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

Maestro
801101

D5.5
Adaptive Transport Release

WP 5: Low-level Middleware

Christopher Haine
Utz-Uwe Haus

Date of preparation (latest version): 2020-03-31

DOCUMENT INFORMATION

Deliverable Number	D5.5
Deliverable Name	Adaptive Transport Release
Due Date	2020-03-31 (PM 20)
Deliverable lead	HPE
Authors	Christopher Haine Utz-Uwe Haus
Responsible Author	Utz-Uwe Haus (HPE) e-mail: utz-uwe.haus@hpe.com
Keywords	[Memory Systems, Data Middleware, HPC]
WP/Task	WP 5/Task(s) 5.1
Nature	Software
Dissemination Level	PU
Planned Date	March 2020
Final Version Date	2020-03-31
Reviewed by	

DOCUMENT HISTORY

Partner	Date	Comment	Version
HPE	2020-03-30	Initial draft	0.1
HPE	2020-04-28	Revision after internal feedback	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 5 (WP5) of the Maestro project develops the low-level middleware framework.

In this document we describe how to exercise the data transport code as it is currently available in maestro-core (git tag `d5.5-review/d5.5` after submission of the deliverable), showing decoupled transport on a shared file system and using MIO/MERO[2], with the transport initiation happening in the Maestro pool communication protocol, decoupled from the actual transport, making adaptive transport usage possible.

Contents

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, as described in Deliverable D5.3 is 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 Preparation

2.1 Mero testbed

If you want to use MIO, the only supported object store for MIO at this time is Mero. If you have access to a system where Mero 1.4 is installed you can skip this section. The Mero version on the Sage prototype system has not been verified to work correctly with MIO at this time. Any feedback on experiments for this should be reported as issues against `https://gitlab.version.fz-juelich.de/maestro/mio`, or used to close issues on the MIO project.

For now, we suggest to use a virtual machine, built according to the instructions provided at `https://bscw.zam.kfa-juelich.de/bscw/bscw.cgi/3066374`. Alternatively, we have prepared a Vagrant + VirtualBox recipe at `https://gitlab.version.fz-juelich.de/maestro/maestro-mero-vm`. Follow the instructions in the `README.md` there to build a VM.

Note that you will need `git-lfs` installed to clone this repository[1], and then a recent Vagrant[2] installed along with the vbguest additions[3].

2.2 Obtaining the maestro-core source code

Check out the release tagged `D5.5` from `https://gitlab.version.fz-juelich.de/maestro/maestro-core`. This is a tag on the `master` branch; rapid additions, in particular with regard to attribute handling, are expected on top of it during the review period.

If you are using the VM-based MIO/Mero testbed **you need to check out the repository in a directory inside the VM**, e.g., the home directory of the vagrant user. Trying to use the host-side diretory mounted at `/vagrant` is known to fail in various

[1] macos: `brew install git-lfs`

[2] http://vagrantup.com/

[3] `vagrant plugin install vagrant-vbguest`

weird ways, in particular when the host system is not a linux system. We have enabled `ssh-agent` credential forwarding, so `git clone` and also pushes should be easy.

2.3 Configuring Maestro-core with or without MIO/Mero support

Follow instructions to setup maestro-core in `https://gitlab.version.fz-juelich.de/maestro/maestro-core/-/blob/master/INSTALL.md`.

To enable MIO, specify `--with-mio` as an argument to `configure`. It is not enabled by default. The MIO library will be built as as dependency of maestro-core automatically.

Each Mero client (via MIO) needs a YAML configuration file. For the tests in the maestro-core distribution such files are created automatically[4]. Instructions to create such configurations files in general are in `https://gitlab.version.fz-juelich.de/maestro/mio/-/blob/master/README`). To make maestro-core aware of the location of the MIO configuration file, set the environment variable `MSTRO_MIO_CONFIG` to the path of the configuration file.

When maestro-core is built with MIO support, and an application does not have `MSTRO_MIO_CONFIG` set, adaptive transport will degrade by disabling MIO for this application. A `WARN` message will be logged. Similar fallback does not happen if `mio_init()` fails – that is considered a hard error.

3 Demonstration

Adaptive transport demonstration is carried by an early version of maestro-core that supports some primordial multi-process functionality, that is inter-process communication commanded by the client via Maestro API and operated and relayed by the maestro-core instance to a pool manager process, with basic CDO movement without metadata. Maestro-core supports two transport methods at the moment: GFS and MIO.

3.1 Principle

We will be assuming Mero 1.4 is up and running on the user system (steps on how to achieve that on `https://bscw.zam.kfa-juelich.de/bscw/bscw.cgi/3066486`), and Maestro-core installed as described in the previous section, be it with or without Mero support.

The pool manager interlock demonstration (`./tests/check_pm_interlock.sh`) – which is automatically launched with `make check` – is a multi-process setup consisting in having two clients `OFFER` one CDO each, and then each `DEMAND` the CDO the other has offered. A shell script starts the three processes: one pool manager and two clients emulating maestro-enabled applications. Client behaviour consists in simple Maestro API usage, and is set at configuration time. In this demonstration, both clients `DECLARE` and set attributes – a size and data pointer – for both CDOs, then each `OFFER` one. Afterwards, both clients `REQUIRE` and `DEMAND` the CDO the other has offered, then `DISPOSE`

[4]See for instance `.../tests/mio-config-{PM1,C1,C2}.yaml` for the `check_pm_declare.sh` test.

of the CDOs and finalize. The pool manager exits when clients finalized and thereby concludes the demonstration.

A `DEMAND` message is a trigger of the dedicated 4-step protocol, as described in D5.3 [1], which handles CDO movement within Maestro. Maestro-core provides two transport methods at the moment, namely GFS and MIO. The default transport method in maestro-core is GFS. Clients are offered a way of choosing their own preferred transport method, via the `MSTRO_TRANSPORT_DEFAULT` environment variable, which means maestro-core will try to use the client preferred transport method whenever possible. In some specific cases, certain transport methods are not applicable. Indeed, when a ticket is issued for a given transfer, maestro-core checks transport-specific limitations, such as for instance the inability of MIO to handle data sizes which are not a page size multiple, and is able to finally decide on another transport method on-the-fly, which would be GFS in this case. At this point if MIO is not configured (no –with-mio configure option set) or no MIO config file is specified on a given client, GFS will be selected automatically.

3.2 Expected Outcome

While default demonstration logging is profusely verbose, it is also possible to set logging level to errors and warnings only via the `MSTRO_LOG_LEVEL` (error level = 0, warning level = 1) environment variable. Note that Mero itself may bypass Maestro to print errors and/or fail fatally. A small subset of errors logged by MIO (for instance, the `errno=-2` linux error code, when trying to create a file) are not actual Maestro-core/MIO errors, and the demo may still succeed.

3.2.1 Success

After launching the pool manager interlock script and a reasonably short amount of time, the test suite tool will report one success per client, the prompt will be returned, and after a few moments some more logging will appear on the command line, notifying the pool manager's clean exit.

3.2.2 Failure

Failure of a client will typically cause a live lock in the other client waiting for its CDO, and also in the pool manager waiting for them forever.

Mero panic errors bypass Maestro error handling. Typical known causes of such catastrophic failure are:

- -2 No such file or directory: typically when the `MSTRO_MIO_CONFIG` environment variable is not set or if the path is not correct.
- -110 Time out: will make a client hang indefinitely long.
- -28 No space: (especially on virtual machine) it means Mero virtual disks are full. Mero panics lead to dumping heavy trace files, and it takes a few of them to fill up all the space. Quick fix: `rm m0trace*`
- -11 Try again: causes Mero panic.

4 Known issues

- Starting a job with the same mero configuration too quickly will result in *address already in use* error from MIO/Mero.
- On a system with an aggressive firewall setup all maestro-core startups with pool manager functionality may hang. This is because maestro-core is opening one connection on each interface (or rather: each usable `libfabric` endpoint) of the node to accept incoming connections. When using the `tcp;ofi_rxm` or `sockets` provider the port number will be chosen randomly, and is thus hard to open up in the firewall configuration (if you have access to that at all). For high-performance interconnects this is usually not an issue.
- We've occasionally seen a hard-to-reproduce hang in `check_pool_local`, mainly on MacOS. This will be investigated after completion of D5.5
- Logging from the maestro core can be excessive. While it is easy to reduce it to `ERR` or `WARN` level by setting `MSTRO_LOG_LEVEL=0` or `MSTRO_LOG_LEVEL=1`, a finer granularity than using `INFO` and `DEBUG` (e.g., by maestro-core subsystem) seems desirable in the medium term.
- The schema parsing infrastructure is incomplete and the `check_schema_parse` test is currently failing.

References

[1] C. Haine, U.-U. Haus, and A. Tate. *Maestro D5.3*, 2019.

[2] S. Wu, G. Umanesan, and S. Narasimhamurthy. *Maestro D5.2*, 2019.