MAESTRO
DATA ORCHESTRATION

D5.1 Telemetry Design

Work package:	**5 – Low level middleware**	
Author(s):	Nguyen Sai Sining Ganesan	Laurent Narasimhamurthy Wu Umanesan
Reviewer #1	Smart	Simon
Reviewer #2	El Sayed	Salem
Dissemination Level		
Nature		

Date	Author	Comments	Version	Status
05/01/2019	Laurent Nguyen			Draft
03/05/2019	Sai Narasimhamurthy	Mero	0.1	

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.

03/05/2019	Ganesan Umanesan	Mero	0.2	
22/08/2019	Sai Narasimhamurthy & Sining Wu	Overall Document	0.3, 0.4, 0.5, 0.6	
13/09/2019	Sai Narasimhamurthy	Document	0.7	
26/09/2019	Laurent Nguyen	Document	0.8	
29/09/2019	Sai Narasimhamurthy & Sining Wu	Document	0.9	

Executive Summary

This document provides architecture of the Maestro Telemetry framework that will be used to gather and analyse telemetry records from the different component pieces of the Maestro software stack. Telemetry data is stored in a Maestro Telemetry Database and data can be collected from various components associated with Maestro. In this document we exemplify data collections from MIO (Maestro I/O Interface), the Application-Maestro Interface, and the storage backend. The architecture can be generalised for use with various other telemetry data originating sources.

Contents

Glossary of Acronyms

Acronym	Definition
ADDB	Mero Analysis and Diagnostics Database
MIO	Maestro IO interface
selFIe	Self and Light proFIling tool Engine

1. Introduction

In this document, we describe the design of telemetry infrastructure in the Maestro project, and, present examples of telemetry data collection methods across the software stack associated with Maestro. The examples we focus on are the application-maestro interface, with data collected by a tool called “selFIe”, the backend Mero object store, through ADDB records and the Maestro I/O interface.

The key sequence of developments for telemetry design and development is:

- D5.1 (M09) – Design of overall framework and the extensions of selFIe and Mero to fit within the framework (This deliverable is extended to M15).
- MS5.1 (M20) – Telemetry available from all components for performance analysis
- D6.6 (M36) – Telemetry demonstration
- MS6.2 (M36) – Initial release of telemetry tools

Section 2 will discuss the overall framework for telemetry data handling and analysis in Maestro. Section 3 will discuss the selFIe tool used for collecting telemetry between the application and maestro middleware. Section 4 will discuss telemetry data collection from the backend object store, Mero. Section 5 discusses telemetry data collections from MIO. Section 6 discusses aspects of controlling the data volumes in the telemetry database. Section 7 provides the conclusions and next steps.

2. A Framework for Maestro Telemetry

The motivation behind the use of telemetry in Maestro is to gain a deeper understanding of the behaviour of the Maestro middleware when it is subjected to application workloads. Telemetry is observation data, which primarily includes data points on system performance and associated log messages, obtained during the operation of Maestro. The telemetry data is gathered from the various components and interfaces associated with Maestro. These different components/interfaces are:

1. The boundary of the applications/workflows and the Maestro middleware
2. The boundary of the Maestro middleware and the backend storage (Maestro I/O Interface, or, MIO)
3. The Maestro middleware (& its operations)
4. Backend storage[1]

[1] Please note that we do not at this point consider arbitrary inputs to Telemetry data base other than those listed here – within the scope of this project, though it is possible to extend this list.

(1) and (2) are gathered with the "selFIe" [5] tool for higher level workflows/applications that interface with the Maestro middleware, and also for the interface with backend storage. (4) is gathered from the ADDB (Analytics and Diagnostics Data Base) subsystem of the Mero Object store. Note that ADDB records or ADDBs are essentially structured telemetry records from Mero operations. (3) involves any additional data from the Maestro middleware, for example, from Core Data Object operations/movements etc. as described in the earlier Maestro project deliverable, D3.1 Initial Core Middleware specification, API document. The goal is to combine and extract meaning from these differing sets of interfaces and components to provide a holistic "story" of the timeline of the workflow. To that end we propose the Maestro Telemetry Database into which data is put from the different components and will be amenable for analytics.

The following are some of the main design considerations of Maestro Telemetry database:

1. Having a common schema[2] for managing all the different telemetry data streams.
2. Allocation of computational (& memory and storage) resources to manage and run the datastore/database services. At the moment, we consider a design where the service will be run external to Maestro (eg: close to storage services with storage nodes backing the database) so that it does not consume resources dedicated for Maestro operations.
3. Telemetry analysis during runtime or post mortem: Managing telemetry analysis during runtimes will be extremely challenging. Within the project, we focus on post mortem analysis.
4. Duplication of data in this database and elsewhere should be avoided as much as possible.
5. Leveraging commonly available Open Source analytics tool (eg: ElasticSearch) for post-mortem analysis.
6. The volumes of data that makes it to the Maestro telemetry database should be amenable for "dialling up/down" based on the granularity of telemetry data analysis needed, of course keeping scalability considerations in mind as job concurrency increases.
7. Possibility of defining new data sources for the telemetry database, generating data with the common schema and potentially having a common API for all telemetry data sources.

Figure 1a and Figure 1b below provide an overview of the telemetry infrastructure. Figure 1a shows the design that will guide the first implementation, Figure 1b shows a more

[2] As the number of components that generate telemetry data increases, it will be increasingly difficult to conform to a common schema. In that case, schema-less databases (eg: Graph Databases) may have to be looked at.

streamlined implementation for the future, potentially towards the end of the project based on what is learnt from the implementation of the First Version:

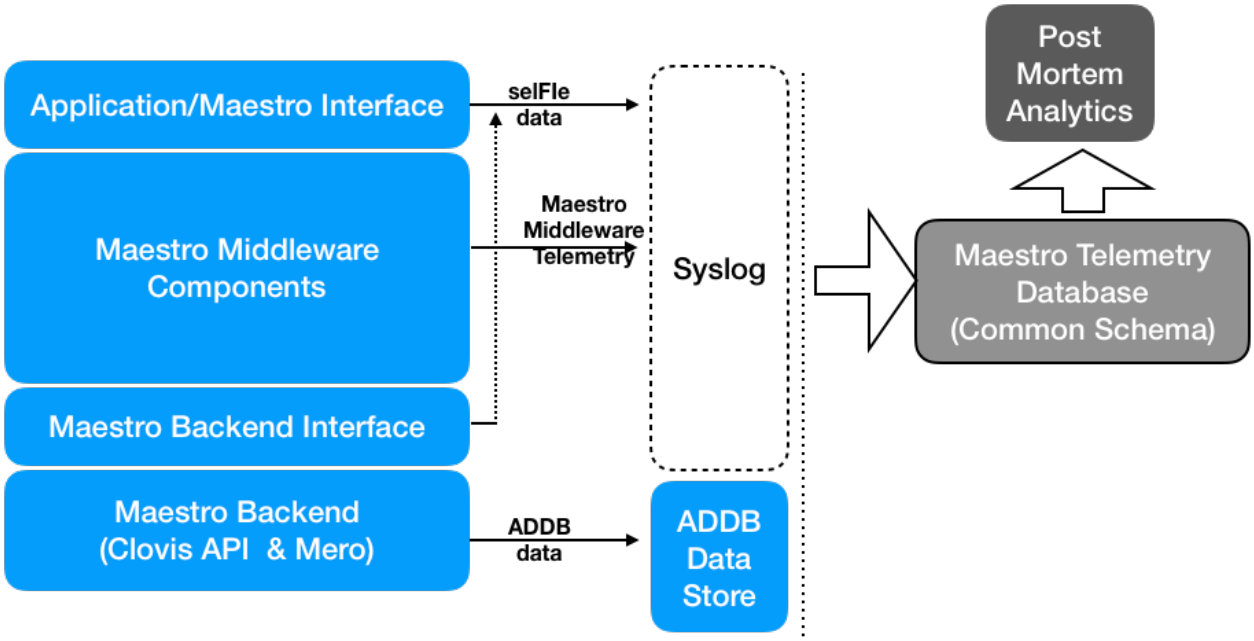

Figure 1a: Maestro Telemetry Infrastructure (First Version)

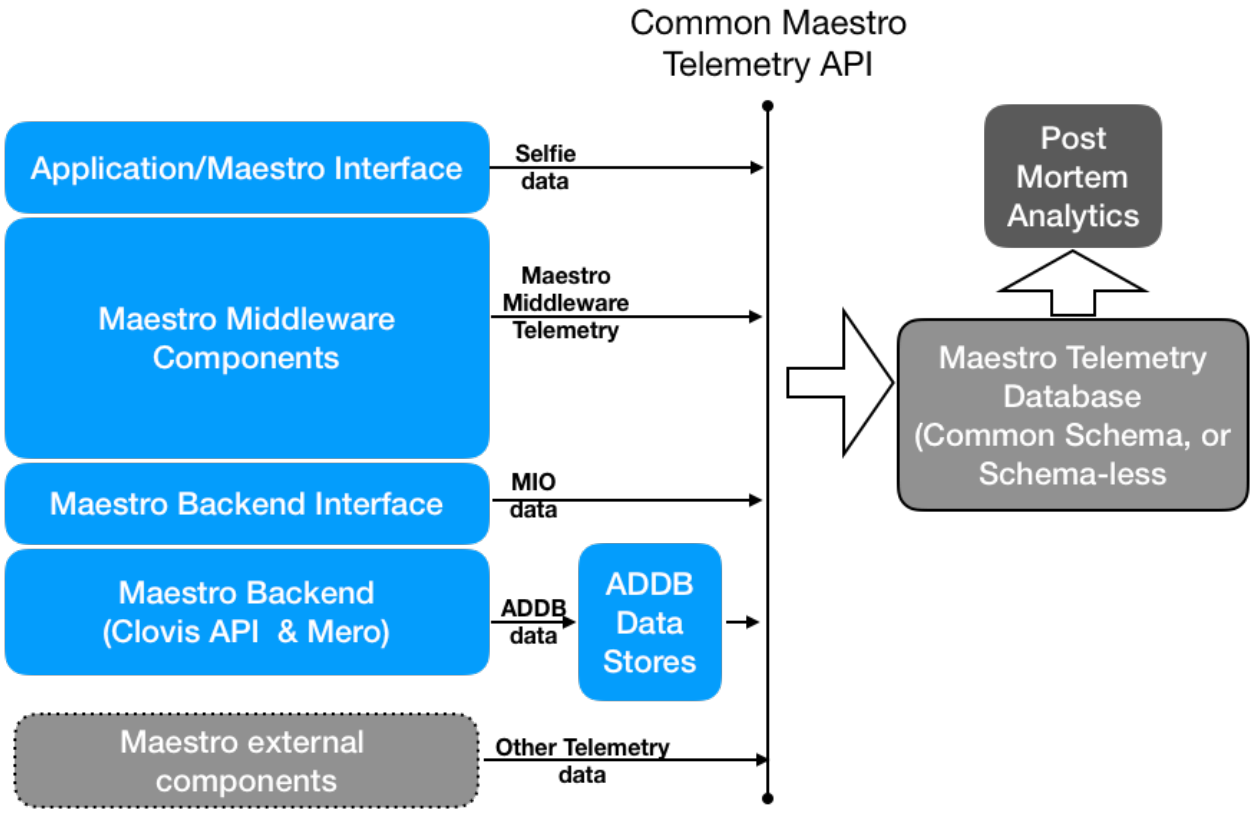

Figure 1a: Maestro Telemetry Infrastructure (Second Version)

As can be seen from Figure 1a, we will leverage the ability of the selFIe tool to obtain telemetry information from multiple interfaces (Application/Maestro and the Maestro IO Interface (MIO) in our case). selFIe data will initially be written to system logs. Data from the Maestro Backend will be in the distributed data stores which are part of the storage system (ADDB Data Store), i.e. in a location that is separate from the system logs. These disparate Telemetry data streams will be converted to a common schema and stored in an external Maestro Telemetry Database which can be made available for post mortem analysis.

A common format/schema will be co-designed as part of Maestro Core ("Maestro Middleware Components" in the above picture) logging subsystem. It will then provide the possibility (and flexibility) to associate and easily correlate telemetry records gathered from various sources, for example for all records belonging to a particular Maestro instance.

We note that there will be possibilities for avoiding data copying between the telemetry components into the system logs before they make it to the Maestro Telemetry Database. This is desirable as managing data between system logs and the Maestro Telemetry Database increases administrative overheads. These aspects will be considered in the next iteration of the design as shown in Figure 1b. Figure 1b also shows a possible

common API that can be adhered to by any Maestro associated component that generates telemetry. Figure 1b also shows a multiplicity of components generating telemetry data, which makes it increasingly difficult to adhere to a common schema. In that case a schema-less database (eg: Graph Database) to store all the telemetry data is also being assessed as a possibility. More details on the common schema will be provided in the prototype design and implementation deliverable. We are currently assessing all possible options on selecting the database (SQL, NoSQL, Graph, NewSQL, etc). Choice of the types of tools needed to extract information and analyse telemetry data in the Maestro telemetry database is also assumed to have achieved more clarity by then.

In this deliverable we will focus on the first version of the architecture and provide more details on the selFIe and Maestro Backend Interface/Maestro Backend telemetry data, which will be the first entities providing telemetry data into the Maestro Telemetry Database. We are currently working on:

1. Common Schema for the telemetry data & choice of the database
2. Maestro Middleware Telemetry
3. Full/detailed list of telemetry items that make it into the database
4. Choice of telemetry extraction and analysis tools

3. selFIe (self and Light proFIling tool Engine)

3.1 Presentation

selFIe [1] (SElfie and Light proFIling Engine) is a tool to lightly profile Linux applications without the need for recompiling the application. selFIe is inspired by the IPM [2] and Darshan [3] profiling tools.

selFIe is a dynamic library which can be given to the LD_PRELOAD environment variable before the execution of the application. The codes don't need to compile against selFIe.

selFIe does not affect the (functional) behaviour of the application and users do not see any changes during execution. At the end of the execution of an application, it puts one line in the system logs. This log line is in JSON format for post-processing.

selFIe can profile:

- MPI calls (Only C API)
- POSIX I/O calls
- PAPI hardware counters
- OpenMP zones

For MPI and POSIX I/O calls, selFIe collects the number of calls and time spent in these calls. selFIe collects volume data in case of certain POSIX I/O calls. Within this project we plan to add profiling of MIO calls.

selFIe profiles every executable called inside a job on a cluster. It handles MPI jobs as a set of independent launches of an application and does not aggregate the results within an MPI job. The results need to be post-processed to profile a whole job.

selFIe is an open source software that has been published under the CeCILL-C license and can be found here: https://github.com/cea-hpc/selFIe

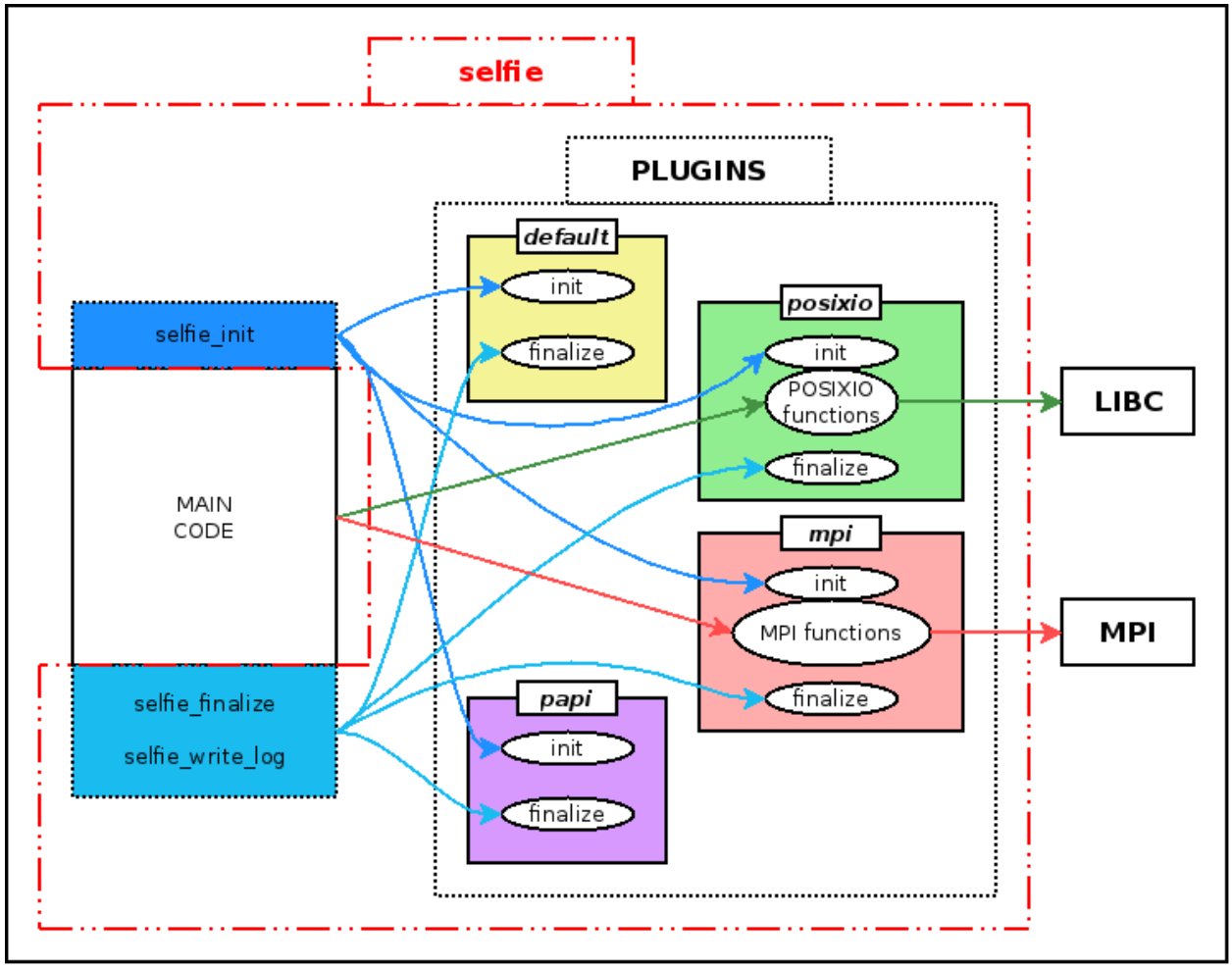

Figure 2 selFIe hooks

To use selFIe [5], the LD_PRELOAD environment variable should be initialized with the selFIe library:

```
$ LD_PRELOAD=selfie.so hostname
myhost
```

The usage is invisible to the user. After each execution, selFIe writes a line in JSON format in system logs like in the following example:

```
selfie[26058]: { "utime": 0.00, "stime": 0.01, "maxmem": 0.00,
"posixio_time": 0.00, "posixio_count": 7569, "USER": "user",
"wtime": 0.01, "command": "/bin/hostname" }
```

Depending on what the project needs, additional metrics can be added.

Provision of records from the MIO interface will be developed as part of the project.

3.2 Usage in Maestro

The usage of selFIe in Maestro project is to enable by default a lightweight profiling (or monitoring) for all jobs. selFIe will get IO profiling data to have a better understanding of where the time is spent during I/O (kernel space vs user space) and what is the impact of calls Maestro operations.

For Maestro selFIe will be extended as follows:

- Tracking calls within the Maestro middleware: Selfie will be modified to track MIO calls between Maestro and the storage backbends

- Tracking time spent in I/O calls in kernel space: Currently only the part of the time spent in I/O is tracked until an I/O call is posted to the kernel and therefore not the full time spent in POSIX I/O is captured (for example, the time needed by the kernel to send data to a parallel file system).

4. Mero

One of the observations in debugging large scale systems with distributed storage and I/O is that it is extremely hard to get a deeper understanding of what is happening in the backend storage, which is typically decoupled from the computing systems. This has been learnt through our experiences with file systems such as Lustre and GPFS where system administrators have to go through rather unstructured system logs. It is extremely hard to correlate I/O problems with overall systems issues. We hence feel it necessary to have more information and structure of the backend I/O telemetry data and make it part of the overall Maestro telemetry framework and will develop tools and methods to gather telemetry data from the Mero backend infrastructure.

The granularity of telemetry data can be tuneable (primarily by the system administrators) as needed by Maestro. The telemetry records could be obtained from the Clovis interface both at runtime and after the job completes, though we focus on analysis of records post job completion at this time. The telemetry records help to provide a better view of the

overall Mero system as well as the monitoring of the performance and I/O access statistics of the individual tasks. The telemetry records are based on very rich structured telemetry records, which are very amenable for data analytics. Telemetry records will be used for post mortem analysis in the project, though real-time analysis is also a possibility.

Mero is only one of backend for Maestro. Our design is such that also other object storage backends can be supported.

Introduction to ADDB

Mero introduces the concept of ADDB (Analysis and Diagnostics DataBase) records, which are well defined structured records that can be collected and analysed a lot more systematically than unstructured logs. These telemetry records are generated by various subsystems within Mero and their generation is always enabled. The record volumes can be dialled up or down based on the needs of the analysis.

The ADDB records are generated in the form of instrumentation points with timestamp data or as contexts of operations. The ADDB record analysis and visualization allows for a quick drill down of the problems within the system. Statistical analysis such as anomaly detection and root cause analysis can be performed quickly on top of these ADDB records using standard analysis tools, which is extremely difficult with unstructured logs. The following examples depict how ADDB records were analysed and visualised to identify anomalous nodes and anomalous time periods - using simple instrumentations captured within the Mero nodes.

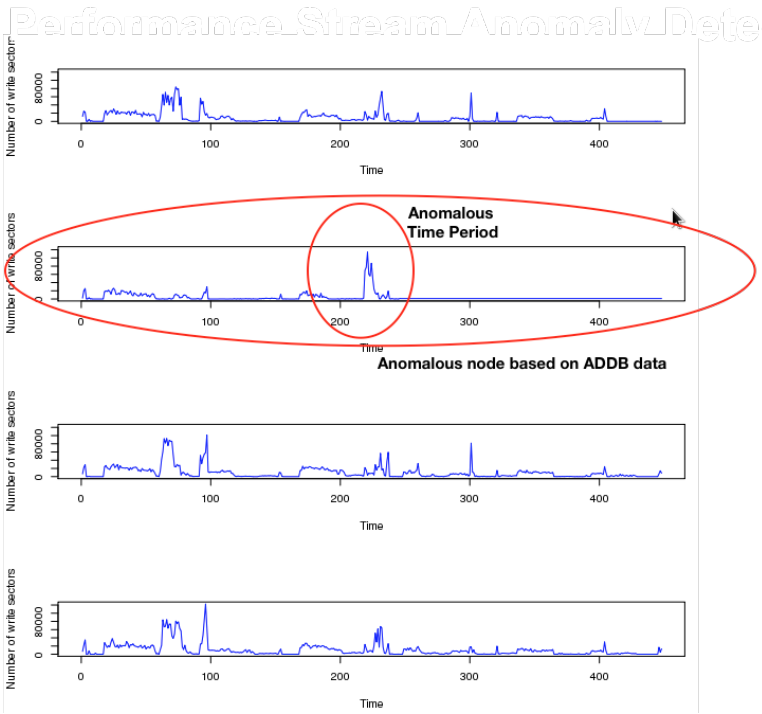

Figure 3 Anomalies based on ADDB data - Example 1

Note that anomalous periods could be a result of either system anomalies or just short periods of bursty I/O activity. In the following example, more analysis is needed to identify differences in the behaviour:

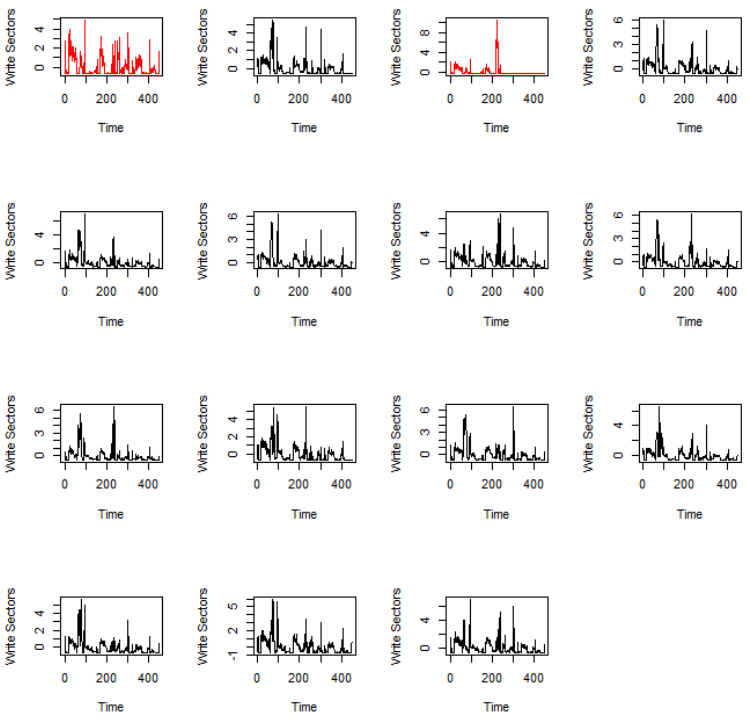

Figure 4 Anomalous nodes based on ADDB data - Example 2

We next provide more details on the structure of the ADDB records and continue to describe how these ADDB records will be exploited in the Maestro project to provide a full picture of the system, which includes the Maestro backend, Mero.

ADDB Data Structure

Mero's ADDB data structure provides values, labels and records.

```
struct m0_addb2_value { /** value: a timestamped set of 64-bit words. */
      uint64_t        va_id;                          /** Value identifier. */
      uint64_t        va_time;                        /** Local timestamp in nanoseconds from epoch. */
      unsigned        va_nr;                          /** Number of words in the payload. */
      const uint64_t *va_data;                        /** Payload. */
};
struct m0_addb2_record { /** record: a measured value and a context. */
      struct m0_addb2_value ar_val;                    /** Measurement. */
      unsigned              ar_label_nr;               /** Number of labels in the context. */
      struct m0_addb2_value ar_label[M0_ADDB2_LABEL_MAX]; /** Labels. */
};
```

An example of a simple ADDB record is shown below:

```
* 2015-04-20-14:36:13.687531192 alloc size: 40, addr: @0x7fd27c53eb20
|  node             <f3b62b87d9e642b2:96a4e0520cc5477b>
|  locality         1
|  thread           7fd28f5fe700
|  fom              @0x7fd1f804f710, 'IO fom' transitions: 13 phase: Zero-copy finish
|  stob-io-launch   2015-04-20-14:36:13.629431319, <200000000000003:10000>, count: 8, bvec-nr: 8, ivec-nr: 1, offset: 0
|  stob-io-launch   2015-04-20-14:36:13.666152841, <100000000adf11e:3>, count: 8, bvec-nr: 8, ivec-nr: 8, offset: 65536
```

The data structure allows the creation of a more sophisticated ADDB records with sensory information which can then be visualized in a histogram for example. In the following example the underlined numbers 1, 3, 5, etc., represents the value of different sensors. The numbers next to these numbers represents the number of times this particular sensor was observed. For example, underlined sensor value 5 indicates that the queue length was 5 and the number 98 tells us that it was observed 98 times, i.e., the queue length was 5 a total of 98 times.

```
* 2018-04-05-14:27:40.563070315 fom-active nr: 710 min: 0 max: 24 avg: 5.029577 dev: 11.147012
  1 1: 226 3: 313 5: 98 7: 26 9: 20 11: 8 13: 5 15: 6 17: 4 19: 3 21: 0 23: 0 25: 0
|         node             <5e341757d0cf46eb:92a6d6e991b46387>
|         pid              6627
|         locality         0
```

This information can be represented in a histogram to show that some events are more frequent than the others. The example histogram below shows the value of sensors on the first column, the frequency in the second column and finally a visual representation of the frequency in the third column.

Please note that providing such a basic structure to telemetry records is very useful for performance data analysis. This is a big improvement over system administrators trying to look over unstructured system logs to comprehend performance issues.

```
   :   1 |
 1 : 226 | **********************
 3 : 313 | ******************************
 5 :  98 | **********
 7 :  26 | ***
 9 :  20 | **
11 :   8 | *
13 :   5 |
15 :   6 |
17 :   4 |
19 :   3 |
21 :   0 |
23 :   0 |
25 :   0 |
```

Within the ADDB infrastructure, execution context information can be added or removed by using the following functions.

```
/**
 * Adds a label to the current context.
 */
void m0_addb2_push(uint64_t id, int n, const uint64_t *value);

/**
 * Removes the top-most label in the current context stack.
 */
void m0_addb2_pop (uint64_t id);

/**
 * Adds one-time measurement in the current context.
 */
void m0_addb2_add (uint64_t id, int n, const uint64_t *value);
```

Mero threads including application threads that have initialised a Clovis API instance maintains internally an ADDB machine where context is incrementally built. A machine keeps track of the current context, maintained as a stack. New labels can be added to a context by a call to m0_addb2_push() and the top-most label can be removed by a call to m0_addb2_pop().

m0_addb2_push(): this operation adds a label to the current context. A label is specified as a 64-bit value and a payload, consisting of a given number of 64-bit values. Number of values in the payload must not exceed a pre-set number.

m0_addb2_pop(): this operation removes the top-level label from the current context and adds 64-bit constant POP to the current trace buffer.

An example of how m0_add2 push() and m0_addb2_pop() are used in MIO to tag ADDB records which are produced by the Maestro thread calling MIO APIs is below.

```
int mio_obj_writev(struct mio_obj *obj, const off_t *offsets,
                   const struct mio_iovec *iov, int iovcnt,
                   struct mio_op *op)
{
    /**
     * MIO_OBJ_WRITE is the label tagged to ADDB records
     * that are generated synchronically in the thread between
     * m0_addb2_push() and m0_addb2_pop(). To enable grouping ADDB
     * records to each WRITE, a WRITE sequence number is added as
     * its payload.
     */
    m0_addb2_push(MIO_OBJ_WRITE, 1, mio_write_seqno);

    /* Real mio_obj_writev() code starts here. */
    ...

    /* Pop the label set at the beginning of mio_obj_write(). */
    m0_addb_pop();
}
```

The following shows a list of currently available record types in Mero.

NODE	node
LOCALITY	locality
PID	pid
THREAD	thread
SERVICE	service
FOM	fom
CLOCK	clock
PHASE	fom-phase
STATE	fom-state
ALLOC	alloc
FOM_DESCR	fom-descr
FOM_ACTIVE	fom-active
RUNQ	runq
WAIL	wail
AST	ast
LOCALITY_FORQ_DURATION	loc-forq-duration
LOCALITY_FORQ	loc-forq-hist
LOCALITY_CHAN_WAIT	loc-wait-hist
LOCALITY_CHAN_CB	loc-cb-hist
LOCALITY_CHAN_QUEUE	loc-queue-hist
CAS_KV_SIZES	cas-kv-sizes
LONG_LOCK	long-lock
SIT	sit

IOS_IO_DESCR	ios-io-descr
FS_OPEN	m0t1fs-open
FS_LOOKUP	m0t1fs-lookup
FS_CREATE	m0t1fs-create
FS_READ	m0t1fs-read
FS_WRITE	m0t1fs-write
FS_IO_DESCR	m0t1fs-io-descr
FS_IO_MAP	m0t1fs-io-map
STOB_IO_LAUNCH	stob-io-launch
STOB_IO_END	stob-io-end
STOB_IOQ	stob-ioq-thread
STOB_IOQ_INFLIGHT	stob-ioq-inflight
STOB_IOQ_QUEUED	stob-ioq-queued
STOB_IOQ_GOT	stob-ioq-got
RPC_LOCK	rpc-machine-lock
RPC_REPLIED	rpc-replied
RPC_OUT_PHASE	rpc-out-phase
BE_TX_STATE	tx-state
BE_TX_COUNTER	BE transactions
BE_OP_COUNTER	BE operations
NET_BUF	net-buf
FOP_TYPES_RANGE_START	fop transitions

Mero's ADDB records are fine-grained and provide an always-on telemetry stream (megabytes per second) stored persistently in Mero servers. The records can be used by tools online as well as offline. The architecture is scalable and provides a publish-subscribe interface so that third party tools can extract records that are interesting to them for debugging, profiling and monitoring.

ADDB Usage

The instrumentation provided by ADDB records (at the bottom of the I/O stack at the level of the Maestro backend, Mero) help to provide a full picture of the behaviour and operation of the backend storage system. As part of the implementation:

- We will provide a mechanism to make ADDB records available for the Maestro Telemetry Database.
- We will identify ADDB records of interest for Maestro, and create new ADDB records as necessary.
- We will also provide methods to capture ADDB records related to specific I/O calls of interest to Maestro.
- At the moment the ADDB interface just records data from the data servers. Methods to generate and correlate these ADDB records with specific Clovis I/O calls on the Mero client side and these records will be made part of the Maestro Telemetry Database.

5. Combining MIO and ABBD telemetry data

We next describe how the data from MIO can be combined with ADDB records.

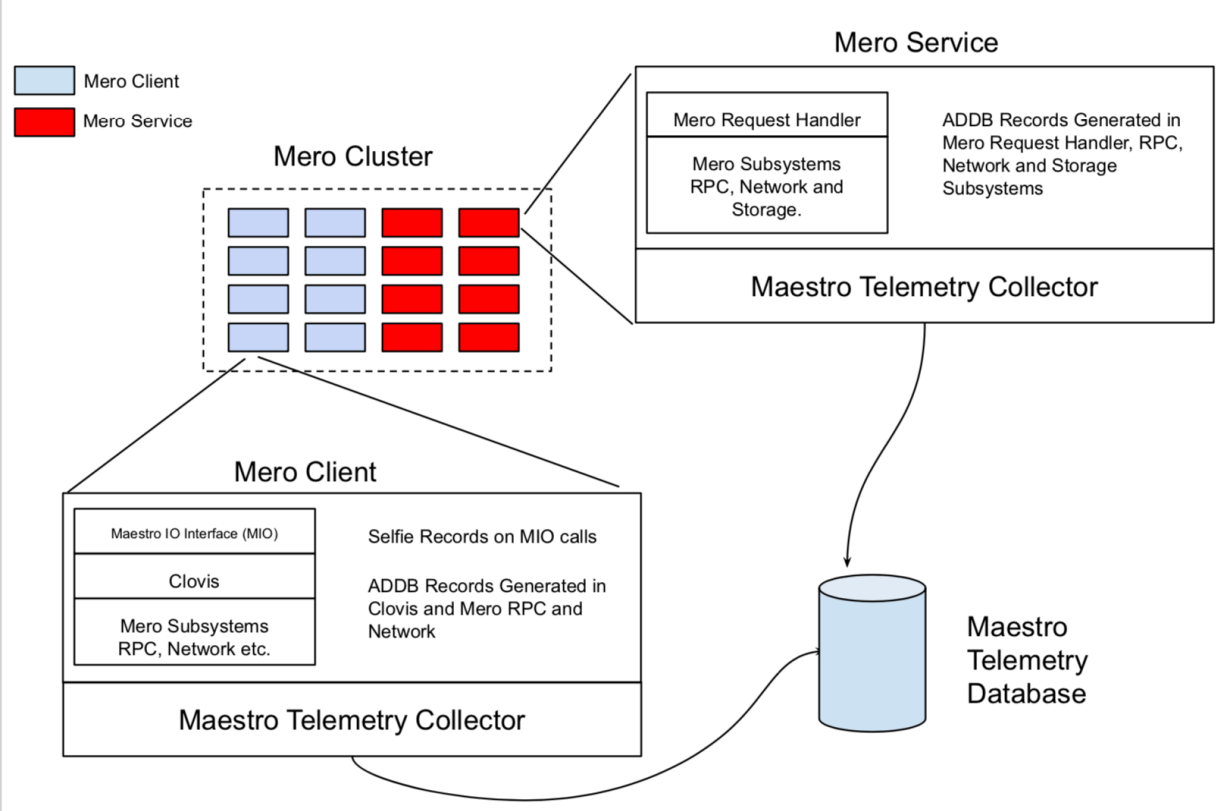

Figure 5 selFIe hooks

Figure 5 illustrates a typical Mero cluster running together with an Maestro instance. We use it to explain what kinds of telemetry data can be collected from Mero and how they are collected:

- Maestro middleware interacts with Mero via the Maestro I/O Interface (MIO) on Mero client nodes. MIO internally relays all requests on object access and key-value store query to the Mero API, Clovis which subsequently sends requests to Mero services via RPC and network subsystems. MIO API calls encapsulate details of Maestro middleware I/O requests and can provide insights of Maestro middleware I/O behaviour. As MIO will be presented as an I/O library, selFIe can be adapted to collect MIO APIs' information. Besides MIO, Clovis and Mero subsystems on the client nodes also generated rich ADDB records on Clovis, RPC and network works.
- Mero services serve requests, access and store data on devices. Mero services have been instrumented to generate ADDB records to help understand the workload of request handlers (such as request queue length) and I/O accesses to storage devices.
- ADDB records are stored on a special storage device in a Mero-specific format. Mero provides utilities to collect ADDB records from nodes. A Maestro telemetry collector is deployed on each node to decode and filter ADDB records, translate them into Maestro telemetry format and store them in telemetry database for future queries and analysis.

6. Controlling information in telemetry database

To avoid the Maestro Telemetry Database being overwhelmed with information from the different components producing telemetry data, mechanisms are being assessed to control the information that is fed into the database, thereby achieving trade-offs between granularity of telemetry analytics and system resource utilization.

Considering the data sources of telemetry data:

(1) For Mero ADDB, since one of the design goals is to have as little overhead as possible, it has mechanisms such as ADDB sensors to summarize measurements in a concise way to reduce data volume produced. ADDB is however designed to be always on at the moment and lacks a way to easily switch off those measurements we are not interested in (though we can summarise), at least in the on-going implementation. We will study further, the resource utilization aspects of Mero ADDB telemetry and assess providing “knobs” to switch off certain ADDB telemetry records after we start the deployment of the feature

(2) For selFIe records generated from instrumenting at library interfaces, mechanisms such as filtering, sampling and serialization can be considered to reduce the amount of data.

(3) Once Maestro core provides common APIs for generating telemetry records, similar mechanisms such as those for selFIe can be considered.

7. Concluding remarks

This document describes the overall framework for Maestro telemetry, with a Maestro Telemetry Database, and the design of two tools (SelFIe and ADDB) as part of the Maestro telemetry framework. The two components will be further developed so that we will be able to track all the telemetry records related to an application run at the Application/Maestro and Maestro/Backend interfaces, and within the Maestro Backend – as described. Design of the additional data stream, primarily from the Maestro middleware itself, that make it to the Maestro Telemetry Database, along with a detailed list of all items that make it to the telemetry database is work in progress. Design of a common telemetry API is also under consideration.

8. References

[1] selFIe, https://github.com/cea-hpc/selFIe, Accessed November 2019

[2] IPM, http://ipm-hpc.sourceforge.net/, Accessed November 2019

[3] Darshan, https://www.mcs.anl.gov/research/projects/darshan/, Accessed November 2019

[4] Elasticsearch Logstash Kibana, https://www.elastic.co/elk-stack, Accessed November 2019

[5] selFIe README, https://github.com/cea-hpc/selFIe/blob/master/README.md, Accessed November 2019