MQTT Based Calibration Service

Claudio Francese

MQTT Based Calibration Service

T.R. 16/2022 March 2022

I.N.RI.M. TECHNICAL REPORT

Abstract

This report describes the study and the implementation of a prototype of a modular, distributed flexible software for Calibration Laboratories.

The aim of the software is the separation of the tasks involved in laboratory activities, i.e. measurement, data management, certification, ISO17025 compliance, etc.

The modular architecture of the software and the use of the MQTT protocol for process communication are suitable for the geographical distribution of the tasks over many laboratories, possibly in different locations.

Contents

Simplified laboratory model

The simplified model of the ordinary tasks of a Calibration Laboratory can be divided into two main areas, the *Calibration Activity* and the *Laboratory Maintainance*.

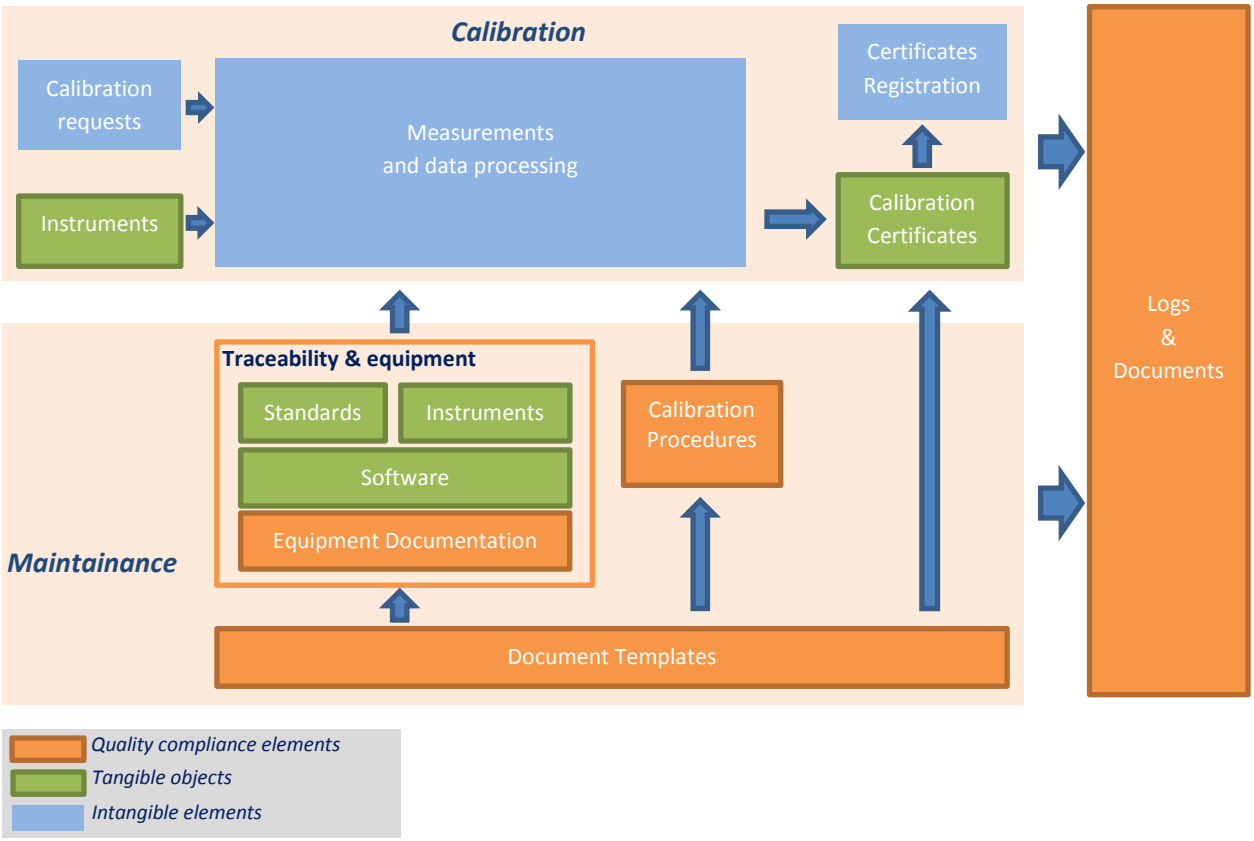


FIGURE 1 - SIMPLIFIED MODEL OF THE LABORATORY ORGANIZATION

In ordinary calibration activity, upon receiving a calibration request and the instrumentation to be calibrated, a measurement and data processing tasks start according to the criteria defined by the calibration procedures and laboratory documentation. The requested measurement traceability is guaranteed by calibration certificates of the Laboratory instrumentation and standards. At the end of data collecting and processing, calibration certificates are issued according to the Calibration Procedures and Certificates Templates ISO-17025 compliant which are accepted by the institutional Quality System. Every relevant event in the activity is registered in a logbook.

Software Architecture

As shown in Figure 1, the laboratory maintainance and calibration process are composed of many tasks with different kinds of access to data and resources, both physical (i.e. instrumentation) and abstract (i.e. procedures, templates, etc). For this reason the software is divided into abstraction layers, each using the specific resources and providing generic resources to the other layers. The used communication protocol for message exchange must be implemented in every node for the system to operate. Additional protocols can be implemented for specific requirements, e.g. access to physical measurement instrumentation trough USB, IEEE488, RS232, etc.

In every module of the system some common base functionalities are implemented thus allowing the integration with existing modules. The depicted behavior is based on an Object Oriented Programming

model which allows to implement the common functions in a superclass and the specialized functions, i.g. measurements, in derived classes.

The below picture shows how the main modules of the software are organized

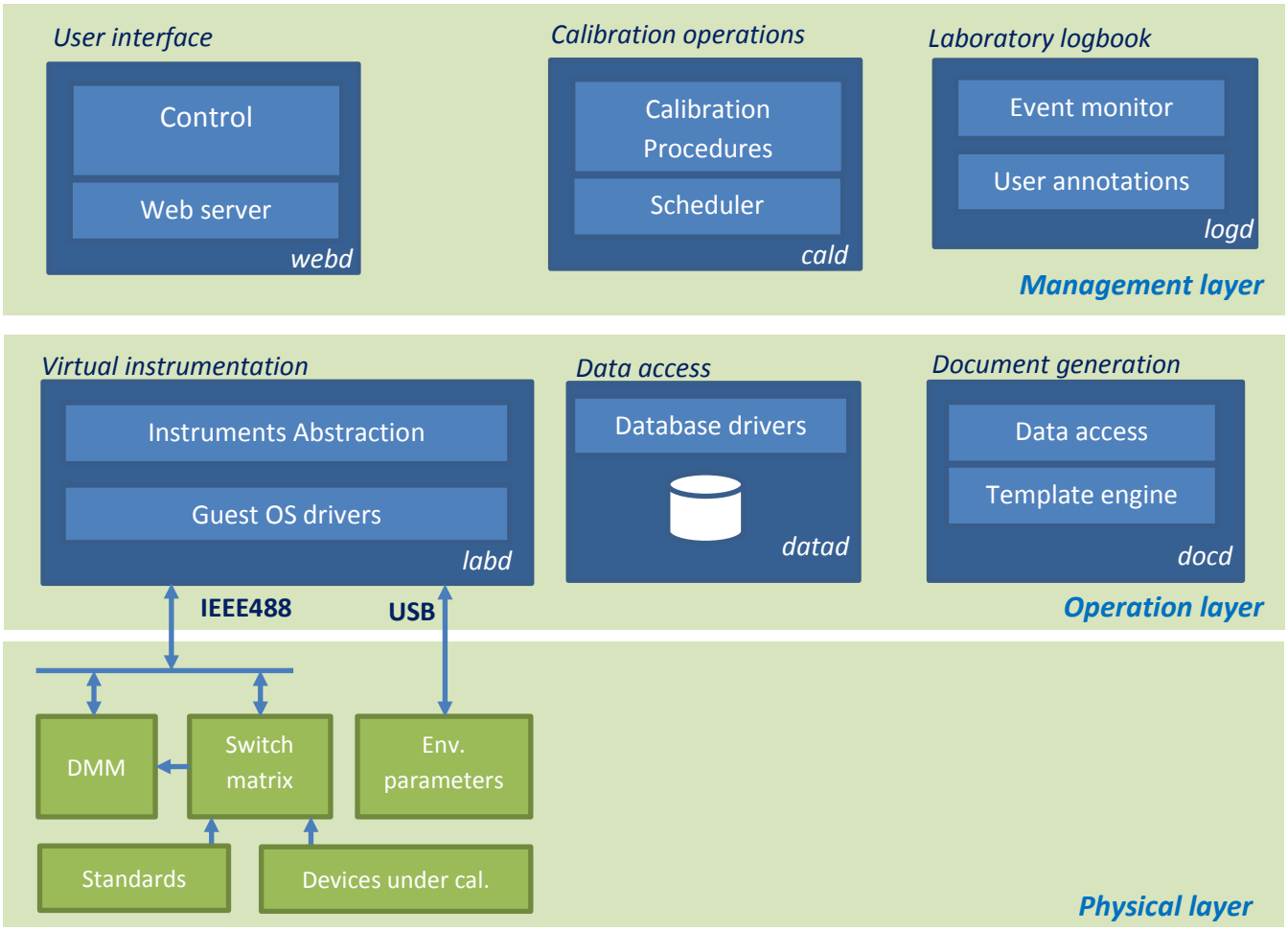


FIGURE 2 - SOFTWARE ARCHITECTURE

The chosen programming language to develop the prototype is Python 3.9 which in object oriented and offers a wide range of libraries for instrumentation, parallel processing, data processing and networking. Although the general concepts described in this report do not depend on the programming language and support libraries exist for many languages, the shown examples and code snippets are written in Python for simplicity.

Base node architecture and resources

The base behavior of a node connected to the system is defined as follows.

Base node communication functions

The base user-callable functions defined by the superclass allow the nodes to communicate and are used to control the execution of the internale tasks. The base functions are

Name	Description
Send	Used to send a message to one or more nodes of the system
RequestAndWait	Send a request to one or more nodes and wait for a reply until timeout

Reply	Used to reply to a received request
Dispatch	Used to forward a request to other nodes. This function is used by nodes which act as data collectors from other subnodes.

TABLE 1 - MESSAGING FUNCTIONS

Base node operation functions

Other user-callable low level functions related to the node operation are

Name	Description
ping	Used to check whether a node is operating
map	Used to request a node the list of *capabilities*
stop	Used to terminate the excution of a node

Base node capabilities

The *capabilities* list defines a list of messages which can be accepted by the node. This allows a new nodes to be connected to a running system and "teach" the surrounding nodes how to request the execution of their functions. Every capabiliy entry has a textual identifier, a textual description, an optional set of parameters. Each listed capability is implemented in the specialized class of the module and the modules using that capability as well – unless the function has been already implemented in the superclass for common tasks.

The base *capabilities* defined in the superclass are a subset of the implemented functions as show in the table below.

Identifier	Returned textual description	Description of the capability
stop	Stop the service	The node is able to terminate
map	List capabilities	The node is able to return the list of *capabilities*

As depicted in Figure 4, when specializing the superclass for a specific task, the implemented functionalities and capabilities list are extended as well.

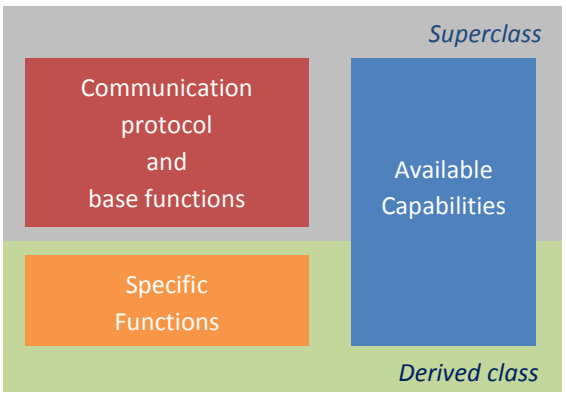


FIGURE 3 - ARCHITECTURE OF A NODE

Nodes integration in the system

The nodes exchange messages through a network connection, this allows the system to be easily distributed on remote machines.

Among the available network messaging protocols, MQTT has been chosen.

MQTT message queue exchange protocol

MQTT is a message exchange protocol between networked devices commonly used in IoT devices as it has small implementation footprint and the capability of handling message queues according to a subscribe/publish model. The protocol defines the *topic* concept which every client can subscribe to. Upon subscription, the client receives messages concerning the specified topics in the message payload section. Sending data to a *topic* does not require any subscription instead. In spite of being a high level protocol, thus the overall behavior of MQTT resembles a pool of devices interconnected by data busses, each bus for each *topic*, where all the connected devices receive messages from the bus. Unlike usual bus implementations, while data receiving requires subscription, the devices are allowed to send data on the other busses.

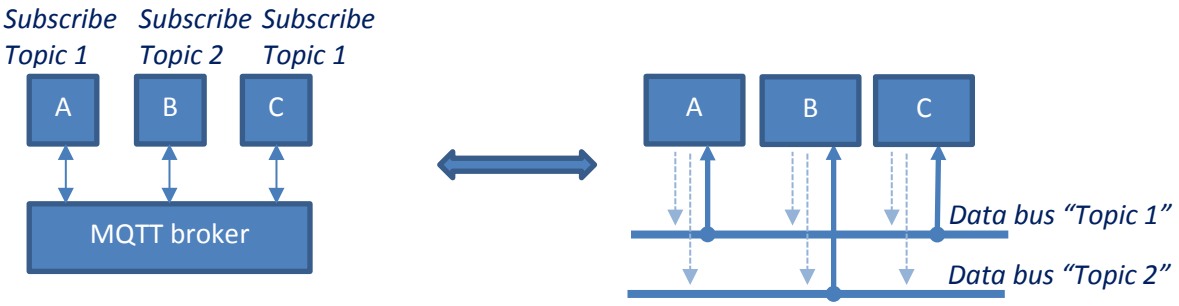


FIGURE 4 - MQTT PROTOCOL - DATA BUS ANALOGY

The core of the message queue handling mechanism is a server running the so-called *MQTT broker* which takes care of receiving the incoming messages and dispatching them to the subscribed clients.

Up to 3 Quality of Service (QoS) level are defined in the protocol depending on the desired reliability of the message exchange and among the features defined in MQTT which can be found in [1], the protocol also implements *Retained Messages* which are kept in memory by the *broker* until new messages are received. This allows to update the new subscribing clients with the last known state, for example the last measurement value sent by a device.

A common implementation of a multiplatform *MQTT broker* is *Eclipse Mosquitto*, which has been chosen in this project and deployed in a *Linux* environment. For security reasons, the Mosquitto instance has been protected with username and password. The connection is certificate encrypted and the connection is limited only to local connections (`localhost, ip addresses 127.0.0.0/8`) during the development.

The used MQTT implementation for Python is *paho* [2], which is part of the Eclipse IoT Project.

The Paho module allows an easy access to the message queue as shown in the following example[1].

```
# import the required modules
import paho.mqtt.client as mqtt
import time

# create the MQTT client instance
client = mqtt.Client()

# configure the secure connection to the broker (optional, suggested)
client.tls_set(ca_certs = '/some/certificate', tls_version=2)
client.tls_insecure_set(True)  # for self-signed certificates
client.username_pw_set('username', 'password')

# connect to the broker
client.connect('host name', 'tcp port', 60)

# callback function  when  a connection has been established
def on_connect(client, userdata, flags, rc):
    print("Connected with result code "+str(rc))
    client.subscribe("#")

# callback function when a message is received
def on_message(client, userdata, msg):
     print( '\r(' + msg.topic + '): ' + str(msg.payload))

# associate the callback functions to connection and message events and start the message queue handling loop
client.on_connect = on_connect
client.on_message = on_message
client.loop_start()

# do nothing until CTRL+C is pressed, handling is event driven by the callback functions
while True:
    try:
        time.sleep(0.2)
    except KeyboardInterrupt:
        break

# stop the message queue handling loop and terminate
client.loop_stop()
```

As the target of the communication is data exchange among the nodes, MQTT itself is only part of the messaging protocol and an abstraction layer must be defined for the communication to happen without relying on a specific network protocol or a specific channel implementation. Changing the lower layers of the messaging mechanism thus means redefining the superclass methods “send”, “reply”, “dispatch”, etc introduced in Table 1.

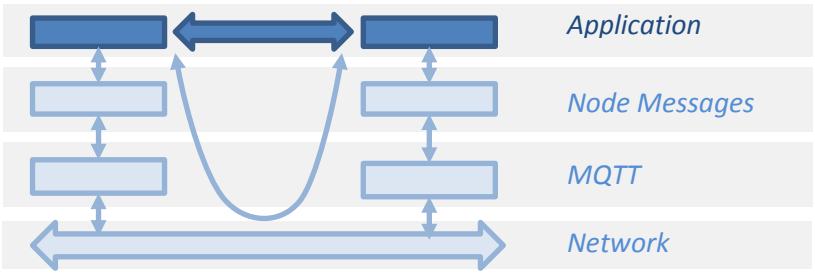


FIGURE 5 - COMMUNICATION PROTOCOL ABSTRACTION

[1] As during development a self-signed certificate was used, a call to `tls_insecure_set(True)` was required for the connection to be established. In production environments, a certificate properly issued by a trusted certification authority is needed for security reasons and the function call to `tls_insecure_set` in not required anymore.

Data format

Data format of the MQTT *payload* is not strictly defined in the protocol neither there are constraints for the definition of the protocol used for the inter-node communication, thus a flexible way to describe data is a textual encoding of a Python *object*. JSON encoding has been chosen for better compatibility with further implementations of the system and integration with other programming languages, i.e. ecmascript for web based nodes.

In all exchanged data messages, binary objects are base64 encoded with the limitation of 256 MB of the total packet size given by the MQTT implementation *of Eclipse Mosquitto*.

All the data in the exchanged messages on a given *topic* are then JSON encoded objects following these conventions.

Object structure

Mandatory fields	Optional fields

The mandatory fields are common to every message and are expected by the receiving nodes.

Field	Comments
`timestamp`	Timestamp of message expressed as seconds from the *Unix Epoch*
`UTC`	Human-readable formatted date and time (UTC timezone)
`from`	Identifier of the sending service/node – used to reply to requests
`to`	Identifier of the receiver for point to point messages. Broadcast messages are identified by **"*"**

The optional fields instead vary according to the nature of the message.

Message types

Two types of messages can be sent by the nodes depending on the expected behavior of the receiving nodes.

Simple message

Simple messages are sent to one or more nodes of the system when **no reply is expected**. An example is the system command used by a node to inform the other nodes of its activation. Other notifications include data coming from a background measurement task or status messages.

Example. A message sent to a node to request it to stop is as follows

Message	
Topic	**Object**
system	to = some_node from = some_controller cmd = "stop"

Python code

```
client.publish(
  'system',
  '{
    "to":"some_node",
    "from":"some_controller",
    "cmd":"stop"
    }' )
```

Upon receiving a simple message, the node **can** send a message to other nodes for logging or debug purposes although a reply is not expected by the originating node.

Example of the notification sent to all the nodes on the "announce" topic upon receiving a "stop" message.

Message description	JSON encoded object

Broadcast information to the system	<pre>{
	"message": "bye",
	"timestamp": 1645788432.493238,
	"UTC": "2022-02-25 11:27:12.493238",
	"from": "some_node",
	"to": "*"
	}</pre>

Request message

Request messages are sent on the "request" *topic* to one or more nodes when **a reply is expected by the originating node**.

An example is the map of the nodes currently connected to the system. Upon receiving the map request every node replies to the sender.

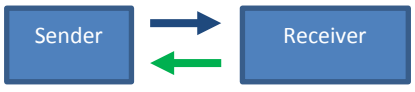
Field	Comments
request	Text of the request sent to the identified service
requestid	ID of the request. The ID is the concatenation of the fields "from" and "timestamp" separated by "-"

The reply to the request is then sent back to the originating node on the "reply" *topic*

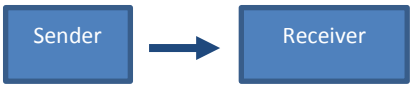
Field	Comments
requestid	ID of the request the reply refers to
reply	Text of the reply to the request

The mandatory fields "to", "from", "timestamp", "UTC" are automatically inserted by the communication methods.

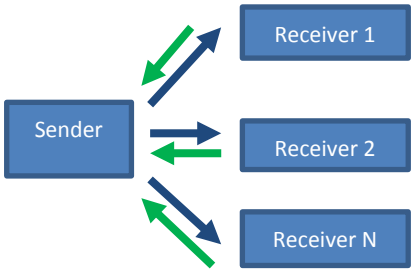
By combining the destination type of messages (single node or broadcast) and the message types, four communication scenarios are possible.



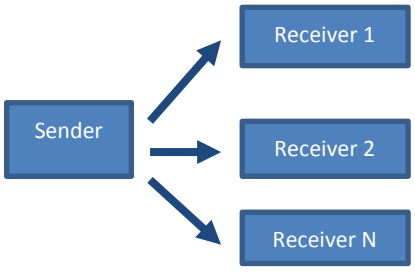
Point to point request and reply



Point to point simple message



Broadcast request and reply



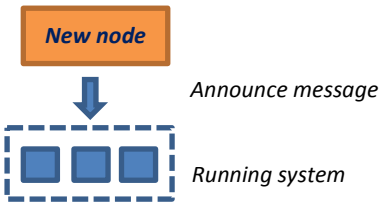
Broadcast simple message

System handshaking

Besides the communication protocol and the base behavior of the nodes, some conventions have been defined for the system to operate.

Joining to the networked services

When a node is added to the system, it sends a greeting message on the announce topic. The payload of the message contains the name of the joining node. Announcing to the system allows the system to start some actions, i.e. asking the node to provide further details or sending initialization commands which depend on the current state of the system or logging the event.



Implementation of the system

All nodes connected to the system share a common architecture and are implemented as derived classes from the parent class **`BaseService`**.

Base service software of the interconnected nodes

The methods defined by the superclass **`BaseService`** handle the message exchange in both directions as described in the previous sections thus reducing the specialization of the derived classes to the core functionalities which are needed.

For the complete system operation the following behavior has been defined upon creation of an instance of the **`BaseService`** class

- the base YAML configuration file is loaded – e.g. for MQTT connection parameters
- a connection to the MQTT broker is executed
- the node subscribes to the `"system"`, `"request"` and `"reply"` *topics* which are used to handle incoming messages

The base initialization code in the **`BaseService`** class is

```
class BaseServiceClass:
    config = {}
    subscriptions = []
    handlers = {}
    client = None
    running = True

    def __init__(self, options = {}):
        '''
        Class constructor
        :param options: service configuration
        '''
        self.config = options

        # Service name, used for addressing in messages
        self.servicename = self.config.get('servicename', None)

        # Raise an exception is no service name was specified
        if None == self.servicename:
            raise Exception("Missing 'servicename' option in BaseServiceClass constructor call")

        # Merge the read options from the configuration file to the options passed to the constructor
        self.configpath = options.get('configurationpath', configurationpath)
        for key, value in yaml.safe_load(open(self.configpath + "baseconfiguration.yaml", "r")).items():
            self.config[key] = value
        . . .
```

After configuring the service, the connection to the broker is established, the connect and message callback function are registered, and the message queue loop is started.

```
        # create an instance of the mqtt client from the paho module
        self.client = mqtt.Client()

        # configure the secure connection to the broker
        if 'ca-cert' in self.config.get('mqtt', {}):
            self.client.tls_set(
                ca_certs=self.config['mqtt']['ca-cert'], tls_version=2)

        if 'secure' not in self.config.get('mqtt', {}):
            self.client.tls_insecure_set(True)  # needed for self-signed certificates

        if 'username' in self.config.get('mqtt', {}) and 'password' in self.config.get('mqtt', {}):
            self.client.username_pw_set(
                self.config['mqtt']['username'], self.config['mqtt']['password'])

        # connect to the MQTT broker
        try:
            self.client.connect(
                self.config['mqtt']['hostname'], self.config['mqtt']['port'], 60)
        except ConnectionRefusedError:
            pass

        # Callback functions

        # callback functions for incoming messages and connection/reconnection to the mqtt broker
        self.client.on_message = self.on_message
        self.client.on_connect = self.on_connect

        # subscribe to the 'system' topic
        self.subscribe(topic_SYSTEM)
        self.subscribe(topic_REPLY)

        # start the message queue handling loop
        self.client.loop_start()
```

An instance of the superclass itself can be easily created for test purposes as well.

```
import time

# start the service
service = BaseServiceClass ({'servicename': 'testnode' })

# idle until service is running. Stop the service on CTRL+C
while service.running:
    try:
        time.sleep(0.5)
    except KeyboardInterrupt:
        service.stop()

print('done.')
```

Thus, the node identifies itself as “testnode” and it provides the base capabilities defined in the superclass.

Implementation of derived nodes

Given the description of the base superclass and the system overview, the following sections show a possible implementation of the required modules of the system.

The needed elements for the system to operate are

- **webd -** user interface node acting as a bridge between the user and the system
- **ctrld -** control node which acts as a monitor of the system
- **cald -** calibration node sending measurement requests and processing measurement data
- **labd -** one or more nodes performing the measurements over possibly many laboratories
- **datad -** storage node used to keep track of measurements, certification data and events
- **tpld -** template engine used to produce quality compliant documents and calibration certificates
- **logd** – logbook management node for event tracking

webd – user interface node

The node acts as a gateway between the network protocol used by the system and a higher level protocol for the system management. The chosen protocol is https and the user interface is implemented in html, javascript /jquery thus giving access from virtually any web browser with a responsive behaviour. The chosen solution has some advantages over other access models (i.e. client server model through a dedicated user interface application running on the users computer)

- Filtering of the functionalities is embedded in the node, this allow not to expose all the functions available thus giving a higher security level
- No software is needed on the client computer
- Network access through a *well-known* TPC/IP port enhances compliance to local network access policies and overall network security

ctrld – system controller

The node presents itself as a single entry point controlling the system. Although all the connected nodes can send and receive messages, having a single node to exchange messages with offers the following advantages

- Abstraction of the implementation of the system, thus offering higher level functionalities like overall status report, starting / stopping calibration tasks (which instead would involve communication with many nodes), etc. This eases interfacing with other software and programming language compatibility exposing an API to the developer
- Enhanced security by isolation of functions offered by the single nodes, by exposing only a subset of the system and single nodes capabilities

datad – data storage

The node gives access to data storage, implemented in a Postgresql database, in order to persist data. The database stored information are

- Measurements results
- Measurement configurations
- Calibration Certificates data
- Logbook entries
- Historical information about measurands

The storage node subscribes to topics where data to be recorded are sent, for example the topic `"meas"` for measurement results. This approach enhances scalability as data from new measurement nodes are automatically recorded provided the new node sends results on the `"meas"` topic.

tpld – template based document generator

The node is used to generate documents (Calibration Certificates, periodic reports, logbook, etc) by automatically filling-in the available document templates. The advantages of this approach with respect to manually typing the information are

- Compliance to the official documentation format
- Reduced typing errors
- Reduced editing time and batch capabilities

logd – logbook access

The node gives access to the laboratory event logbook requested for Quality compliance. Besides the automatically generated events (i.e. a calibration job has started) other events can be manually added by when needed. This approach has the following advantages over a paper logbook.

- All the logbook entries can be easily searched by specific criteria
- The log may contain the configuration of the instrumentation used during a calibration, thus reducing transcription errors
- References to log entries can be used track a calibration process and ease customer management (i.e. event in receiving the measurand, parameters used the calibration, certification reference and delivery details)
- Many ordinary tasks can be automatically logged reducing oversights risks

cald - calibration node

The calibration node coordinates all the tasks which are performed by the system when a calibration is requested. Its capabilities are

ID	Description
status	Report the status of the system (including the status of single nodes)
checkin	Register an incoming DC voltage standard in the laboratory and connect it to a given measurement terminal including the requested calibration specification
checkout	Free a registered DC voltage standard for delivery
jobs	Set the measurement slots

Registering a DC Voltage Standard and calibration requirements, instructs the node about the needed measurement steps to perform the task. When the calibration slot is reached, according to the programmed *jobs* plan, the node sends the commands to the relevant nodes of the system.

labd - Virtual instrumentation node

Using the described approach allows to define a service which abstract the measurements which can be performed inside the laboratory. Even when an instrument, i.e. a DMM, is substituted with a different brand and model, the affected node can keep supplying measurement data compatible with the existing

system and only a part of the software of that node must be modified in order to handle the possibly different communication with the instrument.

As the paradigm of the virtual instrument must not depend on the different physical instruments, connections types to the node (i.g LAN, USB, IEEE488, serial, etc) and physical instrument behavior, the needed abstraction layer must take into account the communication with the instruments as well.

Abstraction layers for the instrumentation virtualization

Abstraction of tasks has been chosen to better integrate third party instruments into the system and to increase the overall reliability. Separation of the functions also allows to distribute the operations among different hardware nodes, for example taking measures from different laboratories or distributed sensors.

In this scenario, every node contributing to a measurement task or using measurement data subscribes to a dedicated *topic*, for example `"meas"` and it performs only the pertinent operations sending its measurement result to the `"meas"` *topic*.

In the example below, two measurement systems are present in two separate laboratories, bot subscribed to the `"meas"` topic. A system controller connected to the network triggers thermal or electrical measurements by sending *simple messages* on the `"meas"` topic. In one of the two laboratories a data logger is connected to the network as well and it subscribed to the `"meas"` topic. When one oth the two measurement nodes receives a trigger message from the controller it starts a measurement and when data are available, it sends the result on the `"meas"` topic. Finally, measurement sent on the network are then stored by the data logger which is waiting for messages on the `"meas"` topic.

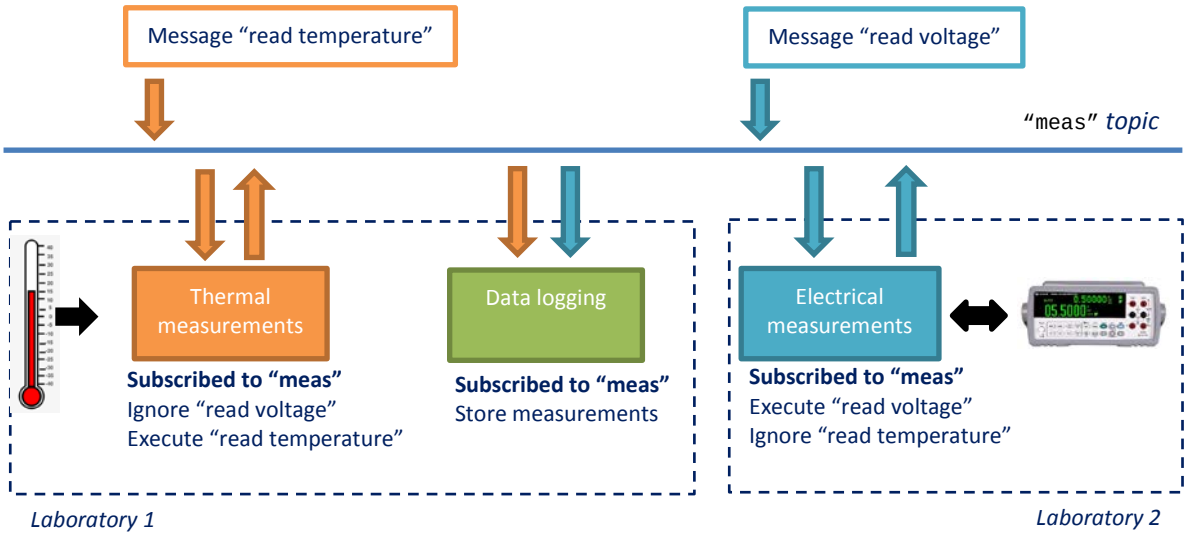


FIGURE 6 - EXAMPLE OF DISTRIBUTED MEASUREMENTS

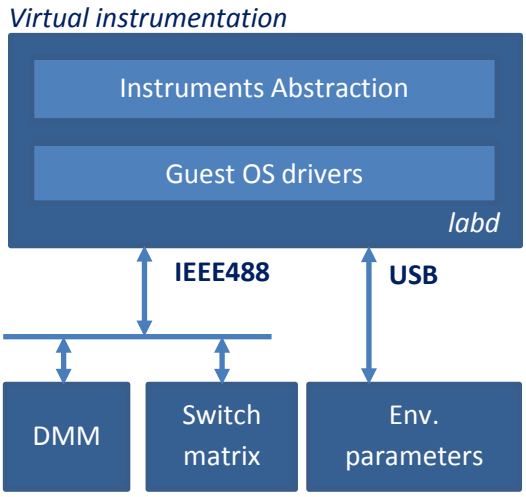


FIGURE 7 - VIRTUAL INSTRUMENTATION - LABD

In the end the Virtual Instrumentation node abstracts the "instrument able to perform all the needed measurements". This means than the module can be run on different computers, possibly in different locations, as already shown in the example in Figure 6.

For the INRIM DC Voltage Standard laboratory, the following quantities must be measured

- The DC voltage of a given voltage standard with different electrical configurations
- The environmental parameters in the laboratory during the measurements

The following picture shows the physical instruments are connected.

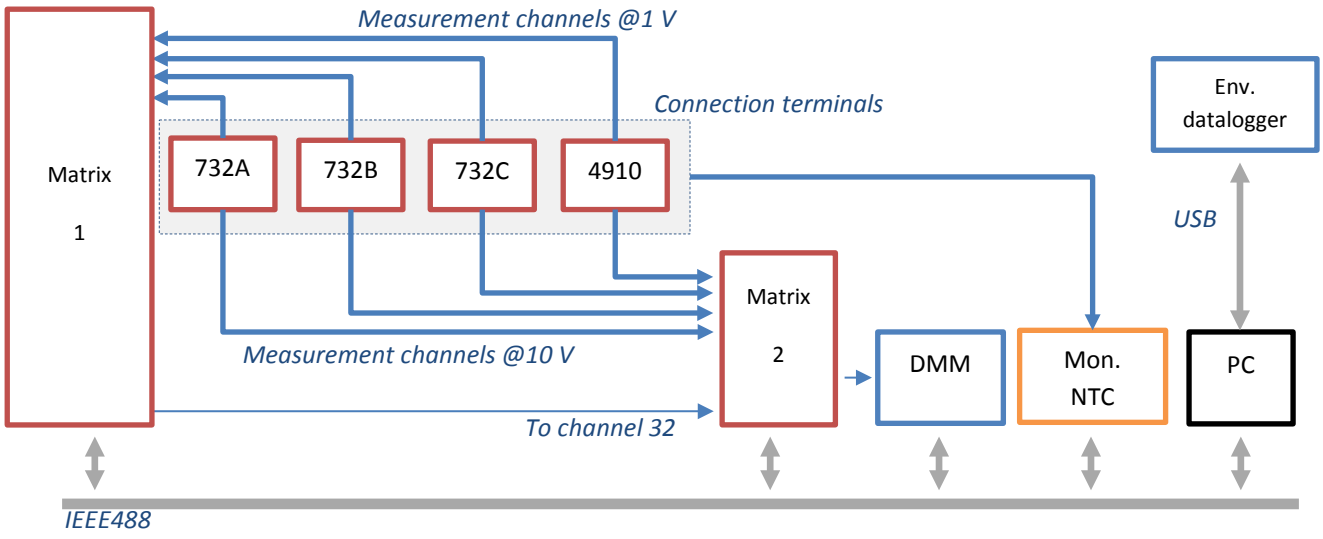
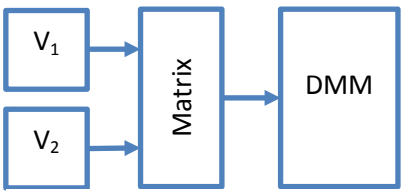


FIGURE 8 - DC VOLTAGE LABORATORY INSTRUMENTATION

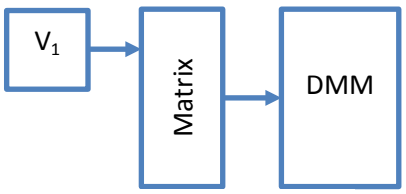
Voltage measurements – Virtual DMM

As Figure 8 shows, the voltage measurements are performed by a DMM which is connected to two low thermal e.m.f. matrixes which are used to select any of the voltage outputs of any of the connected DC voltage standard. The implementation of the Virtual DMM allows to define a multiple inputs DMM

composed of the DMM and the two matrixes. As the measurement can be performed with different integration times and resolution, those parameters are available in the Virtual DMM. Another feature of the instrument is the measurement technique relevant for calibration purposes and the ability to set the matrix in order to perform reverse polarity commonly used in DC measurements for the reduction of the effect of parasitic thermal e.m.f.



Differential measurement V_1-V_2



Absolute measurement V_1

The above requirements take to the definition of the Virtuall DMM which exposes the following capabilities, available by sending messaged on the "`meas`" topic to the node.

ID	Description
measure	Perform a configured measurement
repetitions	Set the number of repeated measurements
NPLC	Set the integration time
Ndigits	Set the number of digits
mode	Set the measurement mode (absolute, differential)
pol	Set the measurement polarity (+ / -)
src	Set the voltage standard ID (two standards for differential measurements)

Available measurements, triggered by the "measure" message, are then sent to the "`meas`" topic as a measurement result with the following format (mandatory fields are omitted for clarity).

Field	Description
`DC-measurement`	Measured value
`OriginUID`	Unique ID of the measurement node (laboratory ID, computer name, node name)
`NPLC`	Used integration time
`mode`	Used mode (absolute, differential)
`pol`	Measurement polarity
`src`	ID of the measured standard (two for differential measurements)

Bibliography

[1] MQTT.org, "MQTT: The Standard for IoT Messaging," 2022. [Online]. Available: https://mqtt.org/.

[2] "paho MQTT," 2022. [Online]. Available: https://www.eclipse.org/paho/.