D6.6
Telemetry demonstration

Work package	**WP6 Middleware Validation and Systems Software Demonstrators**	
Author	Marc Pérache	CEA
Author	Laurent Nguyen	CEA
Author	Sai Narasimhamurthy	Seagate
Author	Christopher Haine	HPE
Reviewer #1	Domokos Sármány	ECMWF
Reviewer #2	Dirk Pleiter	JUELICH
Dissemination Level	Public	
Nature	Report	

Date	**Author**	**Comments**	**Version**	**Status**
21.11.2021	Marc Pérache	Initial version	v0.1	Draft
30.11.2021	Marc Pérache	Final version	v1.0	Final

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

Executive Summary

Contents

Glossary

Abbreviation	Expansion
ADDB	Analysis and Diagnostics Database
CEA	The French Alternative Energies and Atomic Energy Commission (CEA)
CDO	Core Data Object, fundamental data unit understood by the Maestro middleware and communicated between the Maestro middleware and its applications
HPC	High-performance computing
JSON	JavaScript Object Notation is an open standard file format and data interchange format that uses human-readable text
MIO	Maestro I/O interface

Motr	Motr is a distributed object storage system, targeting mass capacity storage configurations.
MPI	Message Passing Interface
OpenMP	Open Multi-Processing is an application programming interface (API) that supports multi-platform shared-memory multiprocessing programming.
PAPI	Performance Application Programming Interface
Rsyslog	Rsyslog is an open-source software utility used on UNIX and Unix-like computer systems for forwarding log messages in an IP network.
Selfie	(SElf and Light proFIling Engine) is a tool to lightly profile Linux commands without compiling.
WP	Work Package

1 Introduction

This document describes the telemetry tools that were developed to efficiently use the Maestro framework. These tools will help Maestro users to optimize their Maestro calls to reduce the time spent in IO. There are different levels of telemetry starting from statistics coming from low-level components such as MIO or Maestro Core to user-oriented statistics thanks to modification in the Selfie tools. We will describe all these tools in this deliverable and we will eventually conclude with a small example of a real workload: the Computational Fluid Dynamics benchmark.

2 MIO

2.1 Introduction

The following section demonstrates the Maestro IO Interface (MIO) telemetry. MIO has been described in detail in the WP5 D5.2(Storage, Data Mapping and Memory Integration Design)[1] and D5.6 (Storage, Data Mapping and Memory Integration Implementation for Maestro)[2]. The MIO telemetry information is included as part of Motr ADDB (Motr Analysis and Diagnostics Database). More details about the Motr object store has been described in D5.2. ADDB has been described in previous Maestro deliverables D5.1 (Telemetry Design). Please note that Maestro middleware Telemetry can also be included as part of Motr ADDB. We, however, focus on gathering MIO specific telemetry from Motr ADDB as an example. General ADDB examples were already provided in D5.4 (Telemetry prototypes)[3].

2.2 Demonstration

A sample MIO "workflow", which is essentially the steps we define to generate and analyse MIO telemetry information, was run to collect, parse and analyze MIO telemetry information. MIO telemetry is essentially telemetry records generated during the operation of MIO. Figure 1 provides a screenshot of the different steps of this workflow. Even though the demo shows how to process telemetry data in Motr ADDB, the same steps are followed for telemetry data in log format.

It is to be noted that the two distinct phases of these steps are telemetry data collection from MIO and telemetry data analysis. Some simple examples of telemetry data analysis is shown as part of these discussions.

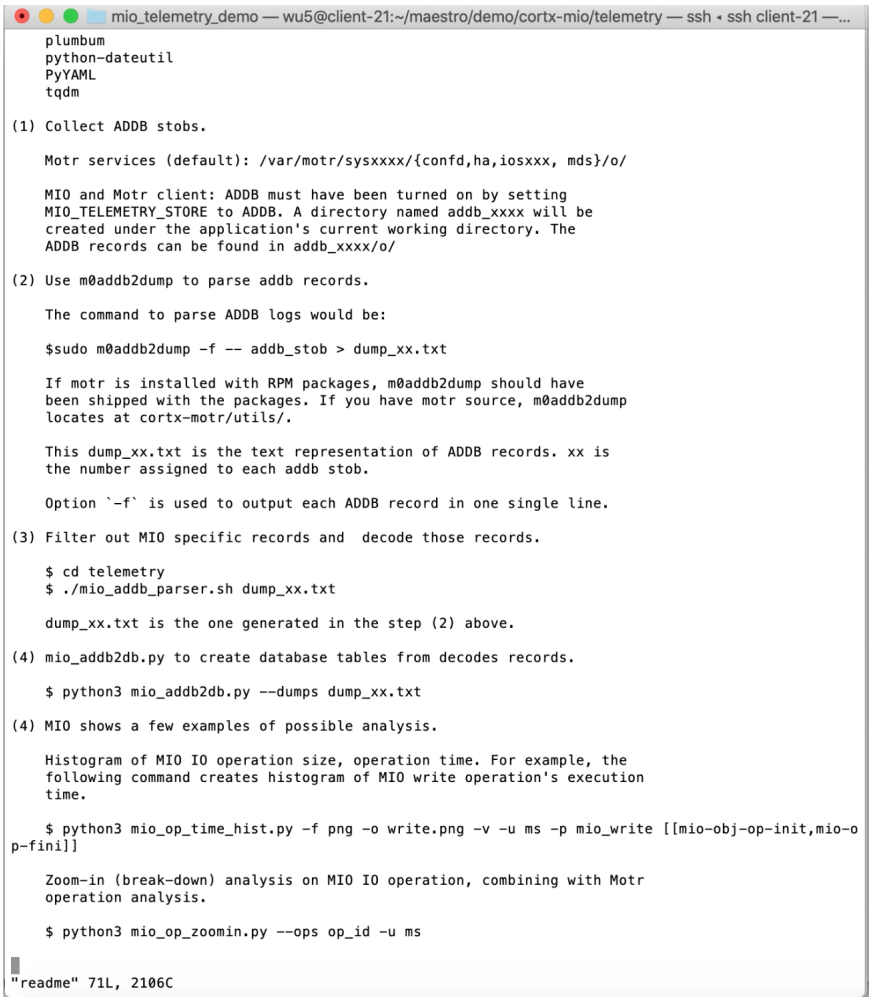

Figure 1: The steps to generate and analyse MIO telemetry

The benchmark “mio_io_perf” was used in the tests on top of the MIO interface to generate the I/O load with MIO which eventually triggers the telemetry records. Figure 2 provides the sample output of mio_io_perf.

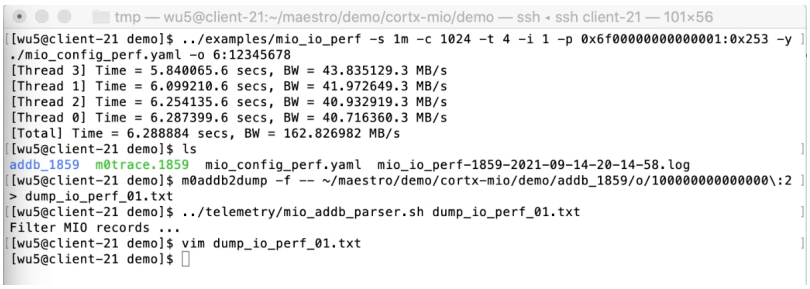

Figure 2: Output of the mio_io_perf benchmark

Python utilities were provided to analyze and decode telemetry records. The MIO telemetry records that are filtered is shown in Figure 3. The telemetry records show the various MIO operations and the time stamps corresponding to those. These show for example, the start of MIO operations, the points where MIO operations are handed over to the backend Motr, the IO sizes corresponding to MIO object sizes, the process IDs invoking these operations, etc. Rather than going into detail into each of these operations, the essential idea is to show that telemetry mechanisms can be used to capture these MIO calls. More details can be obtained here [4]

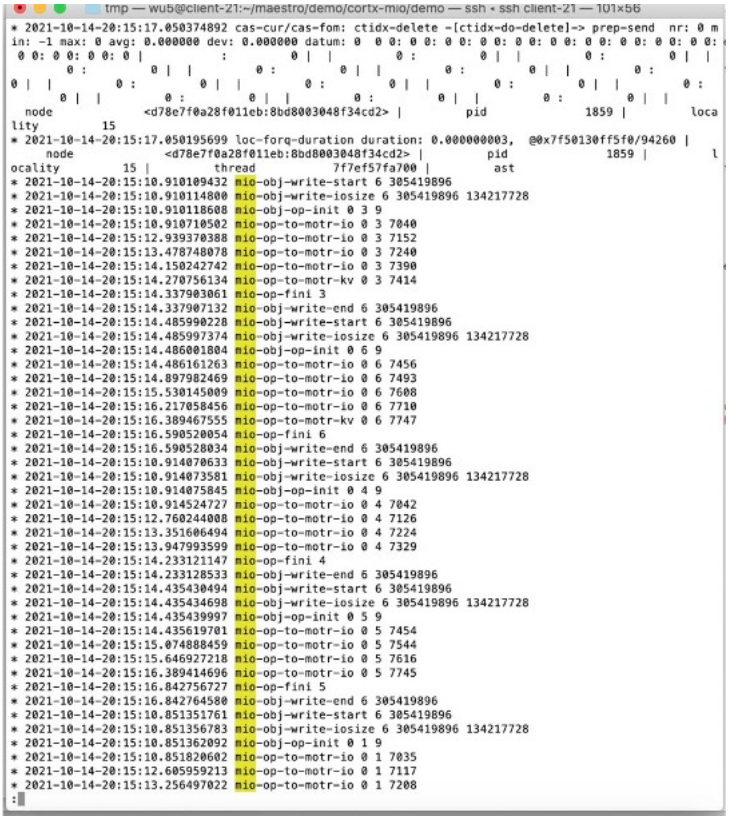

Figure 3: Utilities filtering mio telemetry records

The decoded records were then populated into the database for analysis. This is shown in the figure 4. This picture validates that our telemetry tool (mio_addb2db.py) reads decoded ADDB records and loads them into data-

base tables ready for analysis. In this way, we extract the whole stack information from mio to motr and store them into the same database. It is convenient for all kinds of analysis, particularly if one wants to trace down the stack and to understand some performance bottleneck. The tables shown here are motr related, only those at the bottom with mio prefix are mio related tables, for example, mio_session table records the object open and close time, other tables like mio_op_to_motr_op stores the relationship betwen MIO op and Motr op, so we can link MIO layer IOs to Motr IOs.

Taking the mio_op_to_motr_op table as example, it contains the mapping between MIO op to Motr layer op, so we can link the MIO layer down to Motr. Each table normally records the timestamp of each event and other event specific information.

The black bar just shows the progress of populating the tables.

Figure 4: Decoded ADDB records were included in the database for analysis

An example histogram of key MIO IO operation latencies are shown in the figure 5. They show the latencies of a set of given operations (In this case the mio-obj-op-init mio-op-fini pair) and how many of operations ("frequency") have the given latency. For example there is one operation pair that has a latency of 2400 ms and two operation pairs that have a latency of 2500 ms. More sophisticated analysis can be done, such as detailed IO path analysis as MIO instruments its IO path and can integrate with existing Motr IO path data points. The same is true for Maestro operations included in the Telemetry database.

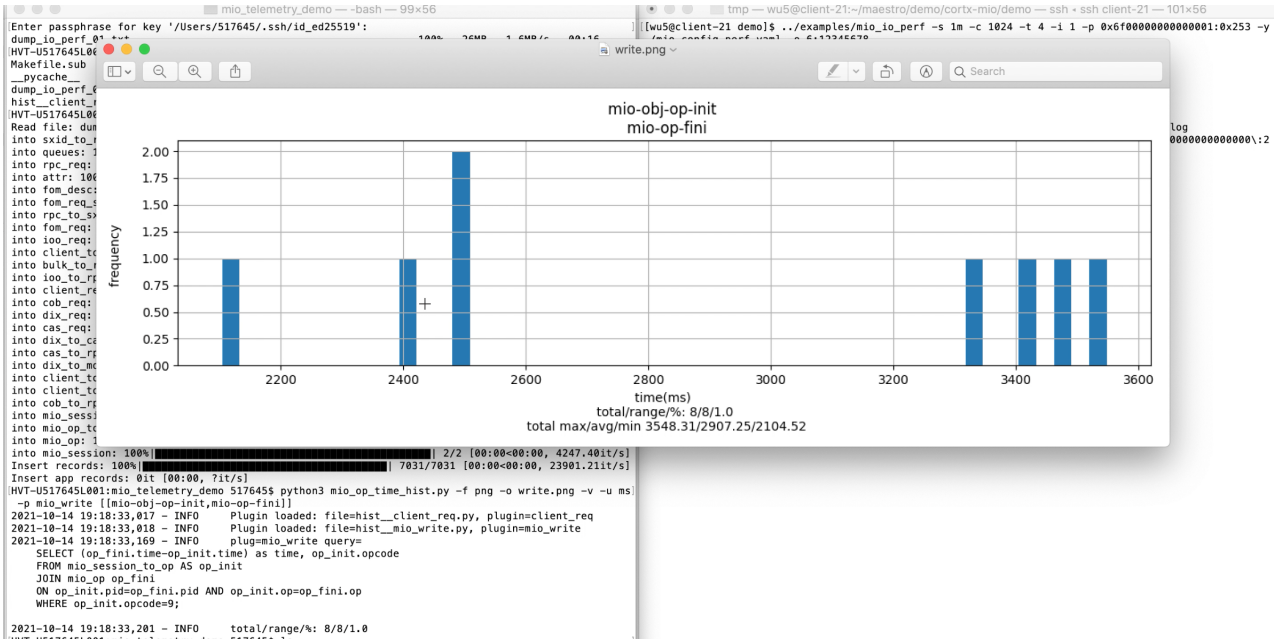

Figure 5: Example mio Telemetry analytics

3 Maestro Core

Maestro Core itself natively provides two ways of reading telemetry parameters, one *post mortem* with a fixed set of telemetry parameters and one live with selected telemetry parameters.

3.1 Execution report

Each Maestro Core client plus the Pool Manager produces a CSV report at the very end of the execution, as part of the regular log with "Info" level of verbosity:

```
1
2  [I:stats] Simple_Pool_Manager:0 1 t140700321308416 (localhost.localdomain 7646) 696135924500: mstro_stats_report_csv(statistics.c:528)
3  "workflow-id";"app-id";"hostname";"pid";"thread";"General:component-runtime:0:1";
4  "Pool:pm-num-imm-withdraws:1";"Pool-Protocol:pm-num-declare:1";"Pool-Protocol:pm-num-demand:1";
5  "Pool-Protocol:pm-num-dispose:1";"Pool-Protocol:pm-num-join:1";"Pool-Protocol:pm-num-leave:1";
6  "Pool-Protocol:pm-num-offer:1";"Pool-Protocol:pm-num-pool-events:1";"Pool-Protocol:pm-num-require:1";
7  "Pool-Protocol:pm-num-retract:1";"Pool-Protocol:pm-num-seal:1";"Pool-Protocol:pm-num-tickets:1";
8  "Pool-Protocol:pm-num-transport-completions:1";"Pool-Protocol:pm-num-withdraw:1"
9
10
11 [I:stats] Simple_Pool_Manager:0 1 t140700321308416 (localhost.localdomain 7646) 696135943038: mstro_stats_report_csv(statistics.c:568)
12 "check_pm_interlock_wf_7644";1;"localhost.localdomain";7646;"t140700321308416";17298149987;2;5;2;5;2;2;2;66;3;1;5;2;2;2
```

Where the first log line consists of labels, and the second of values. For example, "General:component-runtime" denotes the time spent between `mstro_init()` and `mstro_finalize()` by the given component, and here has a value of 17298149987 (ns). Pool protocol operations have counters as well. In this example "Pool-Protocol:pm-num-offer", which keeps track of the number of offered CDOs, has a value of 2. More elaborate telemetry parameters such as roundtrip transport histogram per bucketed CDO size, as first described in D5.4 [3], are currently under development.

Maestro Core logs record in particular the pool operations with timestamps, which can be conveniently parsed via Core tools (`\$MAESTRO_DIR/visualise`) to a friendlier visual output (see Figure 6), to help understanding the progress of the workflow. In addition, Core tools can be used to further analyse the performance of the Maestro core by visualising the distribution of time delays encountered by CDOs during the execution of a Maestro enable workflow. For example, Figure 7 shows the histograms produced by analysing the timings reported in the pool manager logs for CDO operations and state changes. Such visualisations can help locate and pinpoint performance bottlenecks and anomalies.

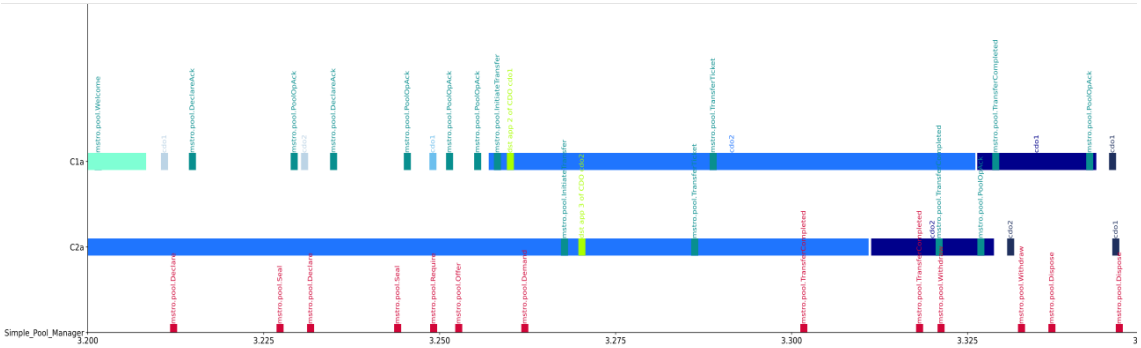

Figure 6: Log-based timeline visualisation of a Maestro-enabled workflow. Each component ("C1a", "C2a", "Simple_Pool_Manager") is represented by a timeline, the latter being labelled with occurring Maestro operations

3.2 Listener

Maestro Core provides a sample application `\$MAESTRO_DIR/tests/simple_telemetry_listener.c` that shows how one may write a custom Maestro telemetry listener, that is an application that peers during execution time into the Maestro-enabled workflow telemetry, namely using Maestro Events.

This allows to cherry-pick exactly the events or telemetry parameters that need to be recorded. For production workflows that may produce such large logs that performances are impacted, they would usually be disabled by users. Figure 8 shows an example of implementation and usage of the telemetry listener. It visualises the lifetime of each workflow component by recording all the JOIN/LEAVE events.

4 Selfie

4.1 Description

Selfie[5] is a lightweight profiling tool developed by the CEA. It runs a light profiling of jobs on supercomputers and provides statistics of the usage of

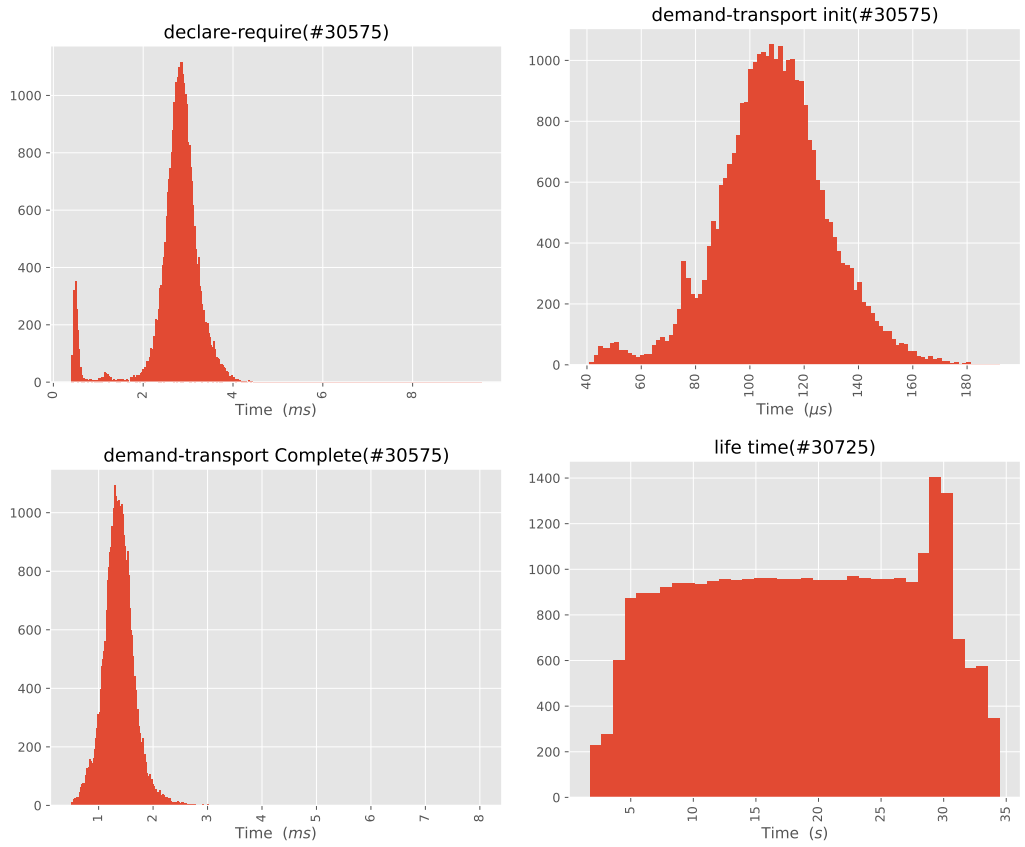

Figure 7: Log-based analysis of CDO operations of a Maestro-enabled workflow. The histograms show the distribution of delays encountered by CDOs, namely: 1. Time from CDO is declared until it is noticed by consumer via event subscriptions and the CDO is in required state (top left); 2. Time from a CDO was demanded until the pool manager issue an initiate-transfer ticket to the producer (top right); 3. Time from a CDO was demanded until it was received on the consumer side (bottom left); 4. CDO lifetime from declare time to withdraw time (bottom right).

the computer to the administrators. This tool has been in production since 2016 on the clusters managed by the CEA. Selfie allows to have a vision of the user jobs on the following points:

- Number of cores used (MPI or OpenMP)
- Time spent in the job: execution time, system time, CPU time
- Maximum memory used by the job processes
- Number of calls and time spent in MPI calls
- Number of calls and time spent in POSIXIO calls
- PAPI hardware counters: number of cycles and number of instructions
- Statistics on Maestro core usage

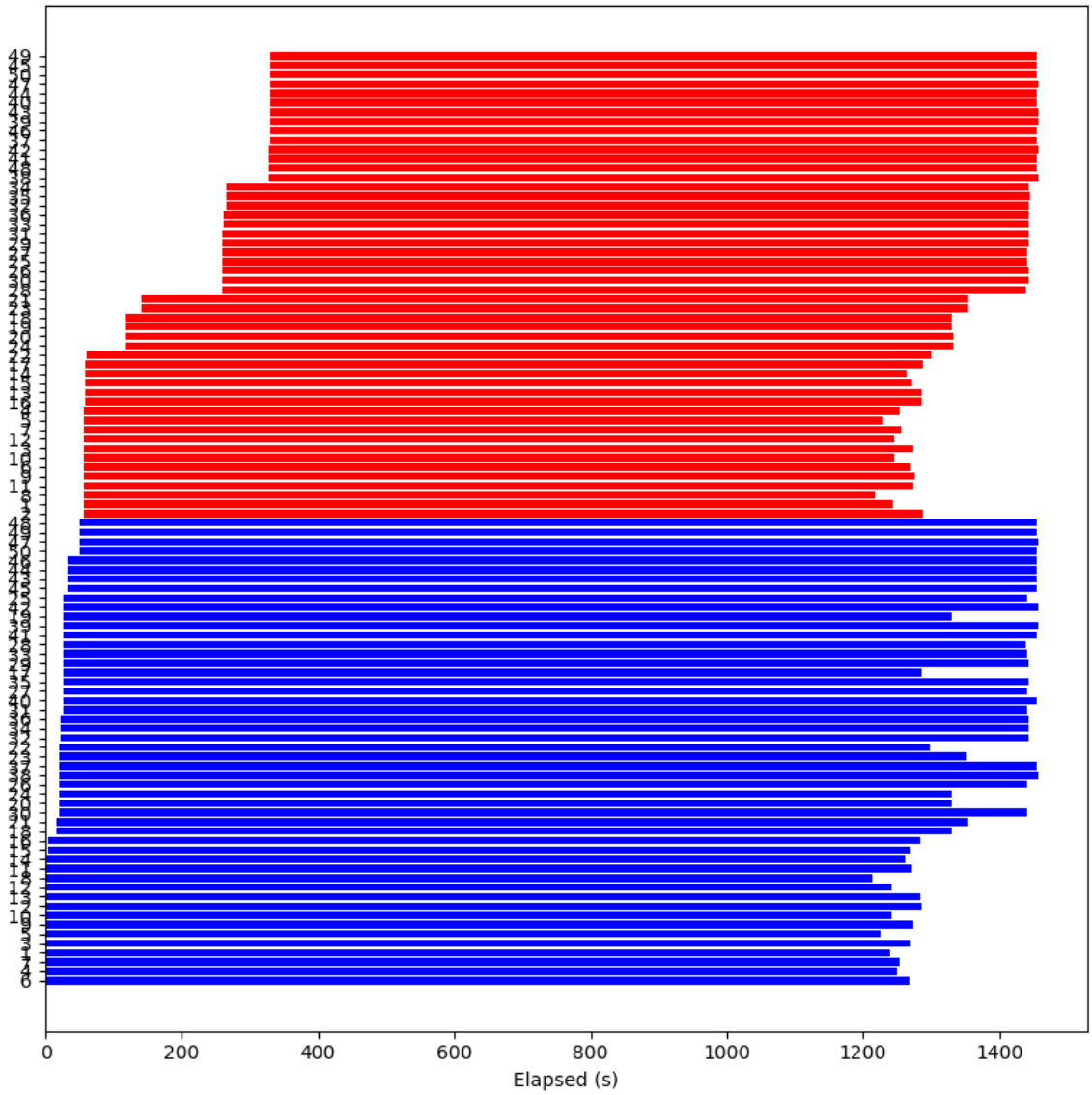

Figure 8: Visualisation based on JOIN/LEAVE events as recorded by a telemetry listener plugged in an IFS workflow, comprising fifty producers (red bars) and fifty consumers (blue bars). Each bar represents one component's elapsed time between JOIN and LEAVE.

- Statistics on MIO usage

When a process is executed, Selfie is loaded via the LD_PRELOAD mechanism and at the end of the job, it sends a summary as JSON in the system logs. On CEA clusters, the profiling data is post processed in an ElasticSearch cluster and visualised with Kibana interface. This processing allows, among other things, to visualize job profiles by user or user group. The CEA wishes to use Selfie to find problematic job profiles (too much IO, too much MPI communication, low memory usage, etc.)

The Maestro project includes a telemetry component. The telemetry should allow us to understand the mechanisms and interactions between the

jobs and the Maestro storage system. In the context of this telemetry, Selfie is part of the user code profiling. In addition to the statistics mentioned above, a development was made in Selfie to profile Maestro API and internal calls to the Mamba library, with focus on the functions used by the Sirius mini-app. Selfie's architecture allows the addition of features in the form of plugins. The "maestro-core" plugin was therefore developed to profile the Maestro API. This plugin allows:

- Obtain a profiling summary of the API. Like the basic Selfie features, this plugin gives a summary of the time spent and the number of calls to the Maestro API routines used in the users' codes, including those to Mamba library.
- To provide traces of calls to the Maestro and Mamba API. The goal here is to be able to trace the Maestro and Mamba calls in the user code to obtain the timestamp of the call. The principle of Maestro for codes is to expose CDO objects. These calls make these objects available. In addition to providing the timestamp, Selfie also retrieves the name of the object and thus makes it possible to follow the life of this object during the execution of the code. In the case of Mamba calls, only timestamps and function names are provided since no CDO objects are involved.

The traces and the summary of the job concerning Maestro are written in the system logs on the local node. These logs should be gathered (using for example Rsyslog) in order to be analyzed and correlated with the other telemetry data to obtain the desired information for the code developers.

4.2 Execution examples

Here an example of Selfie output concerning trace for CDO named "my-cdo-17":

```
Oct 26 03:51:35 localhost.localdomain selfie[14370]: {
    "hostname": "localhost.localdomain", "pid": 14370,
    "timestamp": 1635220295,
    "function": "cdo_declare", "cdo_name": "my-cdo-17",
    "wtime": 0.000902 }
Oct 26 03:51:35 localhost.localdomain selfie[14370]: {
    "hostname": "localhost.localdomain", "pid": 14370,
    "timestamp": 1635220295,
    "function": "cdo_seal", "cdo_name": "my-cdo-17",
    "wtime": 2.647146 }
Oct 26 03:51:35 localhost.localdomain selfie[14355]: {
    "hostname": "localhost.localdomain", "pid": 14355,
    "timestamp": 1635220295,
    "function": "cdo_declare", "cdo_name": "my-cdo-17",
    "wtime": 0.001140 }
Oct 26 03:51:35 localhost.localdomain selfie[14370]: {
```

```
    "hostname": "localhost.localdomain", "pid": 14370,
    "timestamp": 1635220295,
    "function": "cdo_offer", "cdo_name": "my-cdo-17",
    "wtime": 5.977347 }
Oct 26 03:51:38 localhost.localdomain selfie[14355]: {
    "hostname": "localhost.localdomain", "pid": 14355,
    "timestamp": 1635220298,
    "function": "cdo_demand", "cdo_name": "my-cdo-17",
    "wtime": 2.499489 }
Oct 26 03:51:38 localhost.localdomain selfie[14370]: {
    "hostname": "localhost.localdomain", "pid": 14370,
    "timestamp": 1635220298,
    "function": "cdo_withdraw", "cdo_name": "my-cdo-17",
    "wtime": 2.448270 }
```

And then, the summary of concerned PID:

```
Oct 26 03:51:40 localhost.localdomain selfie[14370]: {
    "utime": 9.03, "stime": 1.28, "maxmem": 0.01,
    "hostname": "localhost.localdomain", "maestro_time": 13.38,
    "maestro_count": 264, "USER": "vagrant", "pid": 14370,
    "timestamp": 1635220300, "wtime": 13.38,
    "command":
        "/home/vagrant/maestro-core/tests/ecmwf_events_producer" }
Oct 26 03:51:40 localhost.localdomain selfie[14355]: {
    "utime": 14.62, "stime": 3.65, "maxmem": 0.01,
    "hostname": "localhost.localdomain", "maestro_time": 14.00,
    "maestro_count": 1293260, "USER": "vagrant", "pid": 14355,
    "timestamp": 1635220300, "wtime": 16.47,
    "command":
        "/home/vagrant/maestro-core/tests/ecmwf_events_consumer" }
```

Note that both processes created more CDO during their execution.

4.3 Selfie on the SAGE Prototype

We next describe the installation of Selfie on the SAGE prototype system that is installed at. the Juelich Supercomputing Center. This enables tools, middleware and applications using Maestro running on SAGE to use Selfie. Selfie is currently installed on one of the client nodes “visnode-01” on the SAGE platform. Selfie uses environment variables for setting its configuration parameters. There are three of them as shown on the following config file:

```
umanesan1@visnode-01~[997]cat .env_selfie.sh

# selfie

```

```
export SELFIE_CONFIGFILE=/etc/selfie.conf

export LD_PRELOAD=/usr/local/lib/selfie.so

export SELFIE_OUTPUTFILE=selfie.out

umanesan1@visnode-01~[998]
```

These are 1) location of the Selfie config file, 2) location of Selfie binary and 3) name and location of Selfie output file. In the above example the location of the output file is current working directory, and the name of the file is selfie.out. All that is needed to use Selfie on visnode-01 is simply issuing the following command to load the environment variables:

```
umanesan1@visnode-01~[998]source ~/.env_selfie.sh
```

Once that is completed Selfie can intercept the execution of any non-Linux binary and collect telemetry data. This is demonstrated below:

```
umanesan1@visnode-01~[1000]mkdir mytmp

umanesan1@visnode-01~[1001]cd mytmp/

umanesan1@visnode-01~/mytmp[1002]ls -la

total 4

drwxr-xr-x  2 umanesan1 jusers    6 Nov 24 10:53 .

drwxr-xr-x 34 umanesan1 jusers 4096 Nov 24 10:53 ..

umanesan1@visnode-01~/mytmp[1003]../a.out

sz = 4

sz = 8

sz = 8

umanesan1@visnode-01~/mytmp[1004]ls -la

total 8

drwxr-xr-x  2 umanesan1 jusers   24 Nov 24 10:54 .

drwxr-xr-x 34 umanesan1 jusers 4096 Nov 24 10:53 ..

-rw-r--r--  1 umanesan1 jusers  352 Nov 24 10:54 selfie.out
```

```
umanesan1@visnode-01~/mytmp[1005]cat selfie.out

{ "utime": 0.00, "stime": 0.00, "maxmem": 0.00,
"hostname": "visnode-01", "mpi_time": 0.00,
"mpi_count": 0, "mpiio_time": 0.00,
"mpiio_count": 0, "mio_time": 0.00,
"mio_count": 0, "maestro_time": 0.00,
"maestro_count": 0, "USER": "umanesan1", "pid":
13168, "timestamp": 1637747663, "wtime": 0.00,

"command": "/home/users/jusers/umanesan1/sage/a.out" }

umanesan1@visnode-01~/mytmp[1006]
```

Please note that a.out is simple c program. It does not do much but it is not a standard Linux command. By default, Selfie is configured not to intercept started Linux binary programs.

4.3.1 Selfie for MIO

Selfie can now intercept MIO calls in visnode-01. This is demonstrated below using standalone MIO read/write tests.

```

umanesan1@visnode-01~/github/cortx-mio/tests[1014]
../examples/mio_rw_threads -s 4096
-c 1 -n 10 -t 1 -y ./mio_config.yaml -o 1:12346800

a:12346800      56718ca07bd1d381eef48c77f762d052
56718ca07bd1d381eef48c77f762d052

9:12346800
b7a376289b410b31d7b59e3ce5e0a674
b7a376289b410b31d7b59e3ce5e0a674

8:12346800
6a90d3117fe60ae4bd1ccead5bdb9424
6a90d3117fe60ae4bd1ccead5bdb9424

7:12346800
1cce7efabfa05e1cbb57500b098c7856
1cce7efabfa05e1cbb57500b098c7856

6:12346800
566a09bc2cec8c9f38391d499e145105
566a09bc2cec8c9f38391d499e145105

```

```
5:12346800
9e2b8b1f97afded7f8a5bdfb42fe40e5
9e2b8b1f97afded7f8a5bdfb42fe40e5

4:12346800
ed6db143057f5ce0b0bf2b68709a85ac
ed6db143057f5ce0b0bf2b68709a85ac


3:12346800
ed89f91a732296f20428f9b581854e5c
ed89f91a732296f20428f9b581854e5c


2:12346800
6faf85af791e3069ddeff792aea61336
6faf85af791e3069ddeff792aea61336


1:12346800
316c022293fb92351fa5498baf8144da
316c022293fb92351fa5498baf8144da


[Final Report]  Objects TODO: 10        Completed: 10
Failed: 0       Matched: 10

umanesan1@visnode-01~/github/cortx-mio/tests[1015]more selfie.out

umanesan1@visnode-01~/github/cortx-mio/tests[1016]cat selfie.out

{ "utime": 1.11, "stime": 0.77, "maxmem": 0.23,
"hostname": "visnode-01", "mpi_time": 0.00,
"mpi_count": 0, "mpiio_time": 0.00,
"mpiio_count": 0, "mio_time": 3.85,
"mio_count": 1, "maestro_time": 0.00,
"maestro_count": 0, "USER": "umanesan1", "pid":
5934, "timestamp": 1637674204, "wtime": 6.69,
"command": "/home/users/jusers/umanesan1/sage/
github/cortx-mio/examples/.libs/mio_rw_threads"
}
```

As illustrated above Selfie now collectes MIO telemetry data such as mio_time and mio_count from the read and write test runs.

5 Telemetry usage in real applications

We have also carried out the evaluation with the Computational Fluid Dynamics, Numerical Weather Prediction and Electronic Structure Calculation benchmarks to evaluate the extensibility of Selfie with a real workloads. All three of these benchmarks are described in detail in deliverable D6.5[6].

5.1 Computational Fluid Dynamics

This benchmark uses the open-source proxy application HydroC, a 2D hydrodynamic simulation code. For this test case we tested up to 160 MPI ranks the Maestro-enabled version of the benchmark. For the evaluation, we will not use the standard syslog output of Selfie but instead the user-defined output, configurable by standard users.

Since this is an MPI application, we will not provide a detailed trace for each process. The following trace demonstrates the use of Selfie by applying it on one of theses processes:

```
1  { "utime": 35.99, "stime": 17.41, "maxmem": 0.07,
2  "hostname": "nodename",
3  "mpi_time": 33.32, "mpi_count": 19071, "mpi_version": 3.10,
4  "mpi_libversion": "open mpi",
5  "mpiio_time": 0.00, "mpiio_count": 0, "mpiio_version": 3.10,
6  "mpiio_libversion": "open mpi",
7  "mio_time": 0.00, "mio_count": 0,
8  "maestro_time": 0.62, "maestro_count": 218107,
9  "USER": "USER", "OMP_NUM_THREADS": "1", "pid": 17127,
10 "timestamp": 1636731341, "wtime": 73.24,
11 "command": "PATH_TO_BINARY/hydro.her" }
```

We can see on line 8 that the cost of a Maestro call is pretty small, illustrates the level of optimisation inside the maestro-core library. The $218\,107$ calls take only $0.6s$. HydroC is a standard scientific computing application. In this type of applications, the time spend in I/O is often close to 10% of the execution time. Here, we are close to 2%. We can also see in this trace on line 7 that Selfie can report MIO statistics. The goal of Selfie is to highlight potential performances issues in MPI, Maestro, or MIO. Dedicated tools like Maestro telemetry and/or MIO telemetry will also help to solve these issues.

5.2 Numerical Weather Prediction

The Maestro-enabled version of the IFS workflow, also described in [6], may be configured to act essentially as an I/O benchmark. As such, it benefits from a detailed breakdown of calls in the maestro-core library, rather than of an aggregate. The following is the output for a single producer when all maestro calls all traced. For brevity, we have left out the functions with zero counts (never called).

```
{
  "utime": 217.99,
  "stime": 65.6,
  "maxmem": 1,
  "hostname": "visnode-01",
  "mstro_init_time": 0.14,
  "mstro_init_count": 1,
  "mstro_finalize_time": 1.02,
  "mstro_finalize_count": 1,
  "mstro_version_count": 2,
  "mstro_cdo_declare_time": 0.77,
  "mstro_cdo_declare_count": 24361,
  "mstro_cdo_offer_time": 8.62,
  "mstro_cdo_offer_count": 24360,
  "mstro_cdo_withdraw_time": 1.62,
  "mstro_cdo_withdraw_count": 24360,
  "mstro_cdo_dispose_time": 0.48,
  "mstro_cdo_dispose_count": 24361,
  "mstro_cdo_seal_time": 2.43,
  "mstro_cdo_seal_count": 24360,
  "mstro_cdo_demand_count": 1,
  "mstro_cdo_attribute_set_time": 0.75,
  "mstro_cdo_attribute_set_count": 316081,
  "mstro_cdo_attribute_get_time": 0.2,
  "mstro_cdo_attribute_get_count": 121800,
  "mstro_cdo_attribute_set_default_time": 0.02,
  "mstro_cdo_attribute_set_default_count": 24361,
  "mstro_cdo_name_count": 121803,
  "maestro_time": 16.06,
  "maestro_count": 705852,
  "USER": <user_id>,
  "pid": 5160,
  "timestamp": 1637857506,
  "wtime": 99.98,
  "command": "<build-dir>/bin/multio-hammer"
}
```

We can see that around half of the total time spent in maestro-core calls (line 29) is spent on 'offer' (line 13), while the rest is spent mainly on metadata management. Setting and getting attributes is comparatively cheap (lines 22 and 24), although we emphasise that the this example of one-producer and one-consumer is for illustrations only.

The following is a similar output for the consumer.

```
{
  "utime": 178.15,
  "stime": 131.99,
  "maxmem": 0.22,
```

```
5   "hostname": "visnode-01",
6   "mstro_init_time": 0.13,
7   "mstro_init_count": 1,
8   "mstro_finalize_time": 1.05,
9   "mstro_finalize_count": 1,
10  "mstro_version_count": 2,
11  "mstro_cdo_declare_time": 0.36,
12  "mstro_cdo_declare_count": 19801,
13  "mstro_cdo_offer_count": 1,
14  "mstro_cdo_withdraw_count": 1,
15  "mstro_cdo_dispose_time": 0.81,
16  "mstro_cdo_dispose_count": 19801,
17  "mstro_cdo_seal_count": 1,
18  "mstro_cdo_demand_time": 15.54,
19  "mstro_cdo_demand_count": 19800,
20  "mstro_cdo_selector_create_count": 2,
21  "mstro_subscribe_count": 2,
22  "mstro_subscription_poll_time": 8.63,
23  "mstro_subscription_poll_count": 17296414,
24  "mstro_cdo_attribute_set_time": 0.09,
25  "mstro_cdo_attribute_set_count": 19801,
26  "mstro_cdo_attribute_get_time": 0.12,
27  "mstro_cdo_attribute_get_count": 79201,
28  "mstro_cdo_attribute_set_default_time": 0.01,
29  "mstro_cdo_attribute_set_default_count": 19801,
30  "mstro_cdo_name_count": 59405,
31  "maestro_time": 26.74,
32  "maestro_count": 17534035,
33  "USER": <user-id>,
34  "pid": 5114,
35  "timestamp": 1637857505,
36  "wtime": 102.13,
37  "command": "<build-dir>/bin/multio-maestro-syphon"
38 }
```

We can see that the vast majority of 'Maestro time' is spent in `mstr_cdo_demand` (line 18), which is the call that transfers data from the producer to the consumer, and `mstro_subscription_poll` (line 22), which is the event loop to detect events. In this case, we detect 'offer' events that signal when the producer has made an object available for data transfer. The telemetry data is also useful to sanity-check if the same number of CDOs have been declared, offered and disposed.

5.3 Electronic Structure Calculation

The following is an extract of a Mamba tracing during a run of the Electronic Structure Calculation mini-app.

```
{ "hostname": "nid04000", "pid": 24994, "timestamp": 1637753691,
"function": "mmb_init", "wtime": 0.030868 }
...
{ "hostname": "nid04000", "pid": 24994, "timestamp": 1637753693,
"function": "mmb_index_set", "wtime": 0.002567 }
{ "hostname": "nid04000", "pid": 24994, "timestamp": 1637753693,
"function": "mmb_tile_at", "wtime": 0.005656 }
{ "hostname": "nid04000", "pid": 24994, "timestamp": 1637753693,
"function": "mmb_index_set", "wtime": 0.002567 }
{ "hostname": "nid04000", "pid": 24994, "timestamp": 1637753693,
"function": "mmb_tile_at", "wtime": 0.005658 }
{ "hostname": "nid04000", "pid": 24994, "timestamp": 1637753693,
"function": "mmb_tile_duplicate", "wtime": 0.516800 }
{ "hostname": "nid04000", "pid": 24994, "timestamp": 1637753693,
"function": "mmb_tile_duplicate", "wtime": 0.516821 }
{ "hostname": "nid04000", "pid": 24994, "timestamp": 1637753693,
"function": "mmb_tile_get_space_local_handle", "wtime": 0.001929 }
{ "hostname": "nid04000", "pid": 24994, "timestamp": 1637753693,
"function": "mmb_tile_get_space_local_handle", "wtime": 0.001930 }
{ "hostname": "nid04000", "pid": 24994, "timestamp": 1637753693,
"function": "mmb_tile_merge", "wtime": 0.248715 }
{ "hostname": "nid04000", "pid": 24994, "timestamp": 1637753693,
"function": "mmb_tile_merge", "wtime": 0.248790 }
...
{ "hostname": "nid04000", "pid": 24994, "timestamp": 1637753693,
"function": "mmb_finalize", "wtime": 0.000076 }
```


and the corresponding summary


```
{ "utime": 0.44, "stime": 0.73, "maxmem": 0.15,
"hostname": "nid04000", "mamba_time": 0.81, "mamba_count": 25030,
"USER": "bignamic", "pid": 24994, "timestamp": 1637753693,
"wtime": 1.92,
"command": "/users/bignamic/sirius_miniapp_maestro_new" }
```


6 Concluding Remarks

In this deliverable, we described the telemetry detailed/low-level tools developed during the Maestro. We also presented the user-oriented high-level profiling tool Selfie that was extended to support the Maestro ecosystem. We presented a small evaluation of the Selfie tools with a real workload and a test case that connects Selfie and the ADDB-based performance record collection system.

References

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

[2] S. N. Sining Wu and G. Umanesan, *Maestro D5.6*, 2021.

[3] S. Wu, L. Nguyen, S. Narasimhamurthy, and G. Umanesan, *Maestro D5.4*, 2020.

[4] S. Wu and G. Umanesan, "Mio github." https://github.com/Seagate/cortx-mio/tree/master/telemetry, 2021.

[5] CEA, "Selfie github." https://github.com/cea-hpc/selFIe.

[6] D. Sármány, M. Lompar, S. Smart, J. Badwaik, M. Pérache, and C. Bignamini, *Maestro D6.5*, 2021.