1. Hard deadlines: If Maestro fails to satisfy a lifetime annotation it will return errors.

2. Soft deadlines: Best effort is made, possibly warnings are printed/logged, but Maestro will not generate errors.

3. Ignored deadlines: Lifetime values are only tracked, not respected.

For a longer discussion of these topics see Appendix B.2.

4.4.3 Groups of CDOs

CDO groups (sets) are supported by Maestro Core. One use-case of groups is the splitting of a CDO by the middleware into multiple related CDOs to e.g. fit into DRAM (PP.3.1.3.f).

A CDO group behaves similarly to a CDO, except that its resources are determined by the CDOs it contains. Many valid CDO operations can be presented as group operations (details deferred until D3.2).

1. CDO groups satisfy the same naming rules as CDOs.

2. A CDO can be an element in multiple CDO groups.

3. CDO groups possess attributes

4. A CDO group's cardinality can be queried.

5. CDO groups support iteration over the elements after a `REQUIRE` operation.

6. CDO groups can only be `RETRACT`ed as a whole, and only before any `DEMAND` operation.

7. A CDO group can be created so that its elements refer to parts (elements, tiles, ...) of one or more CDOs.

CDO groups are conceptually orthogonal to namespaces or having multiple pools. They can, however, be combined to, for instance, put all CDOs of a group into a common namespace, and perform allocation of them all in the same pool, or each in a separate pool.

For a longer discussion of these topics see Appendix B.3.

4.4.4 CDO Names and Namespace Support

The minimal declaration of a CDO consists of specifying only a name which is suitable for the application(s) using it to identify the object. In many cases a name will be sufficient for a workflow definition to define input- and output-CDOs for the components, and to define equivalence between different names.

1. Maestro core considers CDO names as opaque octet sequences.

2. It is in the responsibility of higher levels of the Maestro architecture to impose lexical or tokenization rules on these strings to explicitly support, for instance, index notations (PP.3.2.2.a).

3. Namespaces will be supported at the CDO naming level; nesting is conceivable.
4. We propose to use '/' as namespace separator (path-like)
5. Wildcard search for CDOs across namespaces could be supported
6. An application's object naming scheme can be used also within Maestro as long as that scheme adheres to the Maestro's own naming conventions (PP.3.2.2.b).
7. Namespaces conceptually are orthogonal to the possibility of splitting the Maestro pool
8. Internal to the Maestro core each CDO has a unique identifier (CDOID), which can be queried but not influenced by the application.

   For a longer discussion of these topics see Appendix B.4.

4.5 Resilience Considerations

Maestro does not implicitly introduce any additional failure resilience beyond the resilience level already achieved by the application.

Given the nature of Maestro as a middleware, the possibility of Maestro crashing internally must be considered a catastrophic event, similar to an MPI or scheduler failure.

At this time we do not specify whether objects that are in the Maestro pool may be recovered after abnormal task or workflow termination.

An application that does not use Maestro for resource allocation, but only `DECLARE`s CDOs that reference application-controlled resources will be influenced by Maestro failures only by errors returned from Maestro API. Applications that are using Maestro-provided resources must rely on the resource management being stable enough for their purposes.

Failures detected by the application may happen

- at `DECLARE`: Recoverable, but Maestro-based transport of that CDO will probably not be possible. The application can implement a fallback strategy.
- at `OFFER`: Recoverable, but Maestro-based transport of that CDO will probably not be possible. The application can employ the same fallback strategy as for failing `DECLARE`.
- at `WITHDRAW`: Risk of being unrecoverable. We propose to add a `Precious` attribute that can be specified at `DECLARE` time which will ensure that the CDO can always be `WITHDRAW`n. This may incur cost, e.g., by forcing Maestro to only work on a safety copy (which would be created at `OFFER` time).
- at `REQUIRE`: Recoverable, but Maestro-based transport of that CDO will probably not be possible. The application can implement a fallback strategy.
- at `DEMAND`: Recoverable, but Maestro-based transport of that CDO will probably not be possible. The application can employ the same fallback strategy as for failing `REQUIRE`.

- at `DISPOSE`: (Small) risk of Maestro-internal resource leakage. Can be ignored by the application for practical purposes.

API functions will document the error codes and specify guarantees made on success or failure.

The Maestro core does not offer archiving features. As such, unavailability of a CDO will have to be dealt with by the application. Minimum object lifetimes (Section 4.4.2) offer some protection against that, if the storage inside the Maestro resources provides an acceptable safety level. This does not preclude integration of features for archiving CDOs in the future, e.g., through attributes modifying `PROVIDE` and `REQUIRE` behavior. Issues introduced by interception of applications I/O need further investigation.

For a list of predicted failure scenarios in Maestro, see B.5.

A Response to feedback

1. "Consumer must know the size of the data prior to receiving" is an unnecessary assumption (ECMWF)
   Proposal: The `DECLARE` Section 4.3.1.1 has been extended to clarify this (more discussion can be found in Appendix B.1).
   A CDO declaration appears in 3 contexts: In a producer, in a consumer, in an (external textual or programmatic) workflow definition. In the workflow definition a size annotation should not be required (but optionally possible). In the producer situation the CDO size must be specified in the DECLARE phase: a CDO will either be formed from existing data in the application, or by requesting space allocation from Maestro, so in both cases this is not difficult. In the receiving situation, if space allocation is performed by the application the size must be specified to permit Maestro to check whether the incoming CDO will fit the allocated space. If the CDO is declared so that Maestro is used to perform the allocation, size does not need to be specified. It is expected that this is the normal way for consuming applications: It permits Maestro to choose the most suitable memory layer for the incoming CDO and avoid copying it into an existing user allocation.

2. Data much larger than memory (ECMWF)
   Proposal: Maestro can be used to allocate resources (Sec. 4.3.1.1, 4.4.1, more details in B.1) More importantly, the proposed blocking transformation (part of management API) in Section 4.4.3 and more discussion in B.3.

3. Consumer failure may imply loss of CDO (ECMWF)
   Proposal: Resilience properties are discussed in (Section 4.5); CDO lifetime in (Section 4.4.2) and redundance attributes in (Section 4.4.1, more discussion in B.1)

4. Handling names of CDOs, make pool splittable by resource groups, possibly associated with namespaces (or not) (ECMWF)
   Proposal: Multiple CDO pools can be created, multiple namespaces will be supported
   See Section 4.4.4, more discussion in B.4.

5. (ECMWF, CEA)
   Proposal: Allow data lifetime information to be included in the attributes at CDO creation time, define lifetime for object other than implicit lifetime induced by workflow
   Made explicit in Section 4.4.1 and 4.4.2. (more discussion in B.1)

6. CDO sets/groups: naming, referencing ... Allow consumers to access data without knowing explicitly what it will be in advance (i.e. range queries, all that matches, etc.) (ECMWF)
   Proposal: Allow consumers to access data without knowing explicitly what it will be in advance (i.e. range queries, all that matches, etc.). This is supported by CDO name queries Sec. 4.4.4 (more discussion in B.4) and explicit CDO groups Sec. 4.4.3 (more discussion in B.3)

7. OASIS accommodates data layout differences between models such as row-major order or column-major order (Juelich)
   Proposal: CDO transformations, Sec 4.3.2; basic information on OASIS-related use case in 3.2.1.2

8. Parallel interpolation... should happen within Maestro or left to the application? (Juelich)
   Proposal: CDO transformations, possibly with user-provided operations, Sec 4.3.2

9. Most of the CDO data can be cached and reused for subsequent iterations of the time evolution loop (Juelich)
   Proposal: Before `DISPOSE` resources can be transferred to a new CDO efficiently, as discussed in Section 4.3.1.3

B Technical Discussions

B.1 Discussion on CDO Attributes

CDO attributes are properties of a CDO that are attached to the CDO at `DECLARE time` (PP.3.1.3.d).

This data is bound to the CDO itself; equivalence between CDOs does not propagate key-value data among them, but there may be functionality to efficiently inspect the user level attributes of equivalent CDOs.

The full set of possible attributes used for CDO declaration is still not decided. We propose to make attributes an extensible functionality based on a key-value semantic, with keys having a structured scheme that allows essential functionality to be defined and restricted, while permitting future additions. A possible scheme could be a dot-separated key style like `maestro.core.allocation_specification`, possibly even reverse-domain-name prefixing like in java namespaces.

Every CDO can have arbitrary key-value data added to it (“user level attributes”) which the Maestro core will not inspect.

At this time we envision the following minimal set of attributes being required.

1. A CDO name. Rules for CDO names are discussed in more detail in Section 4.4.4.

2. An allocation specification. Needed to distinguish whether the application will provide storage, or whether Maestro resource management is desired.

3. An `ALLOCATE_NOW` flag. This indicates that underlying storage must be available immediately at the end of the `DECLARE` step. It is required if Maestro resource management is desired and the application is not in the consumer position of a workflow (where allocation can be delayed until `DEMAND`) to permit the CDO to be `ACCESSIBLE` after the declaration. Otherwise it is optional.

4. An (optional) lifetime specifier. Can be used to indicate that the CDO must be available for a certain time or until a certain deadline. For details see Section 4.4.2.

5. An (optional) redundancy specifier. Can be used to indicate that Maestro should keep duplicate copies *while the CDO is in the management pool*, see Section 4.3.1.2.

6. An (optional) `NEVER_POOLED` flag. This indicates that the CDO will be never be used in `OFFER/DEMAND`, which potentially permits extra performance optimizations.

The CDO size is not necessarily an attribute per se. It is known and can be queried by application code whenver the CDO is `ACCESSIBLE`. It is known to the Maestro core after it has been first `ACCESSIBLE` to some application. Of course it may be known to the application for other reasons (user-provided underlying resources, out-of-band communication about it) beyond the scope of the Maestro core. A workflow management or workflow observation component of Maestro will not declare `ACCESSIBLE` CDOs itself, so never needs to know the size before observing that it entered the pool, at which time it has a size known to the Maestro core.

A CDO that is only known by name is called a "*level-0 CDO*". It is not suitable for data movement. If, however, equivalence has been established with another CDO, it behaves like a reference to the other CDO, and thus will obtain proxy semantics for it. Equivalence of CDOs is a property that applications explicitly construct. It is thus in the hand of applications to ensure consistency between them, e.g., if two CDOs are created with different data content, but declared to be equivalent, then Maestro can choose either of them to satisfy requests for the other. A warning may be issued for cases that commonly would be erroneous.

Inside applications using Maestro for explicit data movement or using Maestro accessors for automatic memory layer promotion a declaration also needs to include a memory object and a scope object (PP.3.1.2.d). Some of these may be wildcarded to indicate that the application does not possess or does not require very precise information about them.

A CDO is called "*level-1 CDO"* if the following information can be queried (by application, and by the Maestro core): a memory object and a scope object. These descriptors will allow reasoning about actual storage location(s) and size(s), and thus permit reasoning about data movement costs inside the Memory system.

CDOs have two distinct phases in their lifetime: outside the Maestro pool and inside the Maestro pool (see Section 4.4.2, and enter and leave the pool in two distinct roles: for data contribution into or data absorption from the pool. A CDO that has extra information about requirements at the time of entering or leaving the pool (to be defined elsewhere), or extra information that is useful for efficient operation of Maestro at these steps is called "*level-2 CDO*". Possible examples include producer-side annotations

Example	Maestro action
Relative time specification	
"I cannot provide resources longer than X minutes"	CDO may need to be copied to Maestro-controlled storage to free originator before deadline.
"Please ensure CDO availability for at least X hours"	CDO may need to be copied, and even kept alive beyond `WITHDRAW` or `DEMAND`.
Absolute time specification	
"CDO expires at midnight"	Advises Maestro to try to schedule tasks to make sure the deadline can be met. (Produced CDO should have consumers scheduled before, Consumed CDO should have producer scheduled before deadline)

Table 2: CDO lifetime specifications

guaranteeing certain qualities like "Will not need to be consistent when returning from pool", or consumer-side annotations like "will obtain for limited-region R/O access only, not create a duplicate for modification".

In addition to attributes explicitly added by applications when declaring the CDO, there are automatic read-only attributes constructed by the Maestro core that are exposed to these applications, like timestamps, the internal object identifier, possibly the set of equivalent CDOs.

B.2 Discussion on CDO Lifetime

CDO lifetime is always bounded by the lifetime of the Maestro workflow in which the CDO appears. Nested Maestro-enabled workflows are conceivable, in which the outer CDOs survive all CDOs appearing only inside the inner workflow. Still, lifetime annotations are useful. Time specifications may either be relative or absolute; absolute time specifications probably have a stronger influence on Maestro's task scheduling. Table 2 shows examples and resulting Maestro behaviors.

We envision 3 lifetime handling modes that can be configured by the user code:

1. Hard deadlines: If Maestro fails to satisfy a lifetime annotation it will return errors. Possibly a polling mechanism is needed for user code to inquire whether an error has happened, since user code may spend long times without calling Maestro functions.

2. Soft deadlines: Best effort is made, possibly warnings are printed/logged, but Maestro will not generate errors.

3. Ignored deadlines: Lifetime values are only tracked, not respected. This is useful for tracing regular workflow's behavior, in order to design appropriate lifetime requirements (which may require workflow or resource allocation changes).

B.3 Discussion on Groups of CDOs

The Maestro core API is focused on providing a powerful API for handling individual CDOs. It is envisioned that higher levels of the Maestro middleware will free the user from this low level of managing CDOs if desired. To this end, CDO groups are supported by the core API.

At this time we see four main scenarios where CDO groups will arise:

1. Queries for CDOs by name yielding multiple results,
2. Handling of many similar CDOs, setting attributes similarly for all of them,
3. Iteration over multiple CDOs (in particular without ordering of objects).
4. Splitting of a CDO by the middleware into multiple related CDOs (PP.3.1.3.f), e.g., memory-blocked, as required by certain transformations (Sec. 4.3.2).

The Maestro core will hence provide CDO groups as a type. A CDO group behaves similar to a CDO in many ways, except that its resources are determined by the CDOs it contains. It remains to be decided whether the CDO Management API (Sec 4.3.1) transparently supports CDO groups, or whether its functionality will be duplicated under different API functions for CDO groups.

CDO groups satisfy the same naming rules as CDOs. The names of elements of a CDO group does not have to be related to the name of the containing set: A CDO can be an element in multiple CDO groups.

A CDO can be an element in multiple CDO groups.

CDO groups possess attributes; in particular CDO group lifetime attributes are supported by making them extend over all elements. CDO transformations do not apply to CDO groups, but basic set operations (adjoin, union, intersection, subtraction) are supported.

CDO groups cardinality can be queried. It may be possible to support CDO groups with sizes that are not statically determined. Whether a cardinality query on such a set is blocking until the actual size is definite, or whether a reply indicating 'uncertain' can be supported remains to be determined. The size is definite at least after a `OFFER` on the set has been performed. To support giving CDOs of a CDO group to the pool before the group is complete a separate `OFFER_GROWABLE` may be supported, which can be called multiple times on the same CDO group, to indicate that all elements currently in the group are to be `OFFER`ed, but the group will continue to grow. In this case it has a current (minimum) size, but may become larger.

CDO groups support iteration over the elements after a `REQUIRE` operation. There are two ways of iterating: during `DEMAND` or after a complete `DEMAND`.

1. Iteration during `DEMAND` amounts to a `DEMAND_ONE` operation that permits Maestro to deliver some member of the CDO group together with an appropriate termination check.
2. Iteration after `DEMAND` is equivalent to performing `DEMAND_ONE` on all set elements, and then iterating using separate `CDO_GROUP_EMPTY` predicate and `CDO_GROUP_NEXT_CDO` accessor.

Element-wise `DEMAND` from the same CDO group from multiple applications requires consistent `DECLARE`s of the CDO group on all consumers with an attribute of 'will-only-demand-some', or a `REQUIRE` operation indicating that only `DEMAND_ONE` operations will be performed. Otherwise Maestro will need to provide the entire CDO group to each consumer, and each consumer will eventually need to `DEMAND` all elements of it.

CDO groups can only be `RETRACT`ed as a whole, and only before any `DEMAND` or `DEMAND_ONE` operation. After a `RETRACT` in the situation where multiple consumers have declared the CDO group to be used for concurrent `DEMAND_ONE` operations the other applications will eventually receive the CDO elements that potentially would have been received by the `RETRACT`ing application.

CDO groups have the potential to improve application coupling performance if unordered consumption of the set elements on the consumer(s) is possible, but may also decrease the insight into static data dependencies.

A CDO group can be created so that its elements refer to parts (elements, tiles, ...) of one or more CDOs. This is only permitted in such a way that the elements correspond to entities that are accessible using Maestro accessors on the original CDOs (it does not create 'intermediate/anonymous CDOs' for the group elements). It does not open an avenue to define nested CDOs, or CDOs that represent a CDO group. Access to the elements will need a set of accessors similar to those specified in Section 4.3.4, taking the CDO group and the need to identify the underlying CDO into account.
A simple yet powerful case of this is `CREATE_CDO_GROUP_FROM_CDO` which, given appropriate parameters decomposes a single CDO into a group of elements suitable for `REQUEST` and `CDO_[ELEMENT,TILE]_GET` operations. This permits, for instance, piecewise consumer access to a CDO which is too big to fit into any appropriate locally accessible memory layers (PP.3.1.3.f).

CDO groups are conceptually orthogonal to namespaces or having multiple pools. They can, however, be combined to, for instance, put all CDOs of a group into a common namespace, and perform allocation of them all in the same pool, or each in a separate pool.

B.4 Discussion on CDO Names and Namespace Support

The minimal declaration of a CDO consists of specifying only a name which is suitable for the application(s) using it to identify the object. In many cases a name will be sufficient for a workflow definition to define input- and output-CDOs for the components, and to define equivalence between different names.

Fundamentally, the Maestro core considers CDO names as opaque octet sequences. We will assume that names are (printable, UTF-8) strings. As such they can be matched by string matching techniques (prefix, substring, suffix, regex search). This does not preclude the application layer to use structured names for grouping related CDOs by name, as long as the structuring can be represented in string form (e.g., some indexing notation, permanent identifiers like PIDs, etc.). It is in the responsibility of higher levels of the Maestro architecture to impose lexical or tokenization rules on these strings to explicitly support, for instance, index notations. Declarations using the same printable name will lead to CDOs that Maestro considers equivalent. CDOs with different names can be explicitly made equivalent by applications, e.g., workflow managers.

The exception to the opacity is a limitation imposed by namespace support: Namespaces will be supported at the CDO naming level; nesting is conceivable.

We propose to use '/' as namespace separator (path-like), which excludes '/' from the legal characters in CDO names without namespace qualification. A notation that is compatible with RFC8141 (Uniform Resource Names) naming will be described, but users are not expected to use URN-syntax CDO names. Whether namespaces are implemented by having different CDO dictionaries or just at the stage of translating (string) names to the internal unique ID will be determined when the feasibility of performing this translation locally (without global dictionary lookups) can be estimated and the ID-collision resolution protocol is decided.

Wildcard search for CDOs across namespaces will need to be a query to the global CDO registry, but could reasonably efficient on that.

Namespaces conceptually are orthogonal to the possibility of splitting the Maestro pool, in particular the set of resources assigned to the Maestro middleware (In contrast to those handed to the pool referencing application-side resource allocations): It is conceivable to have the Maestro-controlled resources split into multiple pools. Enforcing a mapping of the CDOs of a certain namespace to a specific pool is an independent concern. It can be supported by appropriate attributes to the resource request for a CDO declaration requesting Maestro resources.

Internal to the Maestro core each CDO has a unique identifier (CDOID; a scheme to generate these based on UUID-v5 in network byte order seems a suitable choice), which can be queried but not influenced by the application. If the Maestro core decides to create copies internally it may create "anonymous" CDOs with different CDOIDs, but they will not be visible to the application.

The Maestro core will possess an introspection capability that permits higher levels of Maestro to reason about currently declared names and declare names. It is not expected that user code will use this interface.

B.5 Failure Scenarios of Maestro Middleware

B.5.1 Shutdown of a node

Clean node shutdown will send a termination signal to the job. Maestro will attempt to react to the signal appropriately; user code may want to notify Maestro (through a provided notification function) if it does its own signal handling. A node being shut down should trigger a 'job is failing' situation; that will be handled by Maestro as specified elsewhere. Note that even a controlled node shutdown may lead to catastrophic failure of some software system component coupling it to other nodes (e.g., MPI), leading to entire jobs being terminated due to such dependencies. A node being shut down may trigger data movement to preserve (parts of) CDOs currently in the pool and occupying resource on the node. A node shutdown by normal system APIs may not leave enough time for any of the above. Maestro may at a later time offer an explicit 'decommission node' API.

B.5.2 Application crash

In the best case an application crash may simply result in the associated task terminating unsuccessfully.

In general it may happen that (parts of) a CDO (if it uses resources associated with the crashing process) become unavailable. This may trigger unsuccessful termination of other tasks that require access to the CDO (in particular, a producer that `WITHDRAW`s such a broken CDO). This may cause the Maestro workflow to retry executing tasks.

It is conceivable that certain annotations to a CDO ensure that Maestro keeps a copy of the CDO (for the duration of the workflow, or a specified time) that is safe from single/multiple such crashes. The moment to ensure this safety copy has been done is when `OFFER` returns to the calling application.

B.5.3 A task is killed by an Out-of-Memory (OOM) condition

This is considered an error that is caused by user error: either due to application error, or due to inappropriate resource assignment to Maestro.

The current specification treats it as a node failure: Random other OOM events may happen, so the node is considered to have failed.

B.5.4 Node failure

It may be possible to decommission the node on which it happened from the Maestro middleware, depending on the intra-middleware communication method used and/or a heartbeat protocol between its components.

Sudden node failure may result in (parts of) CDOs no longer being available. In this case the tasks concerned with these CDOs will receive error messages from Maestro API functions, and should terminate in a way to indicate task failure. This may permit the Maestro middleware to backtrack in the workflow, and retry completing it, if it has not suffered a middleware crash itself.