D5.4 Monitoring Prototypes and Telemetry considerations

Work package:	**5 – Low level middleware**	
Author(s):	Nguyen Sai Ganesan Sining	Laurent Narasimhamurthy Umanesan Wu
Reviewer #1	Utz-Uwe	Haus
Reviewer #2	Dirk	Pleiter
Dissemination Level	Public	
Nature	Other	

Date	Author	Comments	Version	Status
19/03/2020	Sai Narasimhamurthy		0.1	Draft
30/03/2020	Laurent Nguyen		0.2/ 0.3	Draft
26/04/2020	Sai Narasimhamurthy		0.4	Draft
08/10/2020	Laurent		0.5	Draft

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.

	Nguyen/Utz-Uwe Haus			
09/10/2020	Sai Narasimhamurthy		0.6	Draft
14/10/2020	Sining Wu		0.7	Draft
14/10/2020	Ganesan Umanesan		0.8	Draft
14/10/2020	Sai Narasimhamurthy	Internal Review Version	0.9	Draft
13/11/2020	Sai Narasimhamurthy	Inputs	1.1	Draft
24/11/2020	Sai Narasimhamurthy	Consolidated inputs from other authors	1.2	Draft
30/11/2020	Sai Narasimahmurthy	Additional comments incorporated	1.3	Pre- Final for Submission
01/12/2020	Laurent Nguyen	Addressed specific comments for CEA	1.4	Pre-Final for submission
02/12/2020	Sai Narasimhamurthy	Addressed specific comments from Utz	1.5	Final for submission

Executive Summary

We present in this report the first prototype implementation of telemetry data collection based on telemetry data that can be collected from the different layers of the Maestro stacks. This is enabled through the “selFIe” and the “ADDB” tools.

This report also describes broadly possible telemetry metrics that are (1) Interesting to capture on behalf of applications, and/or (2) Interesting for system administrators, and/or, (3) Interesting for individual component developers within Maestro. This is a continuation of the work on the architecture in D5.1. The goal of these metrics is to help create a “story” of what happened during the running of the workflow/application.

For the purposes of this first prototype, data is all made available in Syslog from which telemetry data corresponding to the different layers (Maestro, MIO and Mero) can be filtered and analysed. This will allow to achieve the main goal of this effort, namely providing an analysis tool for the demonstration efforts within Maestro. We will, however, look towards adding and utilizing a specialised database towards the end of the project.

Contents

Figures

Glossary of Acronyms

Acronym	Definition
ADDB	Mero Analysis and Diagnostics Database
fom	File operation machine
Fop	File operation
MIO	Maestro IO interface
selFIe	Self and Light proFIling tool Engine

1. Introduction

This report provides a first prototype implementation of Telemetry data collection in the Maestro project from Maestro, MIO and Mero enabled by the selFIe and ADDB tools. selFIe captures data at a higher level in the stack between the applications and Maestro. It can also enable instrumentation of some Maestro and MIO calls. The Mero specific telemetry is collected separately through “ADDB” or, “Analytics and Telemetry Data Base”. This follows from the telemetry architecture that was presented in the previous deliverable D5.1[5].

The telemetry records are moved to Syslog at this time in the PoC. Also very limited telemetry records are captured, but will be/can be easily expanded. The main idea is to show that the telemetry framework works. Other aspects such as a central database as described in the architecture deliverable D5.1[5] will be addressed in upcoming deliverables.

The report also describes potentially interesting Telemetry records that may be of interest for system administrators, applications or the developers of individual Maestro stack components (We take the example of the ECMWF use case. Note that ECMWF operates the system as well as owns the application, albeit by different groups within the organisation). These requirements “flow down” to the telemetry tools.

2. Application Perspectives (ECMWF)

ECMWF runs operational weather forecasts, along with a research and development workload. There are several fundamental use cases for telemetry information:

1. System monitoring in the operational workload
2. Workload monitoring in the operational pipeline
3. Failure and performance issue retrospective diagnostics
4. Workload development and performance investigation

The perspective of these users and their associated requirements are summarized by requirement R1.14: “Maestro to record and communicate usage statistics - type of storage used and their load, write (production) rate, read (consumption) rate, etc. - to human operators or system monitoring/logging tools”. This requirement is phrased quite generally, and the types of telemetry data that are useful differ markedly between these uses.

We see that there are three types of data which have use in the various scenarios:

a) Instantaneous (near) real-time performance information.
b) Aggregated (possibly time-bucketed, or histogrammed) performance information broken down by system component.
c) Low level activity monitoring.

For most uses, the first two of these categories are useful. We really see having access to low level activity tracking telemetry as useful primarily for debugging and development. As a result, it is good to be able to turn on logging and telemetry streams to track all actions that are occurring in the system, but keeping this information available for all components at scale is probably not useful (or feasible).

For use case (a) and (b), operators need to be able to see relatively simple aggregated statistics that correspond to the components of the system and the components of the workflow, respectively. The closer to real-time that this information is available, the more useful it is. This information should be at a high level. It would include, for instance, the number of operations per second, the data volumes per second and the total available and used pool resources. This allows the operators to take the correct action or call the appropriate responsible analysts within the time-critical operational window.

Use case (c) needs somewhat more detailed information. In particular, we would like to be able to identify what has caused bottlenecks and failures in the associated workflow execution. But it is sufficient for this information to be available only retrospectively.

We next discuss how these application perspectives are included in defining important metrics captured at the application-Maestro interface (through SelFie), the Maestro middleware telemetry, MIO and Mero telemetry.

3. Metrics

3.1 selFIe

selFIe is a simple profiling tool which captures basic profiling data. In Maestro, the main purpose of selFIe core is to get profiling information from user codes in order to optimize the code. For Maestro, selFIe will help to identify binaries and the different execution phases of the binary during applications workflow. selFIe will also capture MIO and Maestro calls as described in another section. In this section, we will describe basic metrics for selFIe.

Key attributes/ parameters	Key	Metric	Source (if needed)	Potential Significance/why this would be important
Timestamp	timestamp	sec	Workload application	To track when the event happened
Executable/code name	command	string	Workload application	Get the name of the application which triggers the event
Node name	hostname	string	Workload application	Get the node/hostname where the application runs
Jobid	Jobid	Id	Workload application	From job scheduler of self id
Execution wall time	wtime	ms	Workload application	To find the overall execution time of the workload
Time spent in POSIX read calls (or by call)	posix_read_time	ms	Workload application	Bottleneck identification
Number of POSIX read calls	posix_read_count	counter	Workload application	Deeper Understanding of application I/O
Time spent in POSIX write calls (or by call)	posix_write_time	ms	Workload application	Bottleneck identification
Number of POSIX write calls	posix_write_count	counter	Workload application	Deeper understanding of the application I/O

Table 1 selFIe Metrics examples

The metrics above will be useful to understand which application code or binaries were involved during the execution of the workflow. selFIe collects all counters and times spent in functions and gives a summary at the end of execution. For Maestro, the aim is to provide a trace of metrics values.

3.2 Maestro Middleware

The core middleware will keep statistics for its own purposes, in particular to perform adaptive decisions and monitor components. Integration of the measurements is part of the `libmaestro-core` code. Please note that selFIe also captures many of the Maestro middleware API calls.

The decision-making procedures can be influenced by external measurements that are made available by some telemetry interface, but no such interface has been defined so far, so no attempt to ingest external 'live' telemetry information (i.e., data collected during a workflow execution by components outside of `libmaestro-core`) has been made so far. Some of the internal statistics can be made available to the maestro telemetry infrastructure. This can be done by parsing the logging output from standard output, or from syslog (`libmaestro-core` already permits to redirect all logging to syslog by setting `MSTRO_LOG_DST=syslog`). Log based information contains workflow component identifier, hostname, thread ID, process ID and timestamp, as well as the name of the maestro core module, so can be parsed as a time series of events and classified according to these criteria. Statistics are recorded internally in 3 forms: counter, key/value tables, and key/timespan, and can will typically be reported at program end, but could also be dumped in regular intervals or at strategic execution points. We believe that the following data can be produced with low or reasonable overhead and will be useful to the end user, profiling tools, and system administrators:

Key attributes/ parameters	**Metric**	**Source (if needed)**	**Potential Significance/why this would be important**
Component runtime	ms	component	obvious
Component pool association time	ms	Pool manager	
Time spent waiting on LEAVE, time spent waiting on WITHDRAW, time spent waiting on DEMAND	ms/ms per CDO size class/ms per CDO	Component and Pool manager	Can be useful to diagnose situations where CDOs need to be moved off the disassociating component or withdrawing the CDO. Tracking at the individual CDO

			level will be expensive; a method to record data only for select (sets of) CDOs at this fine granularity would be useful T1.4 (see D6.1)
Roundtrip times for pool protocol messages that have an acknowledgement message	ms	Component	Rough to finegrained profiling of network and pool manager handling performance. Bucketing of this is up for discussion: per CDO is too expensive; CDO size may not make sense for some messages (e.g., DECLARE);
Adaptive transport times per source/sink pair and CDO size	ms	Pool manager	Transport performance profiling
Number of CDOs created, possibly by size class	counter	Component, Pool manager	Observing component behavior in complex workflows T1.2 (see D6.1)
Number of CDOs created per second	sec	Component, Pool manager	T1.1 (see D6.1)
number of pool operations, by operation	Counters	Component	Classification of components by their pool usage
Time spent (de-)serializing attributes	ms	component	Pinpointing performance issues in complicated user-defined attribute schemata
Demand-initiated data movement (by size and memory layer)	counter	Pool manager	Baseline unmanaged data movement demands of a workflow
Manager-initiated data movement (by size and memory layer)	Counter	Pool manager	Measuring effects of data movement scheduling algorithms

History of transfer bandwidth, per memory tier	Bps	Component, Pool Manager	T1.3 (see D6.1)
CDO search times when creating/accessing CDO GROUP by attribute matching	ms	Component, Pool Manager	T1.5 (see D6.1)
Type of storage used and their load, write (production) rate, read (consumption) rate, etc	GB/s	Component, Pool Manager	T1.6 (see D6.1)
Allocation statistics from Mamba memory management layer		Component, Pool Manager	T1.7 (see D6.1)

Table 2 Maestro middleware telemetry examples

Parsing log information from maestro-core is a useful method to gather fine-grained data, but extensive logging can cause significant overhead for Maestro, so users will typically want to turn down logging. Hence, instead of parsing maestro-core logs for relevant data and forwarding it to the appropriate telemetry data accumulation framework at scale, we believe that the cleanest and most maintainable solution is to write a 'vampire' application that can be added to any existing workflow that runs with a pool manager, which connects to the workflow like a normal maestro application and uses the SUBSCRIBE functionality of the maestro pool protocol to observe events as they happen. This application can then support logging to various telemetry data accumulation frameworks, without requiring changes to the maestro core library. It can also be parameterized to configure telemetry behavior and could even be written in a scripting language for extra flexibility.

An early prototype will be included in the upcoming 0.2.0 release of the maestro library under the name 'simple_telemetry_listener'.

3.3 MIO

For MIO, selfie will capture functions from MIO such as the below.

Key attributes/ parameters	Metric	Source (if needed)	Potential Significance/why this would be important
Timestamp	sec	Workload application	Log when MIO event happens
Name of MIO function	string	Workload application	Name of MIO event
Time spent in MIO function	ms	Workload application	Time spent in MIO function call

Table 3 MIO telemetry examples

3.4 Clovis & Mero/ADDB

These telemetry examples help to understand the behavior and performance of Maestro backend storage and understand performance problems that will aid both data center *operators running* Maestro workflows and backend storage system developers. This is for both debugging and profiling purposes.

These will help to rule out/identify any problems within the storage system when persistent storage related performance issues are seen by Maestro applications.

Key attributes/ parameters	Metric	Source	Potential Significance/why this would be important
Mero Node, timestamp, state	sec	Mero Server nodes	Time spent in different “states” during the processing of I/O requests on the Mero server nodes. This helps to identify any problems during server-side request processing and helps Maestro backend developers to pinpoint problems on their end. Example Use case: Maestro took too much time to persist a write operation.
Mero Node, length	Bytes	Mero Server	Length of I/O requests on the Mero server side can be analysed as I/O request size is one of the

		nodes	critical performance parameters and observing this helps the Maestro backend developers to optimize their implementation or suggest possible issues higher up the stack. Example Use case: Maestro took too much time to persist a write operation.
Mero Node, timestamp, state (used for throughput calculation)	Bytes/sec	Mero Server nodes	Throughput as measured in the different storage nodes. This helps to identify anomalous nodes causing any possible performance issues. Example Use case: Maestro application read throughput was low, but one example storage node showed very good performance.
Mero Node, timestamp, state (used for latency calculation)	sec	Mero Server nodes	I/O latency as measured in the different storage nodes. This helps to identify anomalous nodes causing any possible performance issues. Example Use case: Maestro application persistent read latency was low.

Table 4 Clovis/Mero telemetry examples

With regards to input from application inputs, the metrics above will help to cover the following,

1. System monitoring in the operational workload – general monitoring of backend storage behavior and performance
2. Failure and performance issue retrospective diagnostics – getting diagnostic information from backend storage for any possible performance issues
3. Workload development and performance investigation – identifying and investigating any performance problems in backend storage.

We will cover aggregated performance information and low-level activity monitoring in the category of types of data.

4. Prototypes

This presents the first set of prototypes of telemetry generation by the different layers as described above and then made available to Syslog as a first step.

This is implemented in a virtual machine environment where an example workflow is run on top of Maestro.

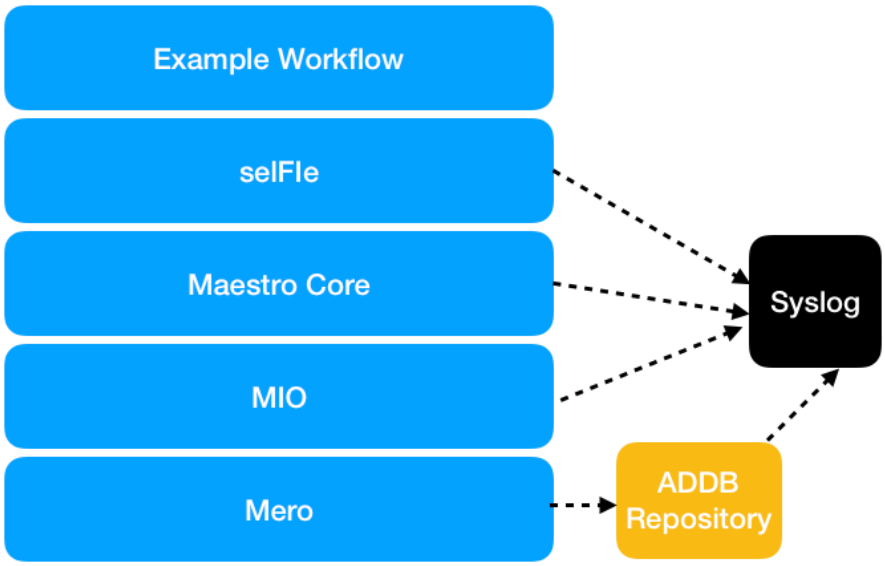

Figure 1 Stack used for PoC

The Virtual Machine runs CentOS 7.7 with a downgraded kernel version 3.10.0 to support Lustre client packages that are required for installing Mero packages [6]. The Mero instance on this virtual machine is configured to run as single node Mero cluster meaning that client, storage and network all run on the same CentOS 7.7 machine and thereby enabling installing and running of MIO, Maestro middleware and selFIe on the same CentOS 7.7 for ADDB, syslog and other telemetry data collection. See figure below for installed software versions.

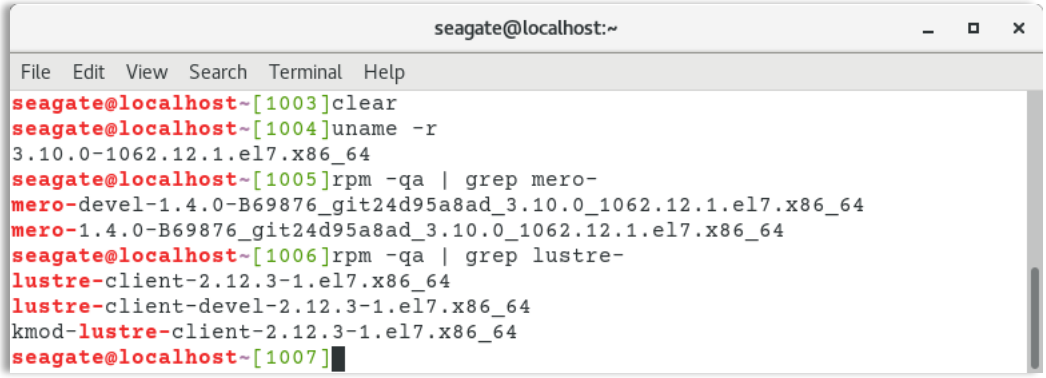

Figure 2 VM environment used for PoC

4.1 Workflow Description

The workflow for testing telemetry is a test named “check_pm_declare.sh” coming from the maestro-core.[2] The test launches a pool manager and then two clients. The clients calls the following functions:

mstro_init ()
mstro_cdo cdo=mstro_cdo_declare(name, MSTRO_ATTR_DEFAULT, &cdo)
mstro_cdo_offer (cdo)
mstro_cdo_withdraw (cdo)
mstro_cdo_dispose (cdo)
mstro_finalize( )

The client hence declares, offers withdraws and disposes the CDO. More details on CDOs can be found in submitted Maestro architecture deliverables in WP3 (D3.1 and D3.2)

4.2 selFIe Telemetry

selFIe is a library which will capture the calls from applications in the workload. The LD_PRELOAD environment variable is used to load the selFIe library and run applications. selFIe doesn’t need system services or root execution: it is the application which runs along with selfie.
selFIe will put information in syslog (with the call “syslog”) with JSON format. By default, it uses the “info” loglevel, but it is configurable at installation.

For example, our demonstrator is launched with the command:

$ ./check_pm_interlock.sh

selFIe is enabled through LD_PRELOAD.We get the following output in the syslog messages:

selfie[12310]: { "jobid": "12310", "pid": "12310", "timestamp": 1602147153, "function": "mio_init", "wtime": 0.18 }
selfie[12310]: { "jobid": "12310", "pid": "12310", "timestamp": 1602147153, "function": "mstro_init", "wtime": 0.18 }

[2] https://gitlab.version.fz-juelich.de/maestro/maestro-core/-/blob/master/tests/check_pm_interlock.sh.in

selfie[12353]: { "jobid": "12353", "pid": "12353", "timestamp": 1602147155, "function": "mio_init", "wtime": 1.70 }
selfie[12353]: { "jobid": "12353", "pid": "12353", "timestamp": 1602147155, "function": "mio_thread_init", "wtime": 0.00 }
selfie[12353]: { "jobid": "12353", "pid": "12353", "timestamp": 1602147155, "function": "mstro_init", "wtime": 1.75 }
selfie[12353]: { "jobid": "12353", "pid": "12353", "timestamp": 1602147155, "function": "mstro_cdo_declare", "wtime": 0.00 }
selfie[12353]: { "jobid": "12353", "pid": "12353", "timestamp": 1602147155, "function": "mstro_cdo_offer", "wtime": 0.02 }
selfie[12353]: { "jobid": "12353", "pid": "12353", "timestamp": 1602147155, "function": "mstro_cdo_withdraw", "wtime": 0.02 }
selfie[12353]: { "jobid": "12353", "pid": "12353", "timestamp": 1602147155, "function": "mstro_cdo_dispose", "wtime": 0.00 }
selfie[12353]: { "jobid": "12353", "pid": "12353", "timestamp": 1602147155, "function": "mio_thread_fini", "wtime": 0.00 }
selfie[12353]: { "jobid": "12353", "pid": "12353", "timestamp": 1602147155, "function": "mstro_finalize", "wtime": 0.01 }
selfie[12354]: { "jobid": "12354", "pid": "12354", "timestamp": 1602147155, "function": "mio_init", "wtime": 1.81 }
selfie[12354]: { "jobid": "12354", "pid": "12354", "timestamp": 1602147155, "function": "mio_thread_init", "wtime": 0.00 }
selfie[12354]: { "jobid": "12354", "pid": "12354", "timestamp": 1602147155, "function": "mstro_init", "wtime": 1.87 }
selfie[12354]: { "jobid": "12354", "pid": "12354", "timestamp": 1602147155, "function": "mstro_cdo_declare", "wtime": 0.00 }
selfie[12354]: { "jobid": "12354", "pid": "12354", "timestamp": 1602147155, "function": "mstro_cdo_offer", "wtime": 0.03 }
selfie[12354]: { "jobid": "12354", "pid": "12354", "timestamp": 1602147155, "function": "mstro_cdo_withdraw", "wtime": 0.01 }
selfie[12354]: { "jobid": "12354", "pid": "12354", "timestamp": 1602147155, "function": "mstro_cdo_dispose", "wtime": 0.00 }
selfie[12354]: { "jobid": "12354", "pid": "12354", "timestamp": 1602147155, "function": "mio_thread_fini", "wtime": 0.00 }
selfie[12354]: { "jobid": "12354", "pid": "12354", "timestamp": 1602147155, "function": "mstro_finalize", "wtime": 0.01 }
selfie[12353]: { "jobid": "12353", "pid": "12353", "timestamp": 1602147157, "function": "mio_fini", "wtime": 2.46 }
selfie[12353]: { "jobid": "12353", "pid": "12353", "utime": 0.26, "stime": 0.13, "maxmem": 0.11, "mio_time": 4.16, "mio_count": 4, "maestro_time": 1.80, "maestro_count": 6, "USER": "user", "timestamp": 1602147157, "wtime": 4.26, "command": "/maestro-core/tests/simple_client" }
selfie[12354]: { "jobid": "12354", "pid": "12354", "timestamp": 1602147158, "function": "mio_fini", "wtime": 2.49 }
selfie[12354]: { "jobid": "12354", "pid": "12354", "utime": 0.29, "stime": 0.14, "maxmem": 0.11, "mio_time": 4.31, "mio_count": 4, "maestro_time": 1.93,

"maestro_count": 6, "USER": "user", "timestamp": 1602147158, "wtime": 4.42, "command": "/maestro-core/tests/simple_client" }
selfie[12310]: { "jobid": "12310", "pid": "12310", "timestamp": 1602147163, "function": "mstro_finalize", "wtime": 0.00 }

In this example, selFIe takes the PID of the process as job identifier. In other context, it can take a job batch identifier (for example from SLURM).

3 processes were profiled by selFIe:

- the pool_manager
- both clients

The lines in italics are the trace of each calls to Maestro or MIO. Those lines contain data to identify the processes or the job which put the data (jobid, timestamp). For now, they have the name of the function call and time spent in those functions. Later, we may put more data like object name, object size.

The lines in bold format are the standard output of selFIe which are the profiling summary of the run. It contains the number of calls and the total time spent the calls.

We can note that the pool manager didn't write a profiling summary: the reason is the process was exited abruptly without giving time to selfie to write its output.

4.3 Mero Telemetry

The following provides some examples of ADDB records at Mero storage services which are actually obtained during the running of the representative workload as described in Section 4. Combining Mero service side ADDB records with the selFIe MIO records, this gives us a more completed picture of Maestro's IO access patterns.

There are two types of ADDB records: (1) Data points: measurements that are captured in a particular moment in particular place of code. (2) Sensors: other measurements correspond to parameters that change "continuously". They are too fast for each change to be sampled individually. ADDB periodically samples sensors and records their measurements

For each ADDB records, a line starting with '*' is a measurement. Following lines, starting with '|', are measurement's context, such as a timestamp, and its context specifies the node and the service process/thread where the measurement was. A few ADDB examples captured are displayed and explained below:

(1) Mero services use a non-blocking state machine called fom (file operation machine) to process a request from client. Each `fom` usually is consist of multiple phases. Detailed ADDB records on each phase of a `fom` give insight into how a Mero service

process a request and may help us understand overhead of each phase.

The following ADDB `fom` records shows how an object is created. The record gives us information such as type of the request, the service ID, process and thread which processes the request, the client which sends the request etc.

** 2020-10-12-11:35:34.396675559 fom-descr service: <7300000000000002:1>, sender: 5389c54c7ff0120, req-opcode: M0_IOSERVICE_COB_CREATE_OPCODE, rep-opcode: M0_IOSERVICE_COB_OP_REPLY_OPCODE, local: false | node <f4eebce02dc43aa:a5e598c6672ae4dc> |pid 8394 |locality 0 |ast |fom @0x7ff5c4025e50, 'COB create/delete/getattr' transitions: 0 phase: init*

The records below describe a READ IO fom and its state transitions. First record is a generic fom description; second record is specific to the fom type (READ request in our case), including information such as the object ID and the byte range on the component device to be read etc. Beware that as Mero parity de-clustering algorithm is used to store an object across multiple nodes and devices, analyzing ADDB records of multiple nodes and devices will give us more information such as if data is evenly distributed among devices etc.

** 2020-10-12-11:35:34.400332998 fom-descr service: <7300000000000002:1>, sender: 5389c54c7ff0120, req-opcode: M0_IOSERVICE_READV_OPCODE, rep-opcode: M0_IOSERVICE_READV_REP_OPCODE, local: false | node <f4eebce02dc43aa:a5e598c6672ae4dc> | pid 8394 | locality 0 | ast | fom @0x7ff5c4027830, 'io-fom' transitions: 0 phase: init*
** 2020-10-12-11:35:34.400335008 ios-io-descr file: <47000000d37e0b5d:dc0da2fa172cb354>, cob: <43000000d37e0b5d:dc0da2fa172cb354>, seg-nr: 1, count: 3000, offset: 0, descr-nr: 1, colour: 0| node <f4eebce02dc43aa:a5e598c6672ae4dc> |pid 8394 |locality 0 | ast |fom @0x7ff5c4027830, 'io-fom' transitions: 0 phase: init*

（2）Examples of sensors are:

** 2020-10-12-11:35:24.408306319 runq nr: 588 min: 0 max: 3 avg: 0.551020 dev: 0.312023 datum: 0 281 1: 307 3: 0 5: 0 7: 0 9: 0 11: 0 13: 0 15: 0 17: 0 19: 0 21: 0 23: 0 25: 0*

```
|         :      281 | **************************************************
|       1 :      307 | ************************************************************
|       3 :        0 |
|       5 :        0 |
|       7 :        0 |
|       9 :        0 |
|      11 :        0 |
|      13 :        0 |
|      15 :        0 |
|      17 :        0 |
```

```
|       19 :        0 |
|       21 :        0 |
|       23 :        0 |
|       25 :        0 |
|node <f4eebce02dc43aa:a5e598c6672ae4dc> | pid 8394 |locality  0

* 2020-10-12-11:35:24.408307451 wail            nr: 289 min: 0 max: 3 avg: 2.231834
dev: 0.655596 datum: 0  15 1: 274 3: 0 5: 0 7: 0 9: 0 11: 0 13: 0 15: 0 17: 0 19: 0 21: 0
23: 0 25: 0
|         :       15 | ***
|        1 :      274 | ************************************************************
|        3 :        0 |
|        5 :        0 |
|        7 :        0 |
|        9 :        0 |
|       11 :        0 |
|       13 :        0 |
|       15 :        0 |
|       17 :        0 |
|       19 :        0 |
|       21 :        0 |
|       23 :        0 |
|       25 :        0 |
|node <f4eebce02dc43aa:a5e598c6672ae4dc> | pid  8394|locality 0
```

These records are, respectively, total number of foms in the locality given by the context, number of foms in locality run-queue and locality wait-list.

It's worth to pointing out that Mero internally associates each fom with fop (File operation). Each fop represents a request sent from client (Clovis) to Mero services and an fop is assigned a unique reference number. This provides us with a way to link the Mero service side ADDB records to its Clovis ADDB records, which can be further connected to those MIO function calls (captured by selFIe) by carefully designing and logging MIO IO request details (such as object ID, MIO operation identifier and matching rules between MIO operation and Clovis ADDB operation records.). So it's feasible to trace the whole data path of an IO request from Maestro Core, down to MIO and further into Mero Clovis and services. This will provide us with much richer information and make many kinds of IO request analysis possible.

5. Conclusion & Next Steps

We have described in this deliverable some motivations for telemetry from an application perspective, and important example telemetry records for Maestro. We provide a basic working prototype of telemetry data collection enabled through selFIe and ADDB for the Maestro calls, MIO calls and Mero – which is the Maestro stack – implemented in a virtual environment for a simplistic workload [6].

References and source code links are given next.

6. References & "Source Code" Links

1. selFIe, https://github.com/cea-hpc/selFIe - maestro branch

2. ADDB Scripts, https://github.com/Seagate/cortx-motr

3. MIO, https://gitlab.version.fz-juelich.de/maestro/mio

4. Mero[3], https://github.com/Seagate/cortx

5. Maestro Deliverable, D5.1, "Telemetry Design"

6. Maestro Mero VM
https://gitlab.version.fz-juelich.de/maestro/maestro-mero-vm/-/tags/d5.4

[3] Mero is now open sourced fully under a "CORTX" brand by Seagate