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

Maestro
801101

D5.3
Adaptive Transport Development

WP 5: Low-level Middleware

Christopher Haine
Utz-Uwe Haus
Adrian Tate

Date of preparation (latest version): 2019-11-29

DOCUMENT INFORMATION

Deliverable Number	D5.3
Deliverable Name	Adaptive Transport Development
Due Date	31.05.2019 (PM 9)
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 5/Task(s) 5.1
Nature	R
Dissemination Level	PU
Planned Date	31.11.2019 (PM 15)
Final Version Date	2019-11-29
Reviewed by	Manuel Arenaz
	Jacques-Charles Lafoucriere

DOCUMENT HISTORY

Partner	**Date**	**Comment**	**Version**
CRAY	2019-09-28	Initial draft	0.1
CRAY	2019-10-10	Internal feedback version	0.9
CRAY	2019-11-15	Revision after internal feedback	1.0
CRAY	2019-11-29	Final version after 1.0 has been re-reviewed	1.1

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 5 (WP5) of the Maestro project develops the low-level middleware framework.

In this document we describe how data transport is being decoupled from the transport initiation happening in the Maestro pool communication protocol, making adaptive transport usage possible: The actual transport of data is a separate layer in the Maestro middleware, extensible by new transport methods without requiring the Maestro core communication protocol that implements the data access and pool semantics to be changed. It also permits usage of different communication channels, resources, or QoS parameters between the core communication and the (bulk) data transfers for CDO content.

Contents

1 Introduction 6

2 Adaptive Transport in Maestro 6
2.1 Decoupling of CDO Transport from Pool operations 7
2.2 CDO Data (Content) Movement 8
2.3 Transport method selection 9

3 Transport Layer Abstraction 10
3.1 API proposal 12

1 Introduction

The Maestro middleware is built to enable memory- and data-awareness in workflows. While a central goal is of course to minimize data transfers, and identify and eliminate useless transfers, some data movement cannot be avoided. However, actual transport operations (or implementation of them) is hidden from the Maestro-enabled applications as much as possible. To enable the core middleware to steer and trigger the transfers the *adaptive transport layer* described in this document is being developed.

The actual transport of data is a separate layer in the Maestro middleware, extensible by new transport methods without requiring the Maestro core communication protocol that implements the data access and pool semantics to be changed. It also permits usage of different communication channels, resources, or QoS parameters between the core communication and the (bulk) data transfers for CDO content.

2 Adaptive Transport in Maestro

Transport in Maestro happens implicitly as a consequence of some participating application of a workflow performing `REQUIRE` and subsequently `DEMAND` on it, while another one performs `OFFER`. As described in D3.1 [2], the `OFFER` places the data constituting the CDO into the Maestro pool. Any `REQUIRE` for the same CDO will be satisfied by giving the requesting application access to the content. It is up to the Maestro middleware to perform the necessary transport at any time between `REQUIRE` and completion of the (possibly blocking) `DEMAND` operation of the client.

There are three fundamentally different access models for the requestor:

1. Direct access.
   This is only possible if both the original CDO content is created using Maestro resources (currently: a Mamba array) and the requesting application will only be using a Maestro API to access the CDO content. Furthermore, providing and requesting applications must be running 'close enough' to each other so that Mamba can provide access methods into the original allocation. Examples are accesses to POSIX shared memory or shared byte-addressable NVM allocations.

2. Paged access.
   This requires the requesting side to use Maestro API (currently: Mamba tiled access) for accessing the CDO content.

3. Full copy.
   An actual transfer of the entire CDO data is performed.

The difference here is not so much the access methods (e.g., Mamba API or raw pointers), but the resource management aspect: Direct access of one application to the CDO's underlying data of another application requires that maestro controls resources on both sides; for paged access it's at least on the receiving side, so that the recipient code can transparently be provided pieces of the data. If there is no Maestro resource control then a full copy (or move) of the data may be needed, incurring use of the transport layer.

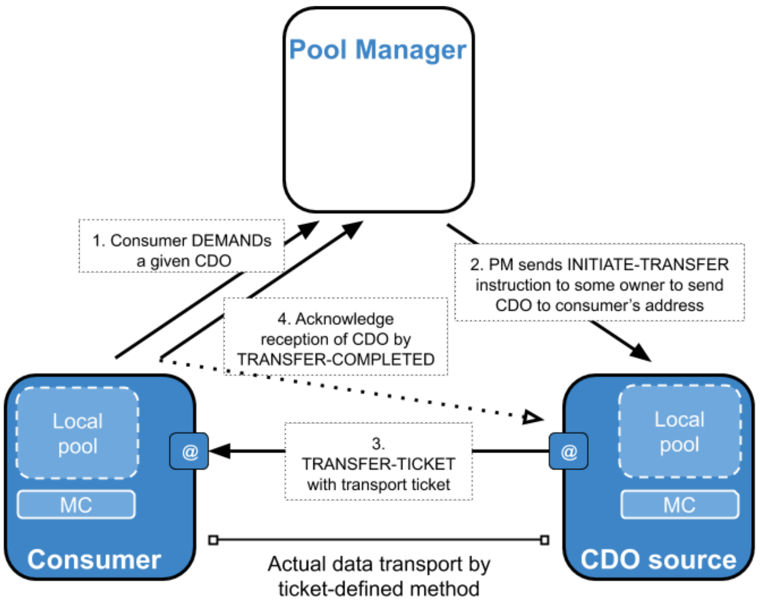
Pool Manager
1. Consumer DEMANDs a given CDO
2. PM sends INITIATE-TRANSFER instruction to some owner to send CDO to consumer's address
4. Acknowledge reception of CDO by TRANSFER-COMPLETED
3. TRANSFER-TICKET with transport ticket
Local pool
MC
@
Consumer
Local pool
MC
@
CDO source
Actual data transport by ticket-defined method

Figure 1: 4-step protocol for CDO transport initiation. Abbreviations: PM – pool manager, MC – Maestro core, @ – denotes communication port available for intra-application messaging (e.g., an OFI endpoint). Tickets are described in Section 3.

Adaptive transport is concerned primarily with the third case, but its functionality can also be used in the second case for partial transfers.

Of course, the adaptive transport may also be used internally if Maestro deems it necessary to create copies of (parts of) a CDO for strategic reasons, but such use may not be related to a particular `REQUEST` for a CDO.

Note also that performing actual transport does not preclude use of appropriate specialized hardware features. In fact, the goal of decoupling the transport initiation and the actual transport is that the most appropriate transport method can be used for each individual data transfer.

2.1 Decoupling of CDO Transport from Pool operations

Figure 1 (taken from the Maestro Architecture document [3]) shows the pool manager protocol steps that are taken to implement the `DEMAND` operation of the Maestro API:

1. An application notices that a `DEMAND` cannot be fulfilled from local data, so it sends a request to the pool manager.

2. The pool manager matches the request with a suitable instance of the CDO data. It sends a `INITIATE_TRANSFER` request to the selected data source, indicating the desired data destination and necessary layout attributes of the CDO requested.

3. The selected application sends a `TRANSPORT_TICKET` to the data destination, passing its own CDO layout attributes. This message also contains a `transport`

   `descriptor` (described below). At the same time it invokes the transport ticket execution function `mstro_transport_execute()`.

4. The data destination's maestro core transport layer also (symmetrically) invokes the function `mstro_transport_execute()` with the transport descriptor received. When this is completed it notifies the pool manager (and optionally the data source[1]) with a `DEMAND_COMPLETED` message. At that time the pool manager knows that a duplicate copy of the CDO exists and can use this information to satisfy other requests.

This 4-step protocol decouples the CDO `DEMAND` resolution and layout metadata transfer from the actual CDO content transfer. It makes the pool operations fast, and independent of CDO storage resource handling, transport methods, and layout transformations.

Note that the above diagram does not prohibit the pool manager to perform a proxy role: It can construct a `INITIATE_TRANSFER` message that directs the CDO to itself and then can perform the `TRANSFER_TICKET` to the `DEMAND` originator. This may be useful if the pool manager wants to actively keep copies of CDOs in its own pool. It can also be the basis for implementing the process of syphoning off CDOs that need to be stored in a persistent location through the pool manager.[2]

In some cases the pool manager may decide for optimisation reasons to spontaneously move a CDO between two different localities, that is without a `DEMAND` from a consumer application in the first place. This implies a skipped first step (no consumer `DEMAND`) with respect to the 4-step protocol, while the next steps occur as usual.

2.2 CDO Data (Content) Movement

The actual data movement is achieved when both data source and data destination start the `mstro_transport_execute()` method. Its arguments are the respective local CDO handle and the `transport descriptor`. The CDO handle and the `INITIATE_TRANSFER` message contain enough information to make it possible for both parties to agree on a unique transport method. The method selection is discussed in 2.3.

Depending on the transport methods chosen and the memory layer that the CDO is in, one or even both of the `mstro_transport_execute()` calls may not actually perform work, see Table 1. It is also left to the transport method implementation whether the transport will operate in a one-sided push or pull regime, or use a more complicated protocol.

We currently plan to implement the following transport methods in Maestro; more can easily be added as needed by implementing the Transport Layer Abstraction described in Section 3.

The Mamba library is currently being developed and already in use in the D3.2 MVP application. The initial version of the Universal Data Junction library (UDJ)[3] has been

[1]Depending on the transport method configuration the message to the data source may be not necessary.

[2]Alternatively, applications could be directed to ensure persistence of such objects in a decentralized fashion.

[3]`https://www.plan4res.eu/wp-content/uploads/2019/07/D6.1_SpecificationsPlatform_Summary_v1.pdf`

CDO src layer	CDO dest layer	Cross-node?	Method	Implementation Basis
RAM Layer X	RAM Layer Y	N	`mamba_copy`	Mamba (layer-specific)
RAM	RAM	Y	MPI-DPM	UDJ
RAM	RAM	Y	Dataspaces	UDJ
RAM	RAM	Y	Posix GFS	UDJ
RAM	RAM	Y	MIO	Mero
File	RAM	Y	`read`	Posix IO/MPI IO
RAM	File	Y	`write`	Posix IO/MPI IO
Object store	Any	Y	MIO	Mero
Any	Object Store	Y	MIO	Mero

Table 1: Planned Transport Methods for Maestro. RAM designates any directly-addressable memory; GFS designates a global file system shared between the workflow participants; MIO denotes the abtract object store API of Mero; `mamba_copy` designates the copy operation of the Mamba library; Posix IO/MPI-IO are the built-in IO operations of Posix and MPI libraries; Dataspaces is described in [1]; MPI-DPM designates usage of an MPI-Intercommunicator across processes.

developed in the scope of the plan4res Project[4] and already provides transparent runtime transport selection without code changes using MPI-DPM, Ceph-Rados[5], Dataspaces[6], and Global File System (GFS); it may be useful to extended it to support libfabric[7] directly to perform RDMA transfers without relying on MPI.

2.3 Transport method selection

A core feature of Maestro is the data-driven management of data movement decisions. The separation of concerns – CDO transfer steering and actual transport – makes it possible to implement various strategies for this.

Initially we envision that that the choice of transports is prioritized at the Maestro configuration level: This can be done at a global level (maestro site configuration), at the workflow level, inside a workflow at the application level, per CDO (individual/wild-carding), or by matching on certain CDO attributes.

For some transport methods – like UDJ – limited automatic adaptability is already built in and can be leveraged from the start.

At a later stage, the workflow manager can, together with the pool manager, perform such transport method choice dynamically according to telemetry data observed, by solving the underlying online optimization problem. This will not require a change in the actual protocol and transport ticket mechanism described, except for possibly an extension of the message payload to include more information about the desired transport parameters.

[4]`http://plan4res.eu/`

[5]`https://ceph.io/geen-categorie/the-rados-distributed-object-store/`

[6]`https://dataspaces.rdi2.rutgers.edu/`

[7]`https://ofiwg.github.io/libfabric/`

3 Transport Layer Abstraction

To invoke a CDO transport operation the Maestro core library in the data source application will create a transport description that we will call *Maestro transport ticket*. It is simply a structure containing all the necessary parameters to initiate the transport on both the sender and the receiver side. Table 2 shows an illustration.

Tickets will be serialized before becoming payload of a `TRANSFER_TICKETx` message; this message also contains the CDO attributes (in particular, source layout information).

Implementing a transport method thus consists of the following steps:

1. Add a new entry to `enum mstro_transport_method` and a suitable parameter data structure to `struct mstro_transport_ticket`.
2. Decide on a strategy whether a push or pull transfer method or both will be implemented, and thus whether layout transformations should happen on source or destination, or negotiated at transport time.
3. Implement the appropriate sender- and receiver-side transport method and add it to the dispatch of `mstro_transport_execute()`.

The full details for the ticket data structure will be defined by the individual transports.

Implementing the transport methods listed in1 is mostly straightforward:

Mamba Mamba-specific memory descriptors can be used directly in the ticket.

UDJ UDJ uses UUIDs to identify data objects; the CDOIDs used by Maestro can directly be used to initiate UDJ transfers. Parameters required are a suitable UDJ workflow name (unique per CDO transfer), the actual transport configuration (as normally set by `UDJ_TRANSPORT_ORDER`), and possibly transport specific parameters like whether to use MPI-DPM or not. This also includes transparently the automatic or guided choice of the UDJ backend, like Dataspaces, Ceph-Rados, MPI-DPM, or Posix file access through a global file system.

GFS The ticket contains a pathname accessible to the receiver, which will simply read the file. Notification of successful completion is desirable to permit deletion at the source, should it have been a temporary file created just for transport.

MPIIO Like for GFS the ticket contains a pathname accessible to the receiver, which will read the file using one of the `MPI_File_read` functions. Other parameters useful in the MPI-IO setup may be included.

MIO Like for GFS, a unique object identifier suitable to retrieve the data from the object store and the container identifier is sufficient.

Ticket serialization is beyond the scope of this document; it will adhere to the method used in the maestro pool manager protocol, which currently is planned to use protobuf-c[8].

[8] `https://github.com/protobuf-c/`

```
enum mstro_transport_method {
  MSTRO_TRANSPORT_METHOD_MAMBA,
  MSTRO_TRANSPORT_METHOD_UDJ,
  MSTRO_TRANSPORT_METHOD_GFS,
  MSTRO_TRANSPORT_METHOD_MPIIO,
  MSTRO_TRANSPORT_METHOD_MIO
  /* more added as needed ... */
};

struct mstro_transport_ticket_mamba {
  /** mamba array handle */
  mmbArray *data;
};

struct mstro_transport_ticket_udj {
  /** UDJ-built-in transport method to be used*/
  udjt_transport_type udj_transport;
  /** UDJ transfer identifier */
  char udj_workflow_id[UDJ_NAME_MAX];
};

struct mstro_transport_ticket_gfs {
  /** file name where CDO is stored */
  char pathname[PATH_MAX];
};

struct mstro_transport_ticket_mpiio {
  /** file name where CDO is available for MPI_File_read */
  char pathname[PATH_MAX];
  /** whether to delete after read at recipient */
  bool delete;
};

struct mstro_transport_ticket_mio {
  /** container to attach to */
  mio_container_t container_id;
  /** object id in container */
  mio_object_t objid;
};


struct mstro_transport_ticket {
  /** transfer method to be used */
  enum mstro_transport_method method;
  /** data source application ID */
  mstro_app_id src;
  /** data destination application ID */
  mstro_app_id dst;

  /** transfer-method specific data */
  union {
    struct mstro_transport_ticket_mamba mamba;
    struct mstro_transport_ticket_udj   udj;
    struct mstro_transport_ticket_gfs   gfs;
    struct mstro_transport_ticket_mpiio mpiio;
    struct mstro_transport_ticket_mio   mio;
    /* more added as needed */
   } data;
};
```

Figure 2: Maestro transport ticket types. For illustration only and not exhaustive, the actual structure content is still subject to change as implementation proceeds.

```
/** Executing a transport operation is triggered by
 ** invoking the following function with the process-local
 ** CDO handle and a transport ticket. The implementation will
 ** be able to recognize whether it is on the sending or receiving
 ** end of the transfer.
 **/
mstro_stat
mstro_transport_execute(mstro_cdo cdo,
                        struct mstro_transport_ticket ticket);

/** A transport implementation needs to provide a source- and a
 ** destination-side function to perform the data transfer; NULL
 ** indicates no operation is needed (e.g., when transfer is a
 ** one-sided action), and fill in a dispatch table with its specific
 ** functions.
 **/
struct mstro_transport_descriptor {
  /** flag whether to notify source application of completed transfer */
  bool notify_src;
  /** the function to call on the source side of the transfer */
  mstro_status (*src_func)(mstro_cdo src,
                           struct mstro_transport_ticket ticket);
  /** the function to call on the destination side of the transfer */
  mstro_status (*dst_func)(mstro_cdo dst,
                           struct mstro_transport_ticket ticket);
   };
```

Figure 3: Transport execution API. This is the API that corresponds to the *"actual data transport by ticket-defined method"* link in Figure 1.

3.1 API proposal

The API is composed of two parts: Ticket definition and transport operation execution.

For the ticket definition see Figure 2. We do not define the final API of the convenience functions that allocate, populate, and deallocate these types, but for simplicity one can imagine functions to create/destroy, serialize/deserialize transport tickers exist:

```
mstro_status mstro_transport_ticket_create(
             struct mstro_transport_ticket **result,
             mstro_enum mstro_transport_method, ...);

mstro_status mstro_transport_ticket_serialize(
             struct mstro_msg *msg_envelope,
             const struct mstro_transport_ticket *ticket);

mstro_status mstro_transport_ticket_deserialize(
             struct mstro_transport_ticket **ticket,
             struct mstro_msg *msg);

mstro_status mstro_transport_ticket_destroy(
             struct mstro_transport_ticket *ticket);
```

Note that the actual functions will not have variable arguments to enable easy generation of `Fortran ISO-C` bindings.

The transport API is given in Figure 3. It consists of transport-specific properties (subject to extension as needed), and a single `mstro_transport_execute()` function. At this time it seems perfectly acceptable to have this function operate synchronously (since it will be executed in the maestro core, in an asynchronous context like a pthread or a user-level thread).

References

[1] C. Docan, M. Parashar, and S. Klasky. Dataspaces: An interaction and coordination framework for coupled simulation workflows. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 25–36, New York, NY, USA, 2010. ACM.

[2] C. Haine, U.-U. Haus, and A. Tate. *Maestro D3.1*, 2019.

[3] C. Haine, U.-U. Haus, and A. Tate. *Maestro Architecture Document*, to appear.