The Vire-CMS to MOS interface
version 0.4

E.Chabanne, J.Hommet, T. Leflour, Y. Lemière, S.Lieunard, F.Mauger, J.-L. Panazol, J.Poincheval

May 19, 2016

Abstract

This document aims to define and describe the requirements of the Vire/CMS interface, i.e. the software bridge between the Vire based online software (Vire server and clients) and the OPCUA based MOS servers which are responsible of the low level control and monitoring operations on hardware devices.

Contents

List of Figures

List of Tables

1 Access to the CMS server by the Vire server

1.1 Introduction

The CMS server is the main service that allows the Vire server (and possibly its clients) to access the hardware resources managed by MOS servers in the OPCUA space (Fig. 1).

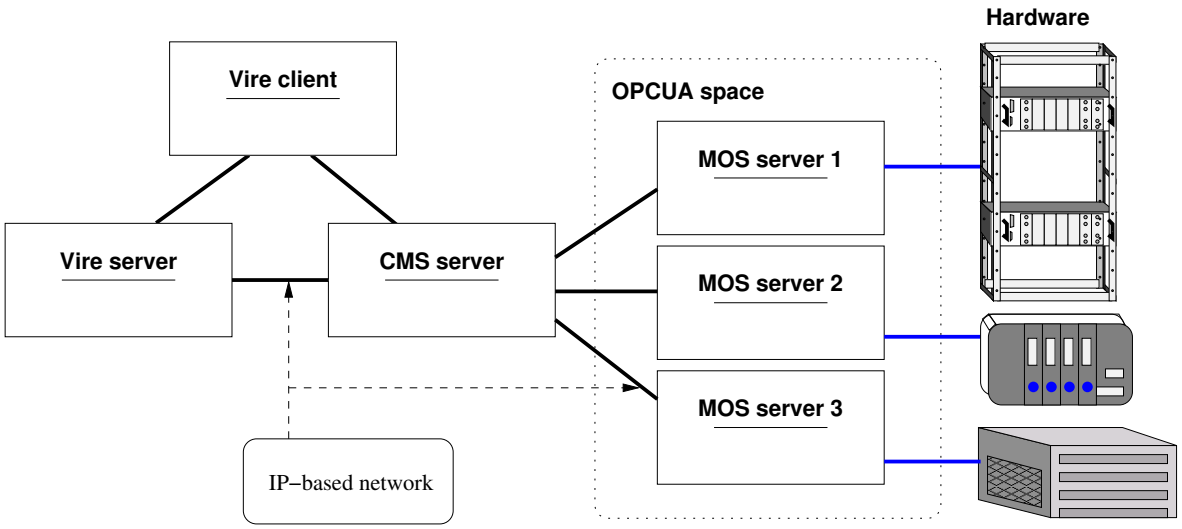

Figure 1: Overview of the Vire server, a Vire client, the CMS server and its connections to the hardware through dedicated MOS servers.

From the Vire server point of view, the CMS server hides the details of the real time and low level communication systems implemented in MOS servers and their embedded device drivers. Its role is to provide, in terms of Vire *resources*, a conventional access to the datapoints (DP) and dedicated actions (methods) associated to MOS devices.

1.2 Common configuration file and parameters

An unique configuration file is used to store the common parameters shared by the Vire server and the CMS server. This file must be available from a central configuration directory[1] and passed to the CMS server at startup. This file, as well as other configuration files, will be distributed and shared through a VCS [2]. It is ensured that the configuration files loaded at startup by both the Vire and CMS server will be synchronized through the VCS, even if they run on different nodes. Appendix A proposes a technique to identify the paths of all configuration files.

Example:

- at CMS server launching we should typically run:

shell
```
shell$ cmsserver \
  --setup-config=${SNEMO_ONLINE_CONFIG_BASE_DIR}/snemo/1.0.2/setup.conf \
  ...
```

- at Vire server launching we should typically run:

shell
```
shell$ vireserver \
  --setup-config=${SNEMO_ONLINE_CONFIG_BASE_DIR}/snemo/1.0.2/setup.conf \
  ...
```

In the example above, the same file `snemo/1.0.2/setup.conf` is distributed through a VCS in a conventional base directory to both Vire and CMS servers. The file should store a minimal set of mandatory parameters:

- The `setup_id` character string is the unique identifier label of the experimental setup.
- The `setup_version` character string encodes the version number of the experimental setup. It should typically use a conventional sequence-based identifier format (*major.minor.revision* scheme).
- The `setup_description` character string contains the description of the experimental setup.
- The `amqp_svr_host` character string encodes the IP address or the URL of the main RabbitMQ server used for communication between remote components involved in the system.
- The `amqp_svr_port` integer is the port number used by the main RabbitMQ server.
- The `amqp_svr_virtualhost` character string stores the identifier of the RabbitMQ virtual host used to confine the communication within the system.
- The `amqp_cms_service_channel` character string is the identifier of the *service* channel used by the Vire Server to request a connection with the CMS server.
- The `amqp_cms_XXX_channel` character string is the identifier of additional channel used by the Vire server, clients and the CMS server (*XXX* can be `control`, `monitoring`, `event` or `pubsub` (see below).

[1] In the example, we use an environment variable to define this base directory.

[2] Version control system. Note that the SuperNEMO experiment uses a Subversion server hosted at LPC `https://nemo.lpc-caen.in2p3.fr/svn`

- The `vire_device_launching_path` character string represents the path of the configuration file which stores the informations about the devices managed by MOS servers and the mapping rules to address them in the Vire naming scheme. The format of this file is described later in this document.

An example of configuration file is shown on Fig. 2. NOTE (FM): the exact format of this file will be detailed later.

Configuration

```
setup_id             = "snemo"
setup_version        = "1.0.2"
setup_description    = "The SuperNEMO Demonstrator Experiment"
amqp_svr_host        = "192.10.0.23"
amqp_svr_port        = 4253
amqp_svr_virtualhost = "asWLjr8hoj"
amqp_cms_service_channel    = "snemo.ampq.cms.service.cZqblZo0mY"
amqp_cms_control_channel    = "snemo.ampq.cms.control.IKA8b7SfK2"
amqp_cms_monitoring_channel = "snemo.ampq.cms.monitoring.n25VDOX7wG"
amqp_cms_event_channel      = "snemo.ampq.cms.event.5Hja55450z"
amqp_cms_pubsub_channel     = "snemo.ampq.cms.pubsub.WAqq7ERzsl"
mos_device_launching_path   = "${SNEMO_ONLINE_CONFIG_BASE_DIR}/snemo/1.0.2/cms/devices_launch.conf"
```

Figure 2: Example of main configuration file shared by the Vire and CMS servers.

1.3 Communication between the Vire and CMS servers

The CMS server and the Vire server are connected together through bidirectionnal communication channels using the AMQP protocol (Fig. 3).
The proposed communication scheme uses five of such channels:

1. the *service* channel is dedicated to system messages between the *system/core* parts of both Vire and CMS servers. This would include connection and disconnection requests, handcheck messages, some specific system log and alarm messages... but no message concerning datapoint (DP) read/write operations or invocation of devices methods. Dedicated *bindings* could be initiated as soon as the connection is established between both servers.

2. the *control* channel is dedicated to the execution of control operations on devices and datapoints through explicit *control* resources (writing setpoints for datapoints, configuration methods...).

3. the *monitoring* channel is dedicated to the execution of monitoring operations on devices and datapoints through explicit *monitoring* resources (reading datapoint values...).

4. the *event* channel is dedicated to the transmission of asynchronous *event* messages (alarm, log, error, signals...).

5. the *pubsub* channel is dedicated to the transmission of asynchronous *publish/subscribe* messages (datapoint value updates). It is expected that specific *bindings* should be associated to the datapoint related resources eligible to the publish/subscribe mechanism.

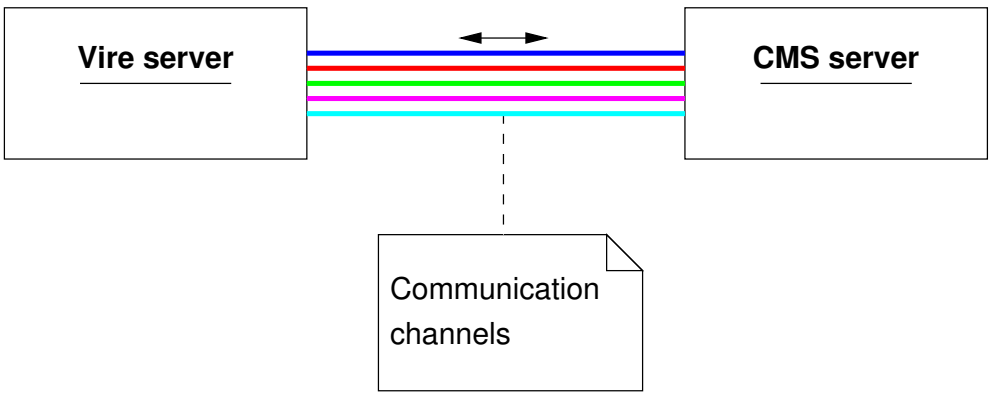

Figure 3: The communication channels between the CMS server and the Vire server.

We will use an AMQP RabbitMQ server as the communication framework between the Vire server, Vire clients and the CMS server (Fig. 4).

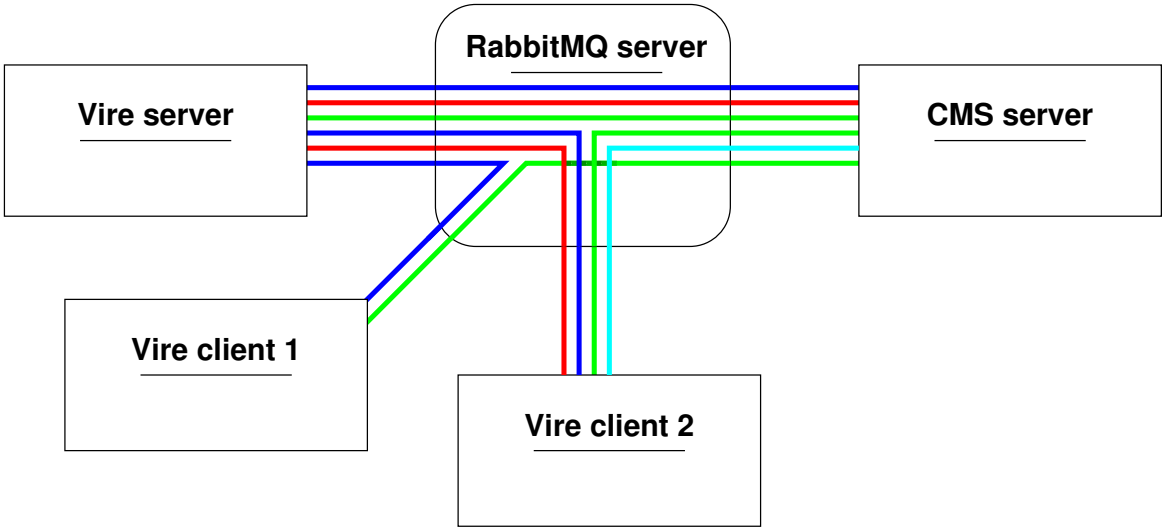

Figure 4: A RabbitMQ based communication framework shared by the CMS server, the Vire server and some Vire clients. Each component uses a set of dedicated channels (service, monitoring, control, event, pubsub).

1.4 Connection parameters

1.4.1 Setup identification and system communication

The CMS server must store the following minimal set of parameters that should allow to identify the experimental setup and connect to the Vire/CMS network with a well defined configuration:

1. `setup_config_path`:

   This is the path of the setup main configuration file. This parameter may be passed at startup of the CMS server or through an environment variable `SNEMO_SETUP_CONFIG_PATH`.
   Example: `${SNEMO_ONLINE_CONFIG_BASE_DIR}/snemo/1.0/setup.conf`

2. `setup_id`:

   The identifier of the setup is extracted from the main setup configuration file. It is a character string that identifies the Vire/CMS harware setup instantiated through OPCUA tools.
   Example: `snemo`

3. `setup_version`:

   The version of the setup is extracted from the setup main configuration file. It is a character string using a standard version numbering scheme.
   Example: `1.0`
   or an arbitrary tag:
   Example: `test_13`

4. `amqp_svr_host`:

A character string that stores the IP address or a plain URL of the RabbitMQ server.
Example: `"snemo.amqp.lsm.in2p3.fr"` or `192.10.0.23`

5. `amqp_svr_port`:

   An integer that stores the IP port of the RabbitMQ server.
   Example: `4253`

6. `amqp_svr_virtualhost`:

   A character string that stores the unique identifier of the RabbitMQ virtual host used to confine the Vire/CMS comunications.
   Example: `"snvhost"`

7. `amqp_cms_service_channel` :

   The conventional identifier of the *service* channel used by the Vire and CMS servers to transmit administration messages is stored as a character string.
   Example: `"snemo.amqp.cms.service.cZqblZo0mY"`.

8. `amqp_cms_control_channel` :

   The conventional identifier of the *control* channel used by the Vire server to transmit *write/control* request messages to the CMS server is stored as a character string.
   Example: `"snemo.amqp.cms.control.IKA8b7SfK2"`.

9. `amqp_cms_monitoring_channel` :

   The conventional identifier of the *monitoring* channel used by the Vire server to transmit *read/monitoring* request messages to the CMS server is stored as a character string.
   Example: `"snemo.amqp.cms.monitoring.n25VDOX7wG"`.

10. `amqp_cms_event_channel` :

    The conventional identifier of the *event* channel used by the CMS server to transmit *event* messages from the Vire server is stored as a character string.
    Example: `"snemo.amqp.cms.event.5Hja5545Oz"`.

11. `amqp_cms_pubsub_channel` :

    The conventional identifier of the *pubsub* channel used by the CMS server to transmit *pubsub* messages to Vire subscribers is stored as a character string.
    Example: `"snemo.amqp.cms.pubsub.WAqq7ERzsl"`.

12. `mos_device_launching_path` :

    This file stores the informations that describe how OPCUA based device handlers are instantiated within MOS servers and how the OPCUA devices naming scheme is mapped (mounted) into the Vire devices/resources naming scheme.

An example of `${SNEMO_ONLINE_CONFIG_BASE_DIR}/snemo/1.0.2/cms/devices_launch.conf` file is shown on figure 5.

1.4.2 Parameters related to the connexion with the Vire server

In order to properly communicate with the Vire server, the CMS server must use the following working data as long as the connection with the Vire server is active:

1. The identifiers of some AMPQ channels dedicated to specific operations like: service operations, control operations, monitoring parameters, events broadcasting, datapoint publish/subscribe system.

   These are the five channels with respectively so-called types: *service*, *control*, *monitoring*, *event* and *pubsub*. These identifiers may be generated randomly possibly using some specific patterns. Table 1 shows a possible set of such channels, each associated to a given type. Only one channel of one type can exists at the same time within a given Vire to CMS connection. The main configuration file defines the identifiers of these channels.

Channel type	Sync/Async	Channel name
vire::amqp::service	sync	"snemo.amqp.viresvr.cZqblZo0mY"
vire::amqp::control	sync	"snemo.amqp.viresvr.IKA8b7SfK2"
vire::amqp::monitoring	sync	"snemo.amqp.viresvr.n25VDOX7wG"
vire::amqp::event	async	"snemo.amqp.viresvr.5Hja55450z"
vire::amqp::pubsub	async	"snemo.amqp.viresvr.WAqq7ERzsl"

Table 1: Conventional named channels with their specific types defined for communication between the Vire server, the CMS server and Vire clients.

.

```
# Vire/MOS device mapping:
# Author: SuperNEMO COllaboration
# Date:   2016-02-25

#
# mos_host                 mos_user                                                                 mos_namespace          vire_device_path (mount point)
#                     mos_port   mos_device_config                                                      mos_root_device

# Calorimeter high voltage power supplies:
lappc-f4981.in2p3.fr 48024 user /MOS/SNEMO/CALORIMETER/CALO_HVPS/CALO_HVPS_1.xml                    CMS HVPS               SuperNEMO://Demonstrator/CMS/HV/Calorimeter/PS_0/
lappc-f4981.in2p3.fr 48025 user /MOS/SNEMO/CALORIMETER/CALO_HVPS/CALO_HVPS_2.xml                    CMS HVPS               SuperNEMO://Demonstrator/CMS/HV/Calorimeter/PS_1/
lappc-f4981.in2p3.fr 48026 user /MOS/SNEMO/CALORIMETER/CALO_HVPS/CALO_HVPS_3.xml                    CMS HVPS               SuperNEMO://Demonstrator/CMS/HV/Calorimeter/PS_2/

# Tracker high voltage power supplies:
lappc-f4981.in2p3.fr 48027 user /MOS/SNEMO/TRACKER/TRACKER_HVPS/TRACKER_HVPS_1.xml                  CMS HVPS               SuperNEMO://Demonstrator/CMS/HV/Tracker/PS_0/
lappc-f4981.in2p3.fr 48028 user /MOS/SNEMO/TRACKER/TRACKER_HVPS/TRACKER_HVPS_2.xml                  CMS HVPS               SuperNEMO://Demonstrator/CMS/HV/Tracker/PS_1/
lappc-f4981.in2p3.fr 48029 user /MOS/SNEMO/TRACKER/TRACKER_HVPS/TRACKER_HVPS_3.xml                  CMS HVPS               SuperNEMO://Demonstrator/CMS/HV/Tracker/PS_2/

# Coil:
lappc-f4981.in2p3.fr 4841  user /MOS/SNEMO/COIL/COIL_PS_1.xml                                       CMS COIL_PS            SuperNEMO://Demonstrator/CMS/Coil/PS/

# Calorimeter frontend crates:
lappc-f4981.in2p3.fr 48071 user /MOS/SNEMO/CALORIMETER/CALO_CB_CRATE/CALO_ELECTRONICS_CRATE_1.xml   CMS Electronics_Crate  SuperNEMO://Demonstrator/CMS/Electronics/Calorimeter/Crate_0/
lappc-f4981.in2p3.fr 48072 user /MOS/SNEMO/CALORIMETER/CALO_CB_CRATE/CALO_ELECTRONICS_CRATE_2.xml   CMS Electronics_Crate  SuperNEMO://Demonstrator/CMS/Electronics/Calorimeter/Crate_1/
lappc-f4981.in2p3.fr 48073 user /MOS/SNEMO/CALORIMETER/CALO_CB_CRATE/CALO_ELECTRONICS_CRATE_3.xml   CMS Electronics_Crate  SuperNEMO://Demonstrator/CMS/Electronics/Calorimeter/Crate_2/

# Tracker frontend crates:
lappc-f4981.in2p3.fr 48074 user /MOS/SNEMO/TRACKER/TRACKER_CB_CRATE/TRACKER_ELECTRONICS_CRATE_1.xml CMS Electronics_Crate  SuperNEMO://Demonstrator/CMS/Electronics/Tracker/Crate_0/
lappc-f4981.in2p3.fr 48075 user /MOS/SNEMO/TRACKER/TRACKER_CB_CRATE/TRACKER_ELECTRONICS_CRATE_2.xml CMS Electronics_Crate  SuperNEMO://Demonstrator/CMS/Electronics/Tracker/Crate_1/
lappc-f4981.in2p3.fr 48076 user /MOS/SNEMO/TRACKER/TRACKER_CB_CRATE/TRACKER_ELECTRONICS_CRATE_3.xml CMS Electronics_Crate  SuperNEMO://Demonstrator/CMS/Electronics/Tracker/Crate_2/

# Calorimeter control boards:
lappc-f4981.in2p3.fr 48091 user /MOS/SNEMO/CALORIMETER/CALO_CB/CALO_CB_1.xml                        CMS ControlBoard       SuperNEMO://Demonstrator/CMS/Electronics/Calorimeter/Crate_0/CB/
lappc-f4981.in2p3.fr 48092 user /MOS/SNEMO/CALORIMETER/CALO_CB/CALO_CB_2.xml                        CMS ControlBoard       SuperNEMO://Demonstrator/CMS/Electronics/Calorimeter/Crate_1/CB/
lappc-f4981.in2p3.fr 48093 user /MOS/SNEMO/CALORIMETER/CALO_CB/CALO_CB_3.xml                        CMS ControlBoard       SuperNEMO://Demonstrator/CMS/Electronics/Calorimeter/Crate_2/CB/

# Tracker control boards:
lappc-f4981.in2p3.fr 48094 user /MOS/SNEMO/TRACKER/TRACKER_CB/TRACKER_CB_1.xml                      CMS ControlBoard       SuperNEMO://Demonstrator/CMS/Electronics/Tracker/Crate_0/CB/
lappc-f4981.in2p3.fr 48095 user /MOS/SNEMO/TRACKER/TRACKER_CB/TRACKER_CB_2.xml                      CMS ControlBoard       SuperNEMO://Demonstrator/CMS/Electronics/Tracker/Crate_1/CB/
lappc-f4981.in2p3.fr 48096 user /MOS/SNEMO/TRACKER/TRACKER_CB/TRACKER_CB_3.xml                      CMS ControlBoard       SuperNEMO://Demonstrator/CMS/Electronics/Tracker/Crate_2/CB/

# Trigger:
lappc-f4981.in2p3.fr 48097 user /MOS/SNEMO/TRIGGER/TRIGGER_1.xml                                    CMS TriggerBoard       SuperNEMO://Demonstrator/CMS/Electronics/TB/

# Gas factory:
lappc-f4981.in2p3.fr 48022 user /MOS/SNEMO/GAS_FACTORY/GAS_Factory_1.xml                            CMS GAS_Factory        SuperNEMO://Demonstrator/CMS/GasFactory/

# Light injection system:
lappc-f4981.in2p3.fr 48021 user /MOS/SNEMO/LIGHT_INJECTION/LIGHT_INJECTION_1.xml                    CMS CalibrationLight   SuperNEMO://Demonstrator/CMS/LightInjection/

# Source calibration system:
lappc-f4981.in2p3.fr 48061 user /MOS/SNEMO/SOURCE_CALIBRATION/SOURCE_CALIBRATION_1.xml              CMS Source_Calibration SuperNEMO://Demonstrator/CMS/SourceCalibration/

# DAQ:
lappc-f4981.in2p3.fr 48040 user /MOS/SNEMO/DAQ/DAQdevice_1.xml                                      CMS DAQ                SuperNEMO://Demonstrator/CMS/Acquisition/

# LSM environment:
lappc-f4981.in2p3.fr 48023 user /MOS/SNEMO/LSM_ENVIRONMENT/LSM_ENVIRONMENT_1.xml                    CMS LSM_Environment    SuperNEMO://LSM/CMS/Environment/
```

Figure 5: Example of `devices_launch.conf` file.

1.5 Interface with the Vire server

The CMS server must provide the following interface to the Vire server:

1. Establish the connection between the Vire server and the CMS server:
2. Terminate the current connection between the Vire server and the CMS server
3. Handle the case of a non expected disconnection between the Vire server and the CMS server.
   NOTE (FM): the policy in case of accidental disconnection will probably depends on some functionnalities available from the RabbitMQ server.
4. Recover a broken connection between the Vire server and the CMS server.
   NOTE (FM): it is not clear for now if this case is really different than the *first connection* above case; also some specific functionnalities available from the RabbitMQ server may be used.

The conventional *service* channel should be used for these operations.

1.5.1 Establish the connection between the Vire server and the CMS server

Pre-conditions:

- The CMS server must not have an existing connection with the Vire server.
- The Vire server must not have an existing connection with the target CMS server.

Description:

1. The Vire server sends a *connection request* with the following parameters:

   - `setup_id` : the identifier of the setup which consists in two records :
     - `name` : a character string which typically stores the name of the experiment.
       Example: `name="snemo"`
     - `version` : a character string which stores the version number of the setup.
       Example: `version="1.0.2"`

     The setup identifier is expected to match the identifier stored in the CMS server (fetched from the main configuration file).
   - The list of Vire paths of all resources that are requested by the Vire server and that are expected to be made accessible through the CMS server conforming to the `devices_launch.conf` file. It is implemented as an array of character strings.
     Example:

     ```
     "SuperNEMO://Demonstrator/CMS/Coil/PS/Control/Current/__dp_read__"
     "SuperNEMO://Demonstrator/CMS/Coil/PS/Control/Current/__dp_write__"
     "SuperNEMO://Demonstrator/CMS/Coil/PS/Control/Voltage/__dp_read__"
     ...
     "SuperNEMO://Demonstrator/CMS/Acquisition/stop"
     ```

JSON

```
{
  "setup_id"      : {
    "name" : "snemo",
    "version" : "1.0.2"
  },
  "requested_resources" : [
    "SuperNEMO://Demonstrator/CMS/Coil/PS/Control/Current/__dp_read__",
    "SuperNEMO://Demonstrator/CMS/Coil/PS/Control/Current/__dp_write__",
    "SuperNEMO://Demonstrator/CMS/Coil/PS/Control/Voltage/__dp_read__",
    "SuperNEMO://Demonstrator/CMS/Coil/PS/Control/Voltage/__dp_write__",
    "SuperNEMO://Demonstrator/CMS/Coil/PS/Monitoring/Current/__dp_read__",
    "SuperNEMO://Demonstrator/CMS/Coil/PS/Monitoring/Voltage/__dp_read__",
    ...
    "SuperNEMO://Demonstrator/CMS/Acquisition/start",
    "SuperNEMO://Demonstrator/CMS/Acquisition/stop"
  ]
}
```

Figure 6: Proposal for a JSON `vire::mosinterface::connection_request` object emitted by the Vire server to the CMS server.

A typical JSON formatted *connection request* message is shown on Fig. 6. The exact format of the message will be discussed in appendix C.

The CMS server will be able to explicitely check that all resources requested by the Vire server are properly defined within some of the embedded MOS servers. It is the responsibility of the Vire server to build this set of requested resources. It is possible to request less resources than the ones that are effectively available from the MOS servers (following the `devices_launch.conf` startup file). In that case, the Vire/CMS connection is established in failsoft mode, for example for debugging purpose or in order to let the possibility to another Vire connection to be set with the same CMS server but requesting another set of resources (without any concurrent access of course).

Example: Assume the CMS server nominally gives access to a set $\mathcal{R}$ of 1000 different resources originating from 10 different MOS servers. The Vire server requests a connection with a set $\mathcal{V}$ containing only 250 target resources. If $\mathcal{V} \subset \mathcal{R}$, there is no problem for the CMS server to establish the connection because all the requested resources are under its responsability. If at least one of the resources in set $\mathcal{V}$ does not belongs to set $\mathcal{R}$ ($\mathcal{V} \subsetneq \mathcal{R}$), then the CMS server must reject the connection request from the Vire server because it will not be able to honor the contract (this case is expected to be a bug in the configuration of Vire and/or CMS servers).

It is important to note that if a given requested resource is recognized (in term of path and hosting device) by the CMS server but missing because of some MOS device connection error, this is not considered as a failure. A dedicated bitset associated to each resource is used to store the *status* of this resource (see section 2.2.4).

2. If the setup identifier and version are valid, and the set of requested resources is checked, the CMS server must:

   (a) generate the Vire/MOS naming mapping table for resources from the "`devices_launch.conf`" file.

   (b) compute the current *status* associated to each requested resources.

Then the CMS server transmits a `vire::mosinterface::connection_request_success_response` object to the Vire server, using the service channel. The message contains the list of resources status. A typical JSON format of a *connection accept* message is shown on Fig. 7.

JSON

```
{
  "resources_snapshot"  : [
    {
      "path"   : "SuperNEMO://Demonstrator/CMS/Coil/PS/Control/Current/__dp_read__",
      "status" : "0000"
    },
    {
      "path"   : "SuperNEMO://Demonstrator/CMS/Coil/PS/Control/Voltage/__dp_read__",
      "status" : "0000"
    },
    ...
    {
      "path"   : "SuperNEMO://Demonstrator/CMS/Acquisition/stop",
      "status" : "0100"
    }
  ]
}
```

Figure 7: Proposal for a JSON *connection accept* message emitted by the CMS server to the Vire server. The meaning of the `status` associated to each resource will be explained later.

If the setup identifier (either the name or the version) or at least one requested resource is not recognized, the CMS server must send back to the Vire server a specific failure message to reject the request. A dedicated `vire::mosinterface::connection_request_failure_response` object is built and encoded in a JSON formatted message (Fig. 8). The failure response message is able to encode two types of errors:

- `invalid_context_error` (error code=1) : if the context is not suitable to establish a connection with the Vire server (maintenance, server already connected...).
- `invalid_setup_id_error` (error code=2) : if requested setup identifier does not match the setup addressed by the CMS server.
- `unknown_resource_error` (error code=3): if at least one requested resource path is not recognized by the CMS server.

Each type of error contains a `code` integer attribute, and a `message` string attribute. Specific additional attributes may be available depending of the type of the error.

Post-condition:

When the connection is accepted and established:

- The Vire server stores the informations related to the connection as long as it lasts: the set of resources and associated informations (status, ...)
- The Vire server cannot send a connection request to the same CMS server through the *service* channel associated to this CMS server.
- The CMS server has a map of resources that permits to identify each requested resource and associate it to an item (datapoint, method) published by a given MOS.

Question: Should we use a keep-alive system messages ?

JSON

```
{
  "error_type_id" : {
    "name"    : "unknown_resource_error",
    "version" : "1.0"
  }
  "error" : {
    "code" : "2",
    "message" : "Connection to CMS server was refused because of not recognized resource path(s).",
    "unknown_resource_paths" : [
      "SuperNEMO://Demonstrator/CMS/NotExistingDevice/start",
      "SuperNEMO://Demonstrator/CMS/NotExistingDevice/stop"
    ]
  }
}
```

Figure 8: Proposal for a JSON serialized *connection rejection* object emitted by the CMS server to the Vire server (see Appendix C for details about error support).

1.5.2 Disconnection of the Vire server

Pre-condition:

- The Vire server must have an existing connection with the CMS server with MQ channels set.
- The CMS server must have an existing connection with a Vire server with the same MQ channels set and a valid resource map.
- All current synchronous resource execution should be terminated.

Description:

1. The Vire server constructs a `vire::mosinterface::disconnection_request` object and sends it to the CMS server.

   A typical JSON formatted *disconnection request* message is shown on Fig. 9. The exact format of the message will be discussed in appendix C.

JSON

```
{
  "setup_id" : {
    "name"    : "snemo",
    "version" : "1.0.2"
  }
}
```

Figure 9: Proposal for a JSON `vire::mosinterface::disconnection_request` object emitted by the Vire server to the CMS server.

2. The CMS server acknowledges the disconnection through a `vire::mosinterface::disconnection_request_success_response` object and disposes of the resource map associated to this connection.
3. The Vire server marks all resources addressed through the CMS server as *missing*.

Post-condition:

- The connection between the Vire server and the CMS server is terminated.
- The CMS server is ready to accept a new connection from a Vire server.
- The Vire server is ready to request a new connection with the Vire server.

1.5.3 Reconnection with the Vire server

NOTE (FM): To be discussed

1.5.4 Exception handling

Use case :

1. The CMS server meets a critical error and needs to stop
2. The CMS server sends an dedicated alarm to the Vire server
3. The Vire server invalidates the current active connection to the CMS and stops all processes related to the CMS (sessions with resources addresses through the CMS)

2 Representation of Vire resources with the CMS interface

2.1 Introduction

The *resource* is a central concept in the Vire API. A resource is unique and identified with an unique *path* in the Vire naming scheme.

Each resource is associated to a single operation that can be performed from some unique component (a *device*) of the experimental setup: it may consist in reading the current value of a datapoint (DP) for monitoring purpose, write a configuration value in another datapoint (control operation), invoke some transition method from a device implementing a finite state machine (FSM)...

Each resource can/must be classified in the *monitoring* or *control* category.

Examples:

- The resource identified with path:
  `"SuperNEMO://Demonstrator/CMS/Coil/PS/Control/Current/__dp_read__"`
  is conventionally classified in the *monitoring* category because its role is to fetch/read the current value of a datapoint accessible through a specific MOS server. Here the `.../Control/Current` identifier corresponds to a control datapoint but this resource only reads the value of the set point.

- The resource identified with path:
  `"SuperNEMO://Demonstrator/CMS/Coil/PS/Control/Current/__dp_write__"`
  is conventionally classified in the *control* category because its role is to set/write the value of a datapoint accessible through a specific MOS server. Here again the resource is associated to the `.../Control/Current` datapoint; however it is used to write the requested value of the set point.

- The resource identified with path:
  `"SuperNEMO://Demonstrator/CMS/Acquisition/stop"`
  is also classified in the *control* category because it corresponds to a method which performs a transition on a specific finite state machine (FSM) embedded in the MOS DAQ device.

The Vire server publishes a large set of resources for its clients. Some of these resources (but not all) are implemented and accessible through the CMS server because the devices they are related to are driven by some MOS servers. We thus need an interface to make these *MOS embedded* resources available in the Vire world.

A crucial point is to make sure that any resource known by Vire through its path can be identified properly in the OPCUA device naming scheme. This has been evoked in the previous section where the `devices_launch.conf` file was introduced (see Fig. 5).

2.1.1 Example : resources published by a FSM-like device

The SuperNEMO DAQ daemon/automaton (implemented as a OPCUA device) is managed through a dedicated MOS server. The DAQ device is addressed by the following set of parameters:

- MOS server: `192.168.1.15`
- MOS port: `48040`
- Device namespace: `CMS`
- Root device name: `DAQ` (from the `CMS` root in the OPCUA namespace)

We assume that two methods are publicly available from this device. These methods are supposed to be defined in the ICD[3] and the associated XML description of the device (for example: `${SNEMO_ONLINE_CONFIG_BASE_DIR}/snemo/1.0.2/cms/MOS/SNEMO/DAQ/DAQdevice_1.xml`):

- the `CMS.DAQ.start` method starts the data acquisition (FSM transition)
- the `CMS.DAQ.stop` method stops the data acquisition (FSM transition)

In this special case, Vire considers that each of these methods is a *resource* which can be executed. Mapping the DAQ daemon object in the OPCUA space as the `SuperNEMO://Demonstrator/CMS/Acquisition/` path in Vire's resource space, we can thus define two resources in Vire:

- the `SuperNEMO://Demonstrator/CMS/Acquisition/start` control resource is associated to the `192.168.1.15:48040:CMS.DAQ.start` method,
- the `SuperNEMO://Demonstrator/CMS/Acquisition/stop` control resource is associated to the `192.168.1.15:48040:CMS.DAQ.stop` method.

Vire behaves like a filesystem where the `192.168.1.15:48040:CMS.DAQ` OPCUA device is *mounted* at the Vire `SuperNEMO://Demonstrator/CMS/Acquisition` mount point. From this mount point, one may navigate and access to various devices and datapoints.

2.1.2 Example : resources from datapoints hosted in a device

The SuperNEMO coil power supply device is managed through a dedicated MOS server. This device is addressed by the following set of parameters:

- MOS server: `192.168.1.15`
- MOS port: `4841`
- Device namespace: `CMS`
- Root device name: `COIL_PS`

According to the description of the device (the `${SNEMO_ONLINE_CONFIG_BASE_DIR}/snemo/1.0.2/cms/MOS/SNEMO/COIL/COIL_PS_1.xml` file defined in the `${SNEMO_ONLINE_CONFIG_BASE_DIR}/snemo/1.0.2/cms/devices_launch.conf` file), the following four datapoints are published in the OPCUA space:

- `Monitoring.Current` with read access (*monitoring* only)
- `Monitoring.Voltage` with read access (*monitoring* only)
- `Control.Current` with read and write access (*monitoring* and *control*)
- `Control.Voltage` with read and write access (*monitoring* and *control*)

From the Vire point of view, this implies the existence of six resources with the following paths:

- `SuperNEMO://Demonstrator/CMS/Coil/PS/Monitoring/Current/__dp_read__`
- `SuperNEMO://Demonstrator/CMS/Coil/PS/Monitoring/Voltage/__dp_read__`

[3] Interface Control Document

- `SuperNEMO://Demonstrator/CMS/Coil/PS/Control/Current/__dp_read__`
- `SuperNEMO://Demonstrator/CMS/Coil/PS/Control/Voltage/__dp_read__`
- `SuperNEMO://Demonstrator/CMS/Coil/PS/Control/Current/__dp_write__`
- `SuperNEMO://Demonstrator/CMS/Coil/PS/Control/Voltage/__dp_write__`

Here the `__dp_read__` and `__dp_write__` names are defined in the Vire API. By convention they corresponds respectively to *read* and *write* methods (accessors) associated to a datapoint. It is the responsibility of the CMS server to translate them in term of MOS specific actions, respectively read and write the value of a given datapoint.
Vire mounts the `192.168.1.15:4841:CMS.COIL_PS` OPCUA device at the Vire `SuperNEMO://Demonstrator/CMS/Coil/PS` mount point.

2.2 Vire/CMS addressing scheme

The mapping of any Vire resource in terms of MOS addressing scheme is resolved by the CMS server through the rules defined in the `devices_launch.conf` file (see example on Fig. 5).

The CMS server is responsible to build a local resource map that associate any resource path to effective item (datapoint read/write action, methods) accessible from a MOS.

2.2.1 Example : resources from a FSM-like device

The Vire server defines the `SuperNEMO://Demonstrator/CMS/Acquisition/start` resource. From this unique Vire path, the CMS server must extract the corresponding *parent path* and *base name*.

- Parent Vire path: `SuperNEMO://Demonstrator/CMS/Acquisition/`
- Base name: `start`

Then the CMS server resolves the MOS address of the parent path using the informations stored in the `devices_launch.conf` file:

- MOS server: `192.168.1.15`
- MOS port: `48040`
- Device namespace: `CMS`
- Root device name: `DAQ`

Now the device is non ambiguously located in the OPCUA space.

The resource base name (here `start`) is thus identified/mapped as the `start` method of the DAQ device model.

The same procedure can be applied to the `SuperNEMO://Demonstrator/CMS/Acquisition/stop` resource.

2.2.2 Example : resources from a datapoint hosted in a device

The Vire server defines the `SuperNEMO://Demonstrator/CMS/Coil/PS/Monitoring/Voltage/__dp_read__` resource associated to the `.../Voltage` monitoring datapoint.

The CMS server must extract the corresponding *parent path* and *base name*.

- Parent Vire path: `SuperNEMO://Demonstrator/CMS/Coil/PS/Monitoring/Voltage/`
- Base name: `__dp_read__`

Then the CMS server resolves the MOS address of the parent path using the informations stored in the `devices_launch.conf` file:

- MOS server: `192.168.1.15`
- MOS port: `4841`
- Device namespace: `CMS`
- Root device name: `COIL_PS.Monitoring.Voltage`

The Vire mount point is extracted from the Vire parent path: `SuperNEMO://Demonstrator/CMS/Coil/PS/`. This allows to identify the top level device handled by the MOS server (`CMS.COIL_PS`) then find the sub-path to the datapoint `Monitoring.Voltage`. Now the datapoint is non ambiguously located in the OPCUA space. The resource base name (here `__dp_read__`) is identified as the conventional operation that reads the current cached value of the datapoint.

2.2.3 CMS map of resources

As soon as the CMS server has accepted the connection with the Vire server, and following the rules explained above, it's possible to build a map of all resources requested by the Vire server. The map is built from the list of requested resources and the contents of the `devices_launch.conf` file.

Once the connection with the Vire server has been established, each time the CMS server will receive a request message about a resource, it will immediately know, thanks to this map, what device or datapoint is targeted. Table 2 displays an excerpt of such a map.

2.2.4 Dynamic informations associated to the resources

Due to the realtime and hardware nature of the devices that publish them, each resource is assigned some flags that reflect its current status. For now we have defined four *status flags*:

- the *missing* flag is set when the resource is not present or accessible from the CMS.

  Typically, this occurs when the device that publishes the resource is not accessible because of a temporary disconnection of the MOS server or a communication error between the MOS server and the device. It is obvious that the disconnection of a device automatically sets the missing flag for all resources associate to this deviced.

  Example: Assume the MOS running at `192.168.1.15:4841` and hosting the `CMS.COIL_PS` device is disconnected from the CMS server. Then automatically all resources identified with a path starting with `"SuperNEMO://Demonstrator/CMS/Coil/PS/"` will be tagged as *missing*.

Vire resource	**M/C**	**OPCUA address & name**	**Type of object**	**Operation**
`SuperNEMO://Demonstrator/CMS/Acquisition/start`	C	`192.168.1.15:48040:CMS.DAQ`	Device	`start` method
`SuperNEMO://Demonstrator/CMS/Acquisition/stop`	C	`192.168.1.15:48040:CMS.DAQ`	Device	`stop` method
`SuperNEMO://Demonstrator/CMS/Coil/PS/Monitoring/Voltage/__dp_read__`	M	`192.168.1.15:4841:CMS.COIL_PS.Monitoring.Voltage`	Datapoint	read the value
`SuperNEMO://Demonstrator/CMS/Coil/PS/Monitoring/Current/__dp_read__`	M	`192.168.1.15:4841:CMS.COIL_PS.Monitoring.Current`	Datapoint	read the value
`SuperNEMO://Demonstrator/CMS/Coil/PS/Control/Voltage/__dp_read__`	M	`192.168.1.15:4841:CMS.COIL_PS.Control.Voltage`	Datapoint	read the value
`SuperNEMO://Demonstrator/CMS/Coil/PS/Control/Voltage/__dp_write__`	C	`192.168.1.15:4841:CMS.COIL_PS.Control.Voltage`	Datapoint	write the value
`SuperNEMO://Demonstrator/CMS/Coil/PS/Control/Current/__dp_read__`	M	`192.168.1.15:4841:CMS.COIL_PS.Control.Current`	Datapoint	read the value
`SuperNEMO://Demonstrator/CMS/Coil/PS/Control/Current/__dp_write__`	C	`192.168.1.15:4841:CMS.COIL_PS.Control.Current`	Datapoint	write the value

Table 2: Example of a map of resources with their correspondances with OPCUA datapoints read/write access and devices methods.

- the *disabled* flag is set when the resource is temporary not *executable* because of the current context. This occurs for example when a finite state machine is in some specific internal state, preventing some transitions to be invoked on this FSM.

  Example: Assume the DAQ device, which implements an internal finite state machine, is in the *running* state. It is expected in this case that the `start` transition method cannot be used, whereas the `stop` transition method can be invoked at any time.

  From the point of view of Vire resources:
  – the `SuperNEMO://Demonstrator/CMS/Acquisition/start` resource is thus *disabled*
  – and the `SuperNEMO://Demonstrator/CMS/Acquisition/stop` resource is *enabled*.

  As soon as the internal state of the DAQ daemon is changed to *stopped* thanks to a call to the `stop` transition, the `SuperNEMO://Demonstrator/CMS/Acquisition/start` resource is *enabled* again while the `SuperNEMO://Demonstrator/CMS/Acquisition/stop` resource is tagged with the *disabled* flag.

- the *pending* flag is set when the resource **is** executing. This occurs when the execution of the resource cannot be considered as instantaneous and implies some operations with a long latency time. The CMS server determines that it will have to wait for a while before the full completion of the resource execution and the making of a proper response (success/failure/output parameters...) to the caller. It thus sets the *pending* flag associated to the resource as long as the processing is not terminated.

  Example: Assume the Vire server requests the execution of the `SuperNEMO://Demonstrator/CMS/Acquisition/start` resource at time t_0. The CMS server invokes the `start` method of the `192.168.1.15:48040:CMS.DAQ` OPCUA device. Given that the execution of this method lasts several seconds, no acknowledge/response from the MOS server is expected before an arbitrary time t_1. Thus the CMS server sets the *pending* flag on the `SuperNEMO://Demonstrator/CMS/Acquisition/start` resource during the time interval $[t_0; t_1]$. As soon as the `start` method is proven terminated (at time t_1), the *pending* flag is unset.

- the *error* flag is set when the execution of a resource has made the device in some error state.

  BAd bad bad Example:

  The Vire server requests the execution of the `SuperNEMO://Demonstrator/CMS/Coil/PS/Control/Voltage/__dp_write__` control resource with a value of the voltage which is out of the valid range for this model of power supply device: V=5000 Volts $>$ V_{max}=20 Volts. Of course, the MOS server will reject this request and the CMS server will set the *error* flag and maintain it as long as no other successful execution of the resource is done. This flag indicates that the last control operation on the resource has failed.

  NOTE: should we also make possible to store some (last) error code/message ?

Given these four boolean flags, it is possible for the CMS server to build a bitset that reflects the current status of the resource at any time. Accessing this status bitset for each resources, any client of the CMS server (the Vire server application or some Vire client applications) can figure out the realtime conditions to access resources.

2.2.5 Actions on resources

We consider now the typical case where the Vire server (or Vire clients connected to the CMS server) access the devices and datapoints in polling mode. Basically, the Vire server can request three kinds of action on any resource managed by the CMS server:

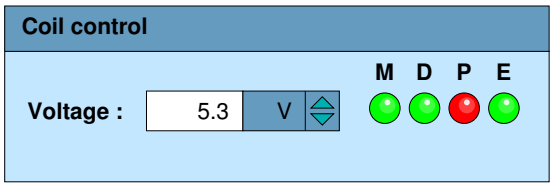

Figure 10: An example of a GUI component which displays the status bits associated to a resource, informing the user of the realtime conditions. Here the **M**issing flag is not set, reflecting that the resource is available to the system typically because the associated device is recognized by the MOS and is currently running with all its functionnalities. The **D**isable flag is not set which means the `__dp_write__` resource associated to this datapoint can be used to set the voltage set point. The **P**ending flag is set (red light) which means that a previous write operation is currently processing and not terminated. Finally, the **E**rror flag is not set, reflecting that no specific problem was formerly detected with this datapoint's resource.

- executing the resource in *blocking* mode,
- executing the resource in *non blocking* mode,
- fetching the status of the resource.

Execute a resource in *blocking* mode: The execution of a resource is equivalent to the invocation of some specific device method or datapoint accessor (get/set OPCUA method). The Vire server is expected to request the execution of a resource with a specific `resource_exec_request` object.

```
struct resource_exec_request
{
  std::string path; ///< Identifier of the resource
  std::vector<argument_type> input_arguments; ///< The list of input arguments
                                              ///  The exact definition of this list depends
                                              ///  on the nature of the resource.
}
```

The Vire server waits for the response of the CMS server until the operation has been completed or failed.

Typical JSON formatted *resource execution request* messages are shown on Fig. 11, 12 and 13.

JSON

```
{
  "path" : "SuperNEMO://Demonstrator/CMS/Acquisition/start",
  "input_args" : []
}
```

Figure 11: Proposal for typical JSON *resource execution* messages emitted by the Vire server to the CMS server. Here the resource corresponds to some methods that takes no input arguments.

The CMS server must decode the message, interpret it and, if it makes sense, launch the execution of the requested operation in the targeted MOS. When it receives the answer from the MOS, it is supposed to build and send a response message to the caller.

JSON

```
{
    "path" : "SuperNEMO://Demonstrator/CMS/Coil/PS/Control/Current/__dp_write__",
    "input_args" : [
      { "setpoint" : "0.341" }
    ]
}
```

Figure 12: Proposal for a JSON *resource execution* message emitted by the Vire server to the CMS server. Here the resource corresponds to some method that takes one input argument.

JSON

```
{
    "path" : "SuperNEMO://Demonstrator/CMS/Coil/PS/Control/Current/__dp_read__",
    "input_args" : [
    ]
}
```

Figure 13: Proposal for a JSON *resource execution* message emitted by the Vire server to the CMS server. Here the resource corresponds to some method that takes no input argument.

In cas of successful execution, a `resource_exec_success_response` object is built by the CMS server and delivered back to the Vire server.

```
struct resource_exec_success_response
{
    "path" : "SuperNEMO://Demonstrator/CMS/Acquisition/start",
    "status" : "0000",

}
```

A typical JSON formatted *resource_ exec_ success_ response* message is shown on Fig. 14 and 15.

JSON

```
{
  "resource_status_record" : {
    "path" : "SuperNEMO://Demonstrator/CMS/Acquisition/start",
    "status" : "0100"
  },
  "reception_timestamp" : "20160203T175203.031743801",
  "completion_timestamp" : "20160203T175217.56143801",
  "output_args" : [
  ]
}
```

Figure 14: Proposal for a JSON *resource execution success response* message emitted by the CMS server to the Vire server. This is typical case where the method resource has been successful. Here the response does not contain any output arguments. Note that the CMS server also informs the Vire server about the reception timestamp of the resource execution message as well as the timestamp at execution completion/failure.

```
struct resource_status_record_type
```

JSON

```
{
  "resource_status_record" : {
    "path" : "SuperNEMO://Demonstrator/CMS/Coil/PS/Control/Current/__dp_read__",
    "status" : "0000",
  }
  "reception_timestamp" : "20160203T175203.031743801",
  "completion_timestamp" : "20160203T175217.56143801",
  "output_args" : [
    "value" : "1.23"
  ]
}
```

Figure 15:

```
{
  std::string path;
  uint16_t    status;
};

struct resource_exec_failure_response
{
   resource_status_record_type resource_status_record;
   error_type_id error_type;
   XXX_error error;
};
```

In case of failure the response failure object will update the status of the resource (Fig. 16).

JSON

```
{
  "resource_status_record" : {
      "path" : "SuperNEMO://Demonstrator/CMS/Acquisition/start",
      "status" : "0000",
   }
   "error_type_id" : {
      "name" : "basic_error",
      "version" : "1.0"
   },
   "error" : {
      "code" : "1"
      "message" : "Operation failed because the DAQ device is already running."
    }
    "reception_timestamp" : "2016-02-03 17:52:03.031743801+01:00",
    "completion_timestamp" : "2016-02-03 17:52:17.56143801+01:00",
  }
}
```

Figure 16: Proposal for a JSON *resource execution response* message emitted by the CMS server to the Vire server. This is typical case where the execution of the resource has failed because the resource was not available when invoked.

List of error object types embedded in the `resource_exec_failure_response`:

- `unknown_resource_error`

- `missing_resource_error`
- `disable_resource_error`
- `pending_resource_error`
- `failed_resource_error`
- `execution_resource_error` (invalid param name, value, context)

NOTE (FM): Management of timeout: Should the Vire server add a timeout info in the resource_exec message to set a limit to the response waiting time ? if the cms is not able to answer within the timeout, an error response message is sent back to the Vire server.

Execute a resource in *non blocking* mode: This part will be detailed later.

Fetch the status bitset of a resource: At any time, the Vire server can ask the CMS server to send the current status of a resource. Typical JSON formatted *resource fetch status* messages are shown on Fig. 17 and 18 (for the response message).

struct

JSON

```
{
  "resource_fetch_status": {
    "path" : "SuperNEMO://Demonstrator/CMS/Coil/PS/Control/Current/__dp_read__"
  }
}
```

Figure 17: Proposal for a JSON *resource fetch status* message emitted by the Vire server to the CMS server.

JSON

```
{
  "resource_fetch_status_response": {
    "path" : "SuperNEMO://Demonstrator/CMS/Coil/PS/Control/Current/__dp_read__",
    "timestamp" : "2016-02-03 17:52:03.031743801+01:00",
    "status" : "0010"
  }
}
```

Figure 18: Proposal for a JSON *resource fetch status* response message emitted by the CMS server to the Vire server.

Activate pub/sub of a resource: On service

```
struct resource_pubsub_activation_request
{
  std::string path; ///< Resource path
  bool activation;  ///< Pub/Sub activation flag
};

struct resource_pubsub_activation_success_response
{
  "resource_status_record" : {
    "path" : "...",
    "status" : "0000"
  },
  "binding_key" : "SKJH_327362kxS"
};

struct resource_pubsub_activation_failure_response
{
  "resource_status_record" : {
    "path" : "...",
    "status" : "0000"
  },
  "error_type_id" : {
    "name" : "no_pubsub_resource_error",
    "version" : "1.0"
  },
  "error" : {
    "code" : "1",
    "message" : "Pub/sub support is not
  }
};
```

3 Access to the CMS server by Vire clients

TODO

4 Access to the Vire server by Vire clients

TODO

A Filesystem and configuration files management

Let's consider a simple situation where one runs the Vire and CMS software tools (servers) on a single Linux machine (the CMS host) under the `"nemoprod"` generic account[4].

- Hostname login : `192.168.1.10` (private IP)
- User login : `nemoprod`
- Main group : `supernemo`
- Home directory : `/home/nemoprod` (a.k.a. `~nemoprod`)

We assume that the SuperNEMO online software has been installed and setup in the home directory, for example in `/home/nemoprod/Private/Software/` :

Filesystem

```
/home/nemoprod/Private/Software
|-- Cadfael/ # base directory of the Cadfael software framework
|-- Bayeux/  # base directory of the Bayeux software framework
|-- Vire/    # base directory of the Vire software framework
|-- OPCUA/   # base directory of the OPCUA+MOS software framework
`-- Falaise/ # base directory of the Falaise software framework
```

We consider here that the Falaise library package will contain the mandatory configuration files that describe the online software, both for the Vire and CMS/MOS parts:

Filesystem

```
/home/nemoprod/Private/Software
:
`-- Falaise/
    :
    `-- Install/
        `-- Falaise-3.0.0/
            |-- bin/
            |   :
            |   |-- flquery
            |   |-- flreconstruct
            |   `-- flsimulate
            |-- include/
            :   :
            |-- lib/
            |   `-- x86_64-linux-gnu/
            |       :
            |       |-- libFalaise.so
            |       :
            `-- share/
                `-- Falaise-3.0.0/
                    `-- resources/
                        `-- config/
```

[4] `"nemoprod"` is the login of th generic account used at the CCIN2P3 cluster to perform automated management operations on experimental and Monte-Carlo data file: data transfer from LSM or LSC labs to CCIN2P3, calibration and reconstruction data processing, storage on HPPS.

```
                                `-- online/
                                    :
                                    :
```

Where:

- the `/home/nemoprod/Private/Software/Falaise/Install/Falaise-3.0.0` is the installation prefix of the Falaise library (binaries, includes) and associated resource files.
- the `share/Falaise-3.0.0/resources/config/online/` subdirectory is the tree of configuration files that should be accessible by any online software component (Vire server, Vire clients, CMS and MOS servers).

Let's consider the `.../config/online/` directory as the base directory for all online configuration files for the Vire and CMS servers. All configuration files should thus be addressed relatively to this place. We propose to use one of the following techniques to represent this base directory:

- a dedicated environment variable `SNEMO_ONLINE_CONFIG_BASE_DIR` recognized by both Vire and CMS servers. It could be setup within the environment with:

shell
```
export FALAISE_INSTALL_DIR=\
  ${HOME}/Private/Software/Falaise/Install/Falaise-3.0.0
export SNEMO_ONLINE_CONFIG_BASE_DIR=\
  ${FALAISE_INSTALL_DIR}/share/Falaise-3.0.0/resources/config/online
```

- a path registration label as implemented in the kernel of the Bayeux library:
  The `@snonlinecfg:` label
  is associated to
  `${FALAISE_INSTALL_DIR}/share/Falaise-3.0.0/resources/config/online`

Thus a specific configuration file `dummy.conf` could be addressed with one of the following syntaxes:

(a) `${SNEMO_ONLINE_CONFIG_BASE_DIR}/snemo/1.0.2/dummy.conf` : supported by Vire and CMS using word expansion utility like `wordexp` (for C/C++ languages),

(b) `@snonlinecfg:snemo/1.0.2/dummy.conf` : supported by Vire only for now, thanks to the path registration mechanism implemented in the Bayeux API,

(c) `snemo/1.0.2/dummy.conf` : can be supported by Vire and CMS but is ambiguous because such a relative path can be also interpreted as a path relatively to the current directory (`./`) and not to the online configuration directory.

We suggests the use of an explicit environment variable as in (a) because it is simple to implement in various languages and software frameworks and should not imply any ambiguous file path resolution.

B Integration of a new device

Both Vire and MOS systems are designed to be expandable in terms of device integration. This section decribes the integration of a new device in the *Control and Monitoring System*.

B.1 Integration of a new device in the MOS environment

Any new device is described through a dedicated XML model file. This XML file is created from a template file elaborated from the *interface control document* (ICD) and associated to the model of the device. The format of the XML file is described in the MOS (Multipurpose OPCUA Server) User Guide.

Typically, a device is embedded in a OPCUA server and implemented as a OPCUA *simple device*. The OPCUA server itself is located through an unique dedicated IP address and port.

The simple device instance, hosted in the OPCUA server, may contains other sub-devices and/or *datapoints*. It is thus considered as the root of a hierarchy of daughter objects at deeper levels. The daughter objects (devices or datapoints) are named relatively to their top level parent device. Figure 19 shows an example of a device embedded in a MOS server.

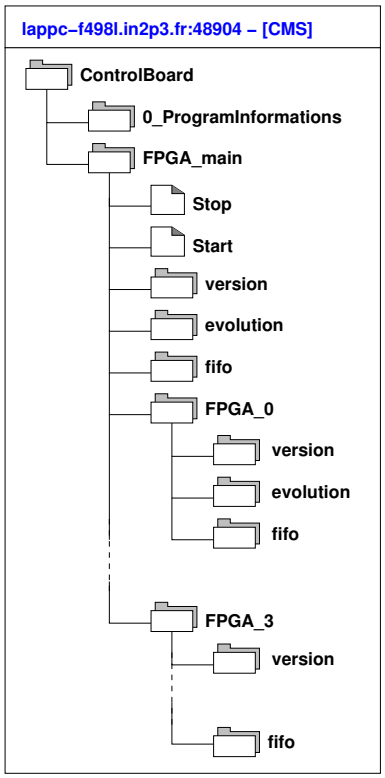

Figure 19: Example of a device managed through a MOS server. The root device is named `ControlBoard`. First level daughter devices are `0_ProgramInformations` and `FPGA_main`. Here the MOS/OPCUA server is labelled `CMS`.

B.2 Integration of a new device in the Vire environment

The Vire API also implements a mechanism to describe a hierarchy of devices. This mechanism is independant of the one used in the MOS system but can be easily made compatible with it. This means that a MOS hierarchy of devices can be represented in Vire. The Vire hierarchy of devices can be considered as some kind of filesystem, each device being a folder with its unique path, as shown on figure 20. The *methods* associated to a devices (or a datapoint) can be considered as plain executable files stored in the device's folder : they constitute the set of *resources* associated to the device.

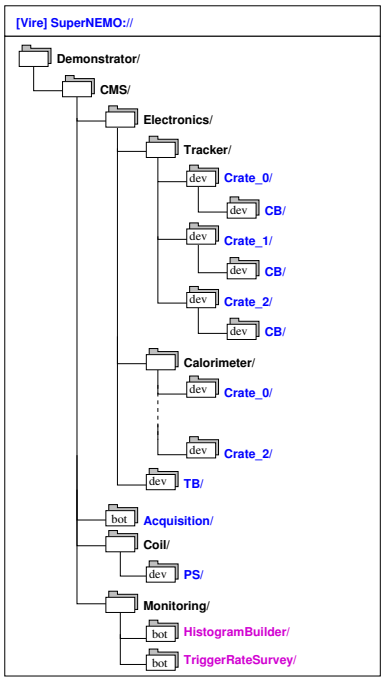

Figure 20: Example of a hierarchy of devices described by the Vire API. The root device is named `SuperNEMO:`. The top level (root) device is named `Demonstrator`. The devices colored in blue are managed through MOS/OPCUA. The devices colored in magenta are directly embedded in the Vire server. Devices with the `dev` tag are typical hardware device. Devices with the `bot` tag are typical software devices. The devices colored in **black** are structural pseudo-devices used to organize and present a comprehensive view of the hierarchy.

The organisation of this hierarchy of devices is arbitrary and defined by the designer of the *Control and Monitoring System*. What is important to understand is that some of these devices can be associated to *hardware devices* (a power supply crate, a temperature probe...) and others can be *pseudo-devices*, i.e. pure software object (a monitoring robot, a file transfer daemon...).

In the context of the coupling of the Vire server and the CMS server, we are in the event that some devices are managed by some MOS/OPCUA servers and others are managed in the Vire server itself. Typically, *hardware devices* are systematically managed through the OPCUA technology. Vire has a mechanism to integrate such devices in its own hierarchy. This mechanism can be considered like the *mounting* of a remote filesystem from a local filesystem. Figure 21 illustrates the case of many hardware devices – managed by MOS – that are integrated in the Vire system. From the Vire point of view, the user does not see the implementation details for such devices. He does not know the identity of the MOS server hosting the device. He does not even know if the device is hosted by a

MOS server. Devices are simply visible through the standard hierarchy published by Vire with its own device naming scheme, regardless their true location.

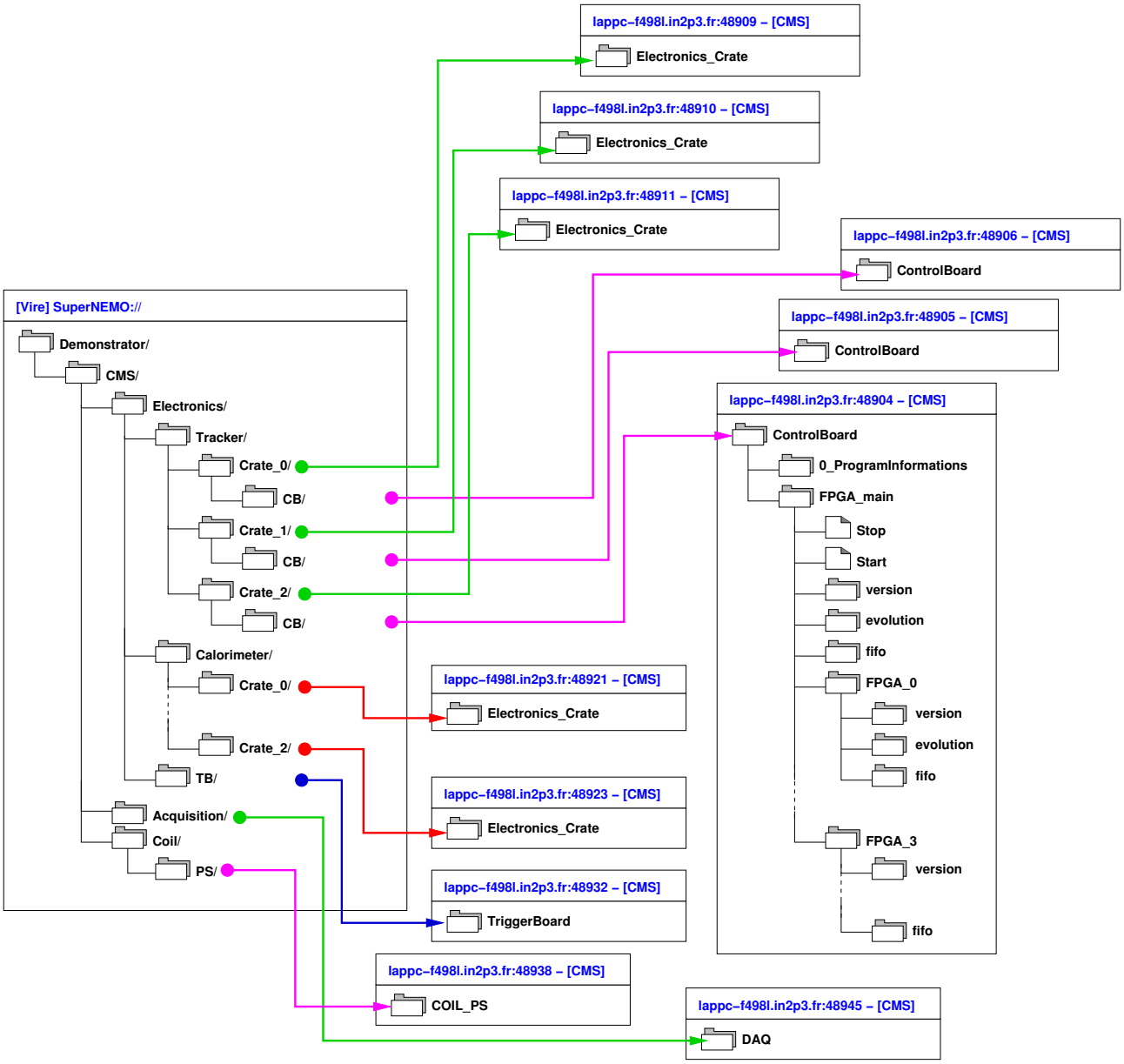

Figure 21: The mounting of many MOS device hierarchies in the Vire device hierarchy. Each OPCUA server runs a simple hardware device that is *mounted* from a specific node with its own path.

B.3 Example

Using the examples displayed in figure 21, we consider in detail the way one specific device managed by MOS is mounted in the Vire hierarchy. Figure 22 illustrates the mounting of a MOS device in Vire.

Here the Vire server publishes the path of a device representing the control board of the third electronic crate for the tracker of the SuperNEMO demonstrator module. The full Vire path of this

device is:

SuperNEMO://Demonstrator/CMS/Electronics/Tracker/Crate_2/CB

This is the only Vire identifier recognized by user to address this device.

On the figure, one can see that the MOS server `lappc-f498l.in2p3.fr` (port 48904) hosts a simple device which is locally named `ControlBoard`.

When mounting this device in the Vire hierarchy, the local `[CMS]` namespace and `ControlBoard` device names are hidden and replaced by the Vire device path. All daughter devices and datapoints of the `CMS/ControlBoard` device are integrated as daughters of the Vire device named `SuperNEMO://Demonstrator/CMS/Electronics/Tracker/Crate_2/CB`.

For example, the `FPGA_main` daughter device is now associated to the following Vire path:

SuperNEMO://Demonstrator/CMS/Electronics/Tracker/Crate_2/CB/FPGA_main/

and its `Stop` method is automatically addressed with the following *leaf* path:

SuperNEMO://Demonstrator/CMS/Electronics/Tracker/Crate_2/CB/FPGA_main/Stop

B.4 Vire/MOS mapping

As it can be seen in the above example, the integration of a new MOS device in the Vire system is achieved through soem kind of filesystem mounting operation. Particularly, it is shown that the MOS name of the mounted root device is replaced by an arbitrary Vire path. However, all daughter nodes (devices, datapoints) attached from this root node have their relative MOS names preserved in the Vire naming scheme.

Any resource (method) associated to any of such daughter nodes inherits this relative naming scheme.

As Vire applications describe resources through their Vire paths, it is thus needed to build an explicit map that associates resource paths to MOS address and name. The CMS server will be able to resolve the MOS server/port and embedded device associated to the resource path.

The goal of the `devices_launch.conf` file is not only to tell the CMS server what MOS server should be loaded and ran at start, but also to describe the *mounting point/names* used by Vire to access the resources associated to MOS devices. From the informations stored in the file, an explicit associative array must be built when the Vire server connect to the CMS server. It will play the role of a resource path resolver when requests about resources will be sent by Vire applications. This associative array must be locked during the Vire/CMS connection.

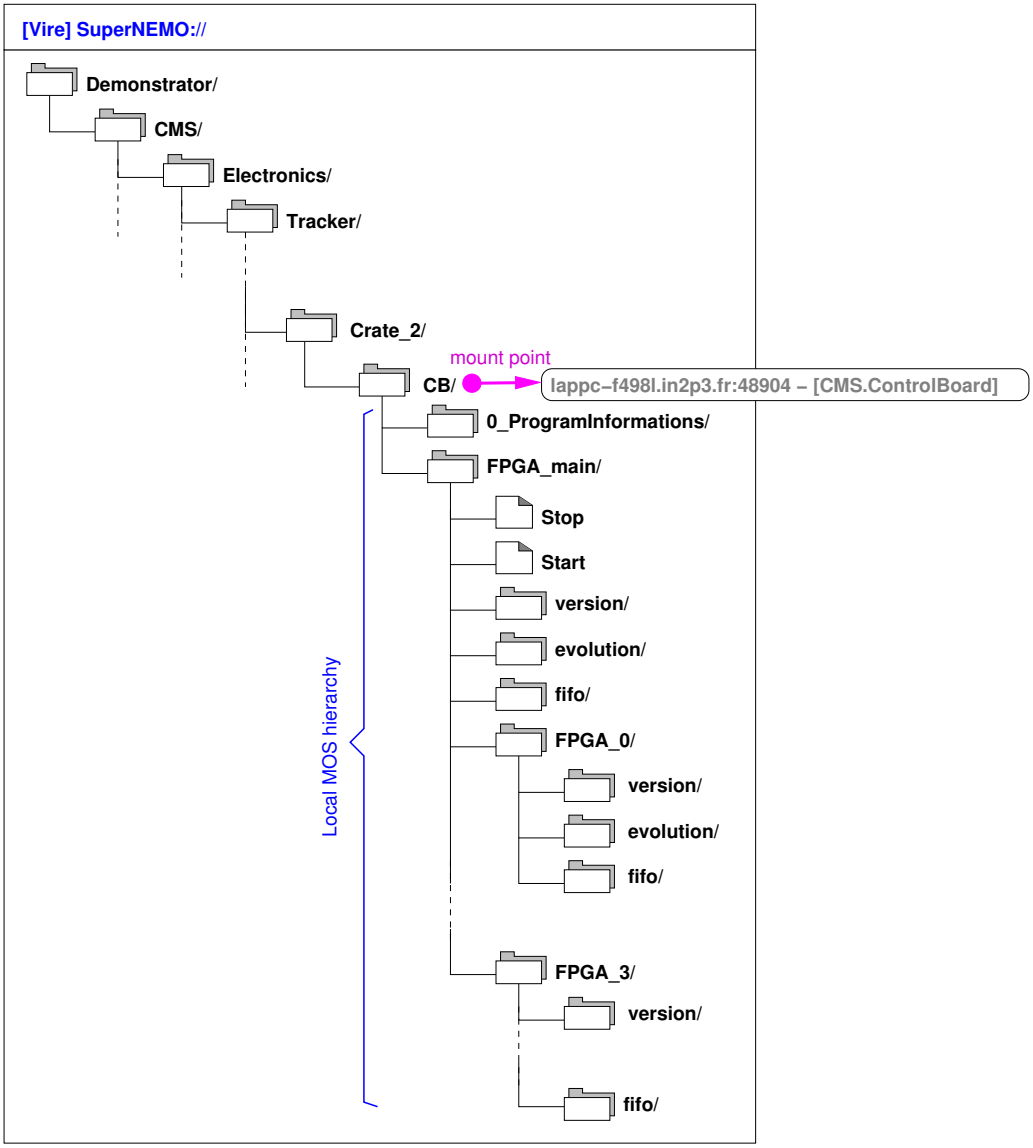

Figure 22: The mounting of one MOS device and its local hierarchy in the Vire device hierarchy.

C Vire messages and JSON formatting

Within Vire and between Vire components and external components, we use a communication system based on JSON-formatted messages. This section describes the structure and the formatting of such messages.

C.1 General structure of a message

Each message consists in two parts (figure 23):

- the *header* is dedicated to generic informations which document the message itself.
- the *body* of the message contains the real data. Its encoding/decoding depends on the format specified in the header.

C++

```
struct message {
  header h; ///< Header of the message
  body   b; ///< Body of the message
};
```

Figure 23: The structure of a message object (C++).

C.2 The message header

The header contains (figure 24):

- The mandatory `message_id` key contains an identifier of the message which document the emitter and a unique message number. Each emitter is responsible of the numbering of the messages it emits, typically using an incremental technique. The message number is a positive integer, starting from 0 (figure 25).
- The `format_id` key contains the identifier of the format used to encode the body section. It is a mandatory character string. The default format for message transmission inside the Vire API is `"vire::message::format::basic"` version `"1.0"` (figure 26).
- The `timestamp` key contains a character string that documents the approximative time point when the message was created. It uses the following format: `yyyymmddThhmmss.uuuuuu` where:

`yyyymmdd` : encodes year/month/day,

`hhmmssd` : encodes hour/minute/second,

`uuuuuu` : encodes microseconds.

- In the case of a *response* message, the `in_reply_to` attribute is set to identify the associated request message.

C++

```
struct header {
  message_identifier message_id; ///< Message identifier from the emiter
  format_identifier  format_id;  ///< Format identifier
  std::string        timestamp;  ///< Timestamp
  message_identifier in_reply_to; ///< Message identifier of the associated request message
};
```

Figure 24: The structure of a message header object (C++).

C++

```
struct message_identifier {
  std::string emitter; ///< Name identifying the emitter of the message
  int32_t     number;  ///< Number identifying the message in the emitter's
                       ///  message numbering scheme
};
```

Figure 25: The structure of a message identifier (C++).

C++

```
struct format_identifier {
  std::string name;    ///< Name identifying the format of the message
  std::string version; ///< String identifying the version of the format
};
```

Figure 26: The structure of a message format identifier (C++).

JSON

```
{
  "header" : {
    "message_id" : {
      "emitter" : "vire.server",
      "number"  : "723"

    },
    "format_id"  : {
      "name" : "vire.messaging.basic",
      "version" : "1.0"
    },
    "timestamp" : "20160516T085346",
    "in_reply_to" : {
      "emitter" : "vire.client",
      "number"  : "843"
    }
  },
  "body" : {
    ... special encoding for the data...
  }
}
```

Figure 27: Example of the header of a JSON formatted message.

C.3 The message body

The default message format in Vire is named `"vire.messaging.basic"` (version `"1.0"`). Each message used within the Vire framework is supposed to use this format. The general idea is that the body of the message encodes an *object* that has to be transmitted between two components of the system. The requirements for the transmitted object are the following:

- The type of the object must be conventionally associated to a *string identifier*, possibly with a *version number*. Each software component that may send or receive the object should agree on this type identification scheme.
- For each software component, the object type must have a JSON encoding/decoding functions available (whatever programming language is used).

Vire uses a dedicated format to encode/decode the body of any message. The structure of the body contains two kinds of informations (figure 28):

1. The `object_type_id` specifies the type of the encoded object (figure 29). This unique name is conventionaly fixed for a given application. A version tag allows to support possible evolution of the object type.
2. The `object_serialization_buffer` is a character string that encapsulated the JSON-serialization stream of the encoded object.

   - Within the producer component of the message, the encoding function associated to the object type is responsible to generate the JSON stream for the object and store it in the buffer.
   - Within the consumer component of the message, the decoding function associated to the object type is responsible to parse the JSON stream stored in the buffer and restore the object in memory.

   It is expected that, on both sides of the connection, the software components may access to dedicated plugins associated to specific object types conventionnaly associated to their type identifiers and JSON encoding/decoding methods. The system allows to support modification in the structure of the objects thansks to version tagging.

C++

```
struct body {
  type_identifier object_type_id; ///< Object type identifier
  std::string     object_serialization_buffer;
                                  ///< Character string buffer for
                                  ///  JSON-based encoding of the transmitted object
};
```

Figure 28: The structure of a message body object (C++).

Figure 30 shows the formatted body of a message which encodes a connection request object from the Vire server to the CMS server.

C++

```
struct type_identifier {
  std::string name;    ///< Name uniquely identifying the type of object
  std::string version; ///< Character string representing the version
                       ///  of the object type
};
```

Figure 29: The structure of the type identifier object related to an encoded object (C++).

JSON

```
{
  "header" : {
     ...
  },
  "body" : {
    "object_type_id" : {
      "name"    : "vire::cmsserver::connection_request",
      "version" : "1.0"
    },
    "object_serialization_buffer" : {
      ... JSON serialization of the object ...
    }
  }
}
```

Figure 30: Example of the body of a JSON formatted message.

C.4 Vire library of serializable objects

C.4.1 Vire server to CMS server communication

`vire::cmsserver::connection_request` (version `1.0`)

C++

```
struct connection_request {
  std::string setup_id;      ///< Name uniquely identifying the experimental setup
  std::string setup_version; ///< Character string representing the version
                             ///  of the experimental setup
  std::vector<std::string> requested_resources;
                            ///< The list of requested resources addressed by path
};
```

Figure 31: The structure of the coonection request object to be emitted by the Vire server to the CMS server (C++).

`vire::cmsserver::connection_request_success_response` (version `1.0`)

C++

```
struct resource_snapshot
{
  std::string path;   ///< Path of the requested resource
  uint32_t    status; ///< Status bits (Missing/Disables/Pending/Error)
};

struct connection_request_success_response {
  std::vector<resource_snapshot> resource_snapshots;
                            ///< The list of requested resources snapshots
};
```

Figure 32: The structure of the coonection request success response to be emitted by the CMS server to the Vire server (C++).

`vire::cmsserver::connection_request_error_response` (version `1.0`)
`vire::cmsserver::disconnection_request` (version `1.0`)

C++

```
struct error
{
  int         error_code;
  std::string what;
}

struct invalid_setup_id_error : error {
  std::string invald_setup_id;
};

struct invalid_setup_version_error
{
 std::string invald_setup_version;
};

struct unknown_resource_error
{
  std::vector<std::string> unknown_paths;
};

struct connection_request_error_response {

// one of the above

};
```

Figure 33: The structure of the coonection request success response to be emitted by the CMS server to the Vire server (C++).