Chapter 9. ExaMon: Exascale Holistic Monitoring

Francesco Beneventi[1], Antonio Libri[2], Andrea Bartolini[1], and Luca Benini[1,2]

[1] *Università di Bologna, Italy*
{francesco.beneventi, a.bartolini}@unibo.it

[2] *ETH Zurich, Switzerland*
{antonio.libri, lbenini}@iis.ee.ethz.ch

Abstract

EXAMON, which stands for EXAscale MONitoring, aims to develop a portable and extensible monitoring framework at application-level that gives the possibility to the application to inspect extra-functional properties (such as, energy) instead of only raw architecture-specific metrics (such as low level values coming from HW counters). The monitoring framework is also developed to support the monitoring of the runtime behaviour of the system. The goal is to continuously and dynamically collect data from the system to make them available to applications and management layers. Besides traditional feedback on HW performance and throughput, novel kinds of scalable monitors, specific for HPC systems, is developed to provide feedback about performance, energy efficiency and thermal efficiency. This will be achieved by designing monitor blocks, as well as the data collectors, which observe application execution and phases, and which are able to detect patterns / signatures of the instructions. Furthermore, access patterns to the instruction and data cache / memory are observed to detect possible optimization of the memory allocation. Proper interfacing SW driver layers are developed for the target platforms to communicate data and events to applications as well as APIs to propagate application events to the collectors. The solution leverages big-data infrastructure to support the exascale monitored data flow.

1 The Hardware Monitoring "EXAMON" Framework

Examon is a highly scalable monitoring framework suitable for integration in a more complete framework in a production environment. It aims to provide performance and energy monitoring of the computing nodes.

Overall the monitoring system is composed by two main parts:

- Back-end: this part of the framework groups all the software components that execute on the computing nodes of the monitored cluster. These components are mainly system services (daemons) that collect data from the sensors available on the nodes and publish them on the network. In the ANTAREX project, we developed the pmu_pub system service. It is a customized hardware monitor, tailored for the Intel based homogenous computing nodes that are available in the CINECA and IT4I supercomputers.

- Front-end: this part groups all the components that consume the data generated by the back-end monitoring daemons. They mainly take care of the data storage, visualization, analysis and integration with the application, autotuner and domain specific language.

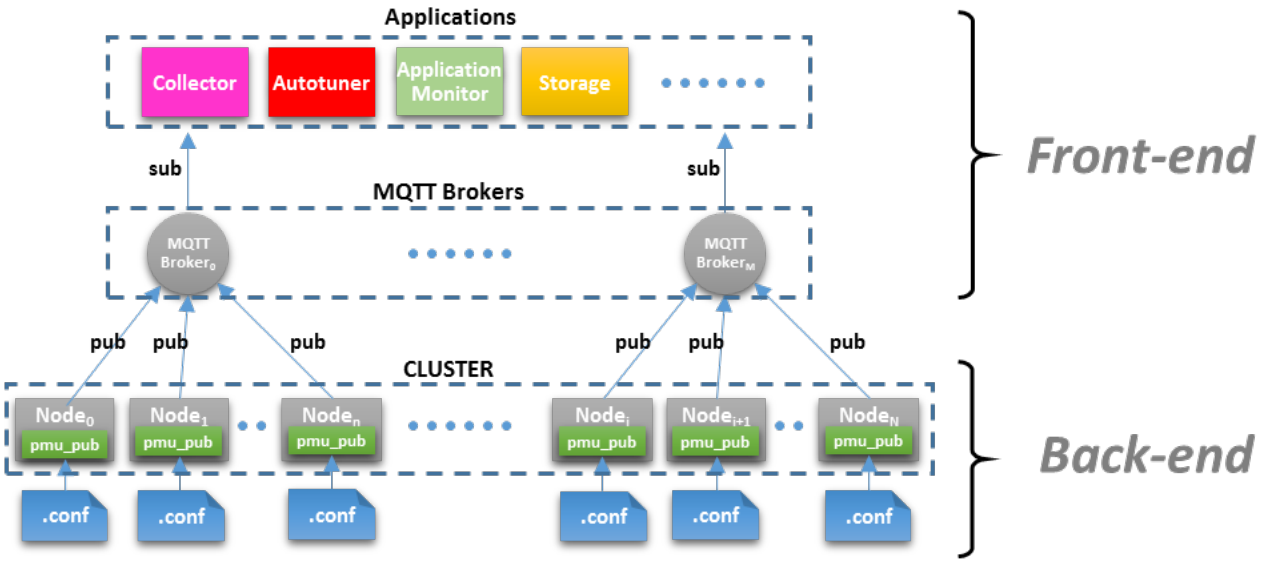

Figure 1: Examon Monitoring Framework tailored for the ANTAREX project

The framework leverages MQTT as a data transport protocol. MQTT (MQ Telemetry Transport) is a lightweight messaging protocol designed for applications with limited network bandwidth. It is built on top of TCP/IP and is used mainly in M2M (Machine-To-Machine) communications and more generally in IoT (Internet of Things) applications. It is well suited for the transmission of sensor data coming from devices having small processing capabilities and large network latencies.

MQTT implements the "publish-subscribe" messaging pattern and requires three different agents to work:

1. The "publisher", having the role of sending data on a specific "topic"
2. The "subscriber", that receives data from a "topic"
3. The "broker" that is responsible for:
   (a) receiving data from publishers
   (b) making topics available to subscribers
   (c) delivering data to subscribers

The basic communication mechanism is as follow. As soon as a publisher agent starts to send data tagged with a certain topic name, a topic is created and available on the broker. Any subscriber attached to the same topic will receive the flow of data associated to it. In the Examon framework, the software component that acts as a "broker" executes in the Front-end. The plugins that execute in the back-end side behave as "publishers" agents. Finally, all the remaining components on the Front-end behave as "subscribers".

2 DiG: high resolution power monitoring extension

DiG (a.k.a. *Dwarf in a Giant*) is the power monitoring extension for ExaMon that we use to measure the power consumption of HPC systems and data-centers at high resolution. DiG

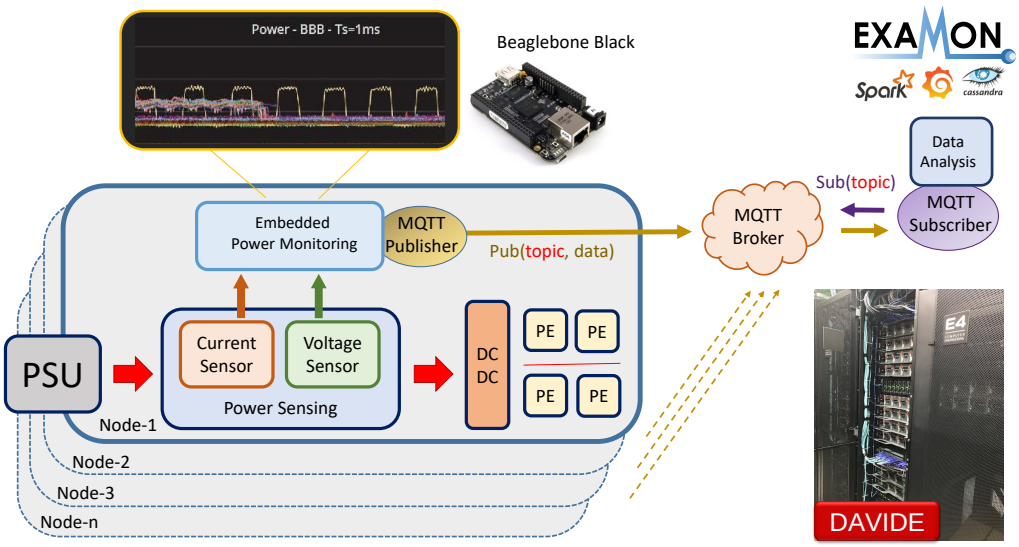

Figure 2: DiG overview.

consists of a set of agents running outside the computing resources of the nodes, but tightly coupled with them.

Figure 2 shows is architecture: we use (i) a power sensing board to measure the power consumption at the plug of computing nodes and (ii) a dedicated embedded platform (namely the Beaglebone Black - BBB) to sample, pre-process and send data via MQTT to ExaMon. DiG can measure the power consumption at $20\mu s$ of time resolution with a precision below 1% (1σ). Moreover, the sampled data are synchronized at microsecond scale (which is below the sampling period) thanks to the Precision Time Protocol (PTP), that is supported in hardware in the BBB. Finally, the monitoring extension is technology agnostic (*i.e.*, already tested in different architectures, such as Intel, ARM and IBM) and low cost (*i.e.*, the custom power sensor does not require any motherboard redesign). In particular, it is already installed in D.A.V.I.D.E., which is a 45-nodes computing cluster, based on OpenPower IBM Power8 and developed by E4 Computer Engineering for PRACE - currently hosted in CINECA and ranked #18 in Top500 Nov'2017 -, with the aim to providing a leading edge HPC cluster with high performance and reduced power consumption.

3 The "pmu_pub" plugin

Pmu_pub is the framework component developed for the ANTAREX project that is in charge of measuring and delivering the CPU sensor data. It runs as a system service (daemon) that regularly samples and provides the following sensors to the broker:

Pmu_pub is the framework component developed for the ANTAREX project that is in charge of measuring and delivering the CPU sensor data. It runs as a system service (daemon) that regularly samples and provides the following sensors to the broker:

1. per-core performance counters
   (a) Instructions retired
   (b) Un-halted core clock cycles at the current frequency

(c) Un-halted core clock cycles at the reference frequency
(d) Temperature
(e) Time stamp counter
(f) Cycles in C3 state
(g) Cycles in C6 state
(h) Aperf Cycles
(i) Mperf Cycles
(j) Programmable PMU events

2. per-CPU/Socket

(a) Package temperature
(b) Package energy
(c) DRAM energy
(d) Programmable Uncore events

3.1 Source code repository

The pmu_pub plugin is part of the Examon framework. A copy of the source code can be downloaded from https://github.com/fbeneventi/examon. The repository is structured as follow:

- publishers: this folder contains the MQTT publishers' plugins.
  - this release contains the pmu_pub plugin.
- parser: this folder contains the software components that run in the Front-end of the framework and process MQTT data delivered by the publishers
  - The pmu_pub_sp.py script provides an example of how to calculate additional metrics in real time, starting from the data delivered by the pmu_pub plugin. This component is designed to execute on the Front-end of the framework. By calculating the additional metrics outside the computing nodes, it helps to get the lowest impact on the user applications that use the pmu_pub plugin and that run simultaneously on the same nodes.
- collector: it contains the Collector component. See below for a detailed description.
- lib: this folder contains external libraries needed by the framework.

3.2 Installation

3.2.1 Dependencies

Pmu_pub requires three libraries to work:

1. Iniparser: used to handle the configuration files (.conf)
2. Mosquitto: used for the MQTT protocol

3. libpfm-4: used to program the performance monitoring events

(These libraries are provided in the "./lib" folder). To properly build the Mosquitto library you also need:

1. libssl
2. libcrypto

which are available in the following distro packages:

1. "libssl-dev" in Ubuntu/Debian
2. "openssl-devel" in Centos

3.2.2 Build

To build all the libraries and the main executable "pmu_pub", go to the main directory and:

```
>$ make
```

3.2.3 Install

WARNING: To install the plugin binary only (and excluding the libraries) DO NOT execute make install in the main directory but move in the plugin directory first:

```
>$ cd ./publishers/pmu_pub
```

Create and edit the configuration files (see the Configuration section for details):

```
>$ cp example_pmu_pub.conf pmu_pub.conf
>$ cp example_host_whitelist host_whitelist
>$ make install
```

The default install folder is ./bin. To specify a different installation folder:

```
>$ make PREFIX=<install-dir> install!
```

The installation steps will copy the executable, the "pmu_pub.conf" file and the "host_whitelist" file to the <install-dir>.

3.2.4 Configuration

The main executable needs at least the "pmu_pub.conf" properly configured. If available, it uses also the "host_whitelist" file to detect the host mane in which being instantiated.

The executable will search for the "pmu_pub.conf" file and the "host_whitelist" file in the current working folder first and then, if not found, in the "/etc" folder.

The "pmu_pub.conf" file in the ./publishers/pmu_pub directory contains the default parameters needed by the "pmu_pub" executable.

1. MQTT parameters:
   (a) brokerHost: IP address of the MQTT broker
   (b) brokerPort: Port number of the MQTT broker (1883)
   (c) topic: Base topic where to publish data (usually it is built as: org/<organization name>/cluster/<cluster name>)
2. Sampling process parameters:

(a) dT: data sampling interval in seconds

(b) daemonize: Boolean value to daemonize or not the sampling process

(c) pidfiledir: path to the folder where the pidfile is stored

(d) logfiledir: path to the folder where the logfile is stored

The "pmu_pub.conf" file must be in the working directory of the executable.
However, most of the parameters can be overridden, when executed, by command line:
`>$ sudo ./pmu_pub -h`

```
usage: pmu_pub [-h] [-b B] [-p P] [-t T] [-q Q] [-s S] [-x X]
                     [-l L] [-e E] [-c C] [-P P] [-v]
                    {run,start,stop,restart}
positional arguments:
 {run,start,stop,restart}
                       Run mode
optional arguments:
 -h                    Show this help message and exit
 -b B                  IP address of the MQTT broker
 -p P                  Port of the MQTT broker
 -s S                  Sampling interval (seconds)
 -t T                  Output topic
 -q Q                  Message QoS level (0,1,2)
 -x X                  Pid filename dir
 -l L                  Log filename dir
 -c C                  Enable or disable extra counters (Bool)
 -e E                  Perf events list (comma separated)
 -P P                  Enable or disable perf subsystem (Bool)
 -v                    Print version number
```

The "host_whitelist" file is an optional file that contains the list of the hosts in which pmu_sub is allowed to execute. If this file is not present, every host is enabled by default.The hostnames which are allowed to execute the plugin are listed row by row. Optionally, it can be included a broker IP address where to connect just a group of hosts. This is useful for example to balancing the load/bandwidth in the front-end nodes.

The proper file format is:

```
[BROKER:] <IP address> <port number>
host0
host1
host2
```

To disable a host or a group of hosts, please use "#" as a general comment marker. Here is an example of the host_whitelist file:

```
[BROKER:] 192.168.0.1 1883
node100
node101
```

```
[BROKER:] 192.168.0.1 1884
#node102
node103
```

There are 4 total hosts and 2 brokers. node100 and node101 will connect to the broker at 192.168.0.1:1883. node102 and node103 will connect to the broker at 192.168.0.1:1884. Host "node102" is disabled so the plugin will not run.

3.2.5 Compatibility

The current implementation of "pmu_pub" is developed and tested on Linux systems with kernel version 3.10. In systems with Linux kernel version greater than 4.0 are needed the following supplementary steps:

1. according to this documentation[1] , set the following parameter:

   ```
   >$ echo 2 > /sys/devices/cpu/rdpmc
   ```

2. it is advisable to execute the pmu_pub process with the full perf support:

   ```
   >$ sudo ./pmu_pub -P 1
   ```

3.2.6 Usage

The following instructions indicate how to build a single node measuring setup composed by:

- A broker used as an endpoint where to send and ask for the CPU data.
- A publisher agent that collects and publishes CPU data to the broker.
- A subscriber agent that receives the CPU data.

In this usage example, we deploy the framework on the simple scenario illustrated in Figure 3, which consists of:

- The computing node that we want to monitor, with "TESTNODE" as a hostname and "TESTNODE_IP" as IP address.
- The service node where the Front-end components of the framework are running. It has "SERVICENODE" as a hostname and "SERVICENODE_IP" as IP address.
- An external node, "USERNODE" that subscribes to the broker and asks for data.

Setup steps:

1. On SERVICENODE, **build** the Examon package. Then run the broker process as a daemon:

   ```
   >$ ./lib/mosquitto-1.3.5/src/mosquitto -d
   ```

2. On TESTNODE, **install** the Examon package. Edit the "pmu_pub.conf" file, and set at least the following parameters:

[1] http://man7.org/linux/manpages/man2/perf_event_open.2.html (section: rdpmc instruction)

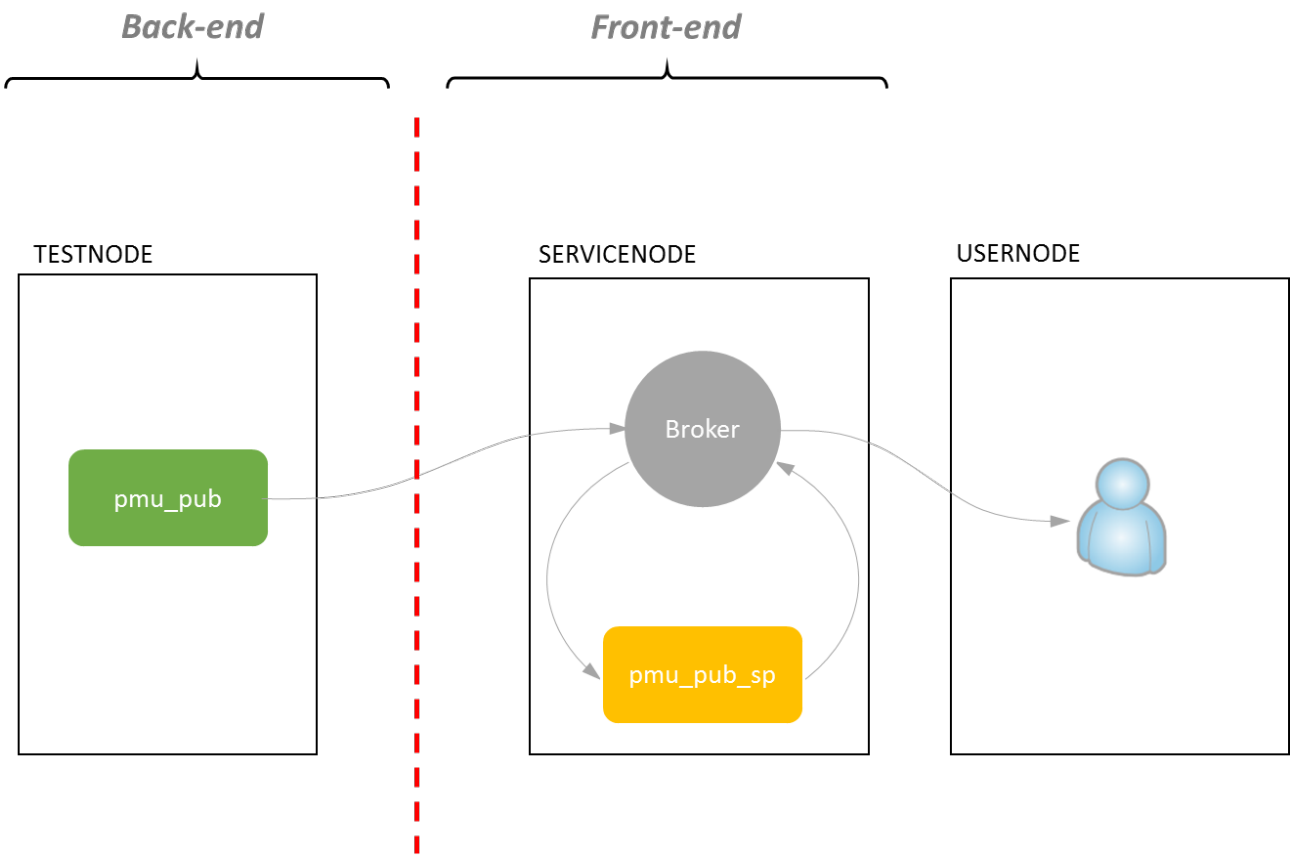

Figure 3: Usage scenario

(a) brokerHost: IP address of the node where the broker is running. In this example set it to SERVICENODE_IP.

(b) topic: set it to "org/antarex/cluster/testcluster"

3. On TESTNODE, make sure that the msr driver is loaded:

```
>$ sudo modprobe msr
```

4. On TESTNODE, **run** the pmu_pub process (publisher) as superurser. Please, remember first to move the pmu_pub directory, "cd ./publishers/pmu_pub/, and then run:

```
>$ sudo ./pmu_pub
```

The service runs silently in the background with no outputs. At this point, the CPU data should be available to the broker at the topic indicated in the .conf file.

5. On USERNODE, **build** the Examon package. It is possible to redirect the data stream to the shell or to a file. An MQTT subscriber client is available in the "./lib/mosquitto-1.3.5/client" folder. Assuming the broker is running at the IP address SERVICENODE_IP, the following command (executed on USERNODE) will print the data published by the sampling process "pmu_pub" to the standard output:

```
>$ LD_LIBRARY_PATH=../lib/:$LD_LIBRARY_PATH \
     ./mosquitto_sub -h SERVICENODE_IP \
  -t "org/antarex/cluster/testcluster/#" -v
```

or, to saving to a file:

```
>$ LD_LIBRARY_PATH=../lib/:$LD_LIBRARY_PATH \
    ./mosquitto_sub -h SERVICENODE_IP \
 -t "org/antarex/cluster/testcluster/#"  -v >> cpudata.log
```

6. On SERVICENODE, in order to calculate additional metrics, please move to the "./parser/pmu_pub_s folder, and execute:

```
>$ python ./pmu_pub_sp.py \
-b 127.0.0.1 \
-p 1883 \
-t org/antarex/cluster/testcluster/node/TESTNODE/ \
        plugin/pmu_pub/chnl/data \
-o org/antarex/cluster/testcluster/node/TESTNODE/  \
        plugin/pmu_pub/chnl/data \
start
```

7. On TESTNODE, in order to kill the sampling process, please move to the "./publishers/pmu_pub" folder and execute:

```
>$ sudo ./pmu_pub stop
```

On SERVICENODE, to kill the pmu_pub_sp process, please move to the "./parser/pmu_pub_sp" folder and execute:

```
>$ python ./pmu_pub_sp.py stop
```

On SERVICENODE, to kill the broker process, please execute:

```
>$ killall mosquitto
```

4 Collector

The collector is the software component which allows the Examon monitoring framework to be integrated with both the Autotuner (designed in T3.2) and the Lara DSL (designed in WP2). The goal of the collector is to aggregate desired metrics (e.g. mean power consumption) per application. With this purpose, it communicates with the broker, subscribing on given topics and collecting data for a specified time. Finally, it returns both the mean value and the timestamps of the start and the end of the monitoring. In order to keep track of the state of the monitored metric, the software is based on a structure (i.e. the *struct collector,* defined in ./collector/*antarex_collector.h)* that contains:

- the topic,
- the mean value,
- the start and the end monitoring timestamps.

The structure is the used by the following collector API (defined in the ./collector/*antarex_collector.c* file) to carry out the monitoring:

- collector_init(collector, broker_ip, broker_port);
- *collector_clean(collector);*
- *collector_start(collector);*
- *collector_get(collector);*
- *collector_end(collector);*

The collector_init() initializes the collector and subscribes to the topic (e.g. *power_package*), creating a thread to handle the received messages, while the collector_clean() is used to clean up the collector and close the callback thread. The remaining API functions are the core of the metrics aggregator: collector_start() starts the monitoring of the metric, taking note of the starting time of the monitoring; collector_get() stores in the struct collector both the monitoring time till that instant and the mean value, and continues the monitoring; finally, collector_end() has the same functionality of the collector_get(), but stops the monitoring.

It is noteworthy that all the monitoring functions of the API handle the collector in a completely transparent way from the user point-of-view. The only required steps are:

1. Include the collector API in your project

   i.e. #include "./collector/*antarex_collector.h*"

2. initialize a struct collector and select the related topic for each metric that needs to be monitored

   E.g.

   struct collector_val pow_pkg = { 0 };

   pow_pkg.mqtt_topic = "topic_pow_pkg";

3. pass a *pointer to a struct collector to the monitoring functions of the API

   E.g.

   collector_init(&pow_pkg, ...);

   collector_start(&pow_pkg);

   Note that for each metric that has to be monitored, the collector_init() and _clean() functions have to be called accordingly.

4. Read the timestamps and the mean value from the struct collector, after either collector_get() or collector_end() is executed.

5 Usage

In order to quickly get started with the ANTAREX Collector API, we provided a usage-example application (i.e. *collector-example.c*) to monitor the CPU power packages. We also provided a Makefile to easily compile it. Below are the steps to **build** and **run** the collector-example application:

```
>$ cd ./collector/
>$ make
>$ ./collector-example
```

The application will start to stress the compute node. Press *Ctrl-C* to get the mean power package value till that moment and continue the monitoring (in this case the *collector_get()* function is used). Press again *Ctrl-C* to stop the stress test and get the final mean power package value (this will instead execute the *collector_end()* function).