The Vire-CMS interface
version 0.3

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

March 1, 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).

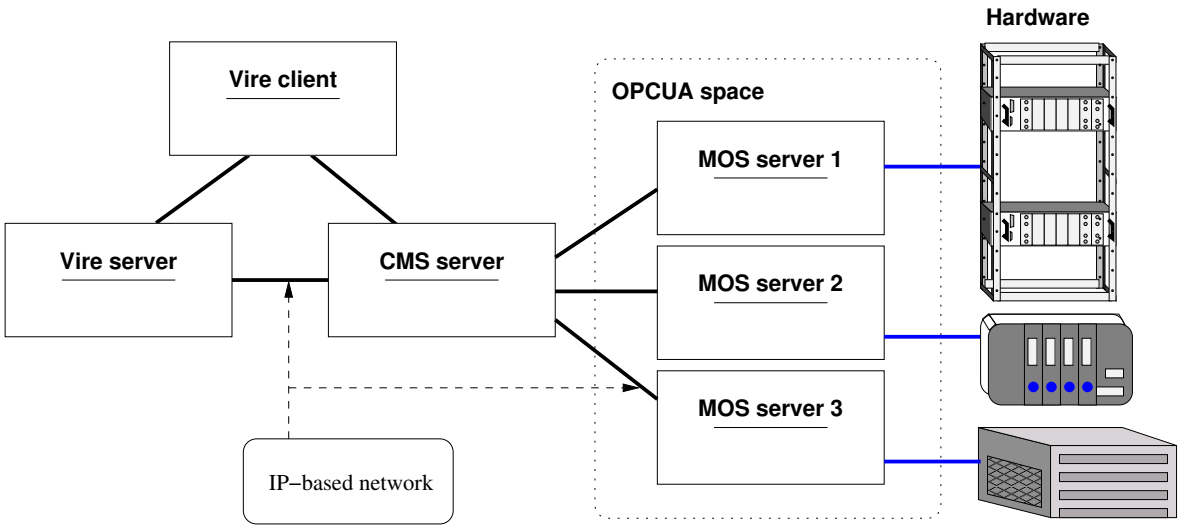

Figure 1: Overview of the Vire server, a Vire client, the CMS server and its connections to the hardware through dedicated MOS servers.

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 represents the unique identifier label of the experimental setup.
- The `setup_version` character string represents the version number of the experimental setup. It should typically use a conventional sequence-based identifier format (*major.minor.revision* scheme).
- The `amqp_svr_host` character string represents the IP address or URL of the main RabbitMQ server used for communication between remote components involved in the system.
- The `amqp_svr_port` integer represents the port used by the main RabbitMQ server.
- The `amqp_cms_gate_channel` character string represents the identifier of the *gate* channel used by the Vire Server to request a connection with the CMS server.
- 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.

[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`

Configuration

```
setup_id           = "snemo"
setup_version      = "1.0.2"
amqp_svr_host      = "192.168.1.10"
amqp_svr_port      = 5672
amqp_cms_gate_channel = "snemo.cms.gate"
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).
A possible scheme could use three 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, log messages, alarm messages... but no message concerning datapoint (DP) read/write operations or invocation of devices methods.

2. the *control* channel is dedicated to the execution of control operations on devices and datapoints through explicit *control* resources (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 (read datapoint values...).

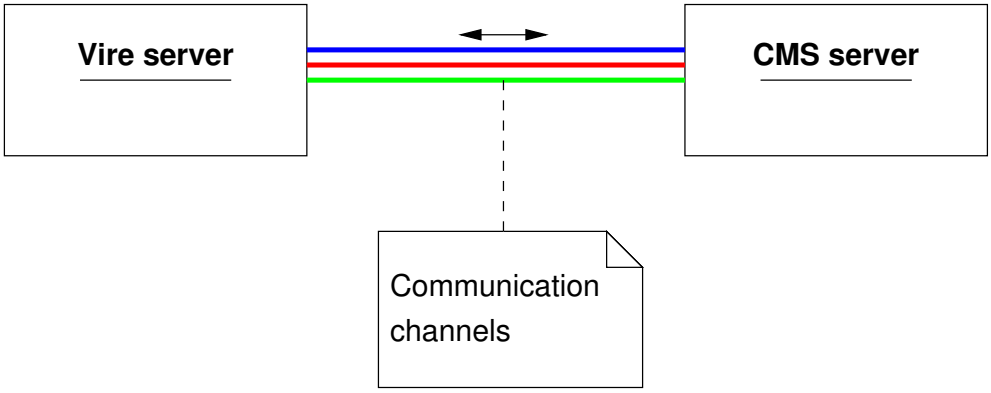

Figure 3: The communication channels between the CMS server and the Vire server.

We plan to use an AMQP RabbitMQ server as the communication framework between the Vire server and the CMS server (Fig. 4).

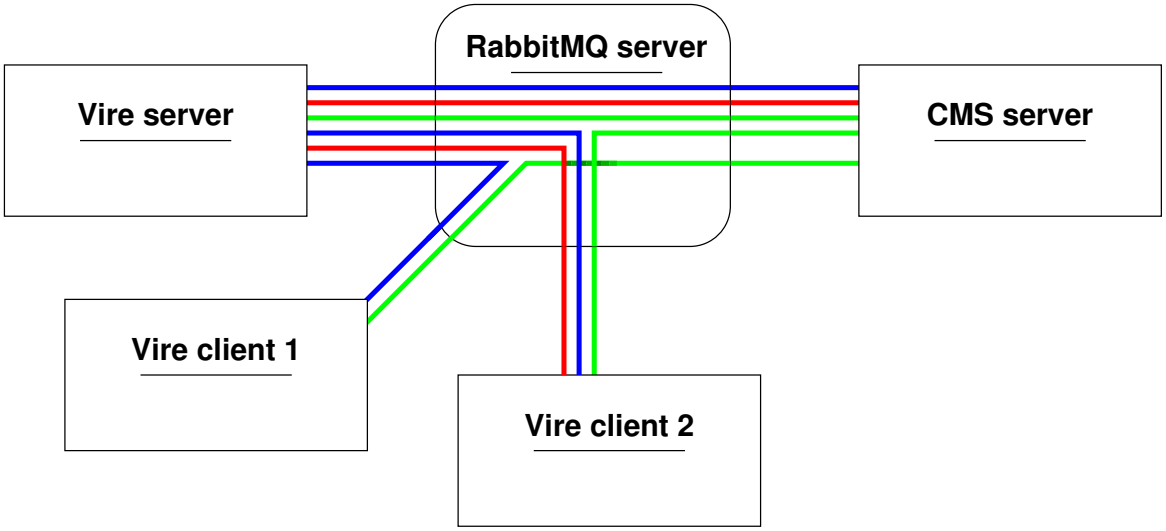

Figure 4: A RabbitMQ based communication framework shared by the CMS server, the Vire server and some arbitrary Vire clients. Each component uses a set of dedicated channels (service, monitoring or control).

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`

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_cms_gate_channel` : the conventional identifier of the *gate* channel used by th CMS server to accept/reject connection requests from the Vire server is stored as a character string.
   Example: `"snemo.cms.gate"`.

7. `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 to 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 set and store the following parameters as long as the connection with the Vire server is validated and maintained:

1. The *signature* of the connection with the Vire server. This is a randomly generated character string with a length which is fixed conventionaly[3].
   Example: `"3mAbGMuxQ47mbHUwTN5ZA8wKtZxD7c"`

2. The identifiers of some AMPQ channels dedicated to specific operations like: service operations, control operations, monitoring parameters. This could be three channels of respectively so-called types: *service*, *control* and *monitoring*. 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.

   NOTE: We don't know yet if additional channels should/could be used for very specific operations: logging, alarms... maybe the *service* channel is enough to host such messages.

Channel type	Channel name
vire.service	"snemo.amqp.viresvr.Djiecj1Fo2"
vire.control	"snemo.amqp.viresvr.IKA8b7SfK2"
vire.monitoring	"snemo.amqp.viresvr.n25VDOX7wG"

Table 1: Example of named channels with their specific types defined for communication between the Vire server and the CMS server.

[3] like the output of the Linux `pwgen` command: `pwgen -secure -numerals 30 -1`

```
# 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 sever

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

1. Establish a first 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 sever.
4. Recover a broken connection between the Vire server and the CMS server.
   NOTE (FM): it is not clear to me 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 *gate* channel should be used for these operations.

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

Pre-condition:

- The CMS server must not have an existing connection with the client Vire server (but see the note below).
- The Vire server must not have an existing connection with the target CMS server.

NOTE (FM): In principle there should be no problem for the CMS server to accept connection with several Vire servers simultaneously, provided that that they do not request concurrent access to resources (typically for datapoints and device methods with write/control access).
Description:

1. The Vire server sends a *connection request* with the following parameters:
   - The identifier of the setup (character string) which is expected to match the identifier stored in the CMS server (fetched from the main configuration file).
     Example: `setup_id="snemo"`
   - The version of the setup (character string) which is expected to match the version stored in the CMS server (fetched from the main configuration file).
     Example: `setup_version="1.0.2"`
   - 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.
     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

```
{
  "connection_request": {
    "setup_id"      : "snemo",
    "setup_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 *connection request* message emitted by the Vire server to the CMS server.

A typical JSON formatted *connection request* message is shown on Fig. 6.

NOTE: the exact format of the message will be discussed later.

Like this, 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 resources. It is possible to request less resources than the ones that are effectively available from the MOS servers. 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 (no 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 $\mathcal{V}_1$ containing only 250 target resources. If $\mathcal{V}_1 \subset \mathcal{R}$, 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 $\mathcal{V}_1$ does not belongs to set $\mathcal{R}$ ($\mathcal{V}_1 \subsetneq \mathcal{R}$), then the CMS server must rejects 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).

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

   (a) generate randomly an unique *signature* (character string) that identifies the communication session with the Vire server.

   (b) generate the identifiers of the *service*, *control* and *monitoring* MQ channels.

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

   (d) generate the list of current status associated to the requested resources.

   Then the CMS server transmits a response message to the Vire server, using the gate channel. The message contains the *signature*, the identifiers of MQ channels and the list of resources status. A typical JSON formatted *connection accept* message is shown on Fig. 7.

JSON

```
{
  "connection_accept": {
    "signature" : "3687DLKDQKJH3XHGDJQSD0873283",
    "amqp_channels"  : [
      {
        "type" : "vire.service",
        "name" : "sncms.amqp.viresvr.Djiecj1Fo2"
      },
      {
        "type" : "vire.control",
        "name" : "sncms.amqp.viresvr.IKA8b7SfK2"
      },
      {
        "type" : "vire.monitoring",
        "name" : "sncms.amqp.viresvr.n25VDOX7wG"
      }
    ],
    "resources_snapshot"  : [
      {
        "path"   : "SuperNEMO://Demonstrator/CMS/Coil/PS/Control/Current/__dp_read__",
        "timestamp" : "2016-02-03 17:52:03.031743801+01:00",
        "status" : "0000"
      },
      {
        "path"   : "SuperNEMO://Demonstrator/CMS/Coil/PS/Control/Voltage/__dp_read__",
        "timestamp" : "2016-02-03 17:52:03.031743801+01:00",
        "status" : "0000"
      },
      ...
      {
        "path"   : "SuperNEMO://Demonstrator/CMS/Acquisition/stop",
        "timestamp" : "2016-02-03 17:52:03.031743801+01:00",
        "status" : "0010"
      }
    ]
  }
```

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, the setup version or the list of requested resources is not valid, the CMS server must send back to the Vire server a message to reject the request. A typical JSON formatted *connection reject* message is shown on Fig. 8.

Post-condition:

When the connection is accepted and established:

- The Vire server stores the informations related to the connection as long as it lasts: *signature*, *channels' identifiers*, set of resources and associated informations...
- The Vire server cannot send a connection request to the same CMS server through the *gate* channel associated to this CMS server.
- Additional MQ channels are created and ready to transmit messages according to their types.
- The *gate* channel associated to the CMS server may be locked if the CMS server is allow to handle only one connection from an unique Vire server.

JSON

```
{
  "connection_reject" : {
    "error_type" : "code_what_error",
    "code_what_error" : {
      "code"    : "23",
      "message" : "Connection to CMS server was refused because of blah-blah-blah..."
    }
  }
}
```

Figure 8: Proposal for a JSON *connection rejection* message 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 CMS server must have an existing connection with a Vire server with associated signature and MQ channels set.

- No Vire client MQ channel requesting resources from the CMS should be running. NOTE (FM): need to rephrase this sentence, unclear

Description:

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

   A typical JSON formatted *disconnection request* message is shown on Fig. 9.

JSON

```
{
  "disconnection_request": {
    "signature" : "3687DLKDQKJH3XHGDJQSDO873283"
  }
}
```

Figure 9: Proposal for a JSON *disconnection request* message emitted by the Vire server to the CMS server.

2. The CMS server erases the stored *signature* and disposes of all resources associated to the connection (mapping tables...).

Post-condition:

- The session between the Vire server and the CMS server is terminated.

- The CMS server is ready to accept a new connection from a Vire server.

1.5.3 Reconnection with the Vire server

NOTE (FM): To be discussed

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[4] and the associated XML description of the device (for example:

[4] Interface Control Document

`${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*. Mapping the DAQ daemon object in the OPCUA space as the `SuperNEMO://Demonstrator/CMS/Acquisition/` path in Vire's resource space, two resources are thus defined 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__`
- `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.
Vire mounts the 192.168.1.15:4841:CMS.COIL_PS OPCUA device at the Vire SuperNEMO://Demonstrator/CMS/C 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).

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 set the missing flag for all associated resources.

  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.

- 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 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.

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.

Example: Assume the Vire server requests the execution of the `SuperNEMO://Demonstrator/CMS/Acquisition/start` resource at time t_0. The CMS server invoke 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 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 met an error. This can occur for example when the set point value for a datapoint is not valid.

  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.

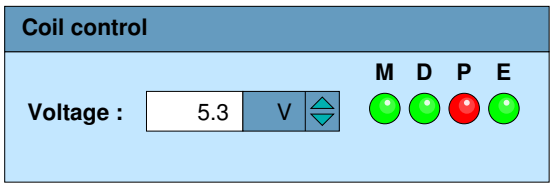

Figure 10: An example of a GUI component which displays the status bits associated to a resource, informing the user of the realtime conditions. Here the **M**issing flag is not set, reflecting that the resource is available to the system typically because the associated device is recognized by the MOS 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.

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:

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

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

JSON

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

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

JSON

```
{
  "resource_exec": {
    "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

```
{
  "resource_exec": {
    "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.

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.

The `resource_exec_response` message is delivered back to the Vire server by the CMS server.

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

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

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.

JSON

```
{
  "resource_exec_response": {
    "error_type" : "null",
    "path" : "SuperNEMO://Demonstrator/CMS/Acquisition/start",
    "status" : "0000",
    "reception_timestamp" : "2016-02-03 17:52:03.031743801+01:00",
    "completion_timestamp" : "2016-02-03 17:52:17.56143801+01:00",
    "output_args" : [
    ]
  }
}
```

Figure 14: Proposal for a JSON *resource execution 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.

JSON

```
{
  "resource_exec_response": {
    "error_type" : "null",
    "path" : "SuperNEMO://Demonstrator/CMS/Coil/PS/Control/Current/__dp_read__",
    "status" : "0000",
    "reception_timestamp" : "2016-02-03 17:52:03.031743801+01:00",
    "completion_timestamp" : "2016-02-03 17:52:17.56143801+01:00",
    "output_args" : [
      "value" : "1.23"
    ]
  }
}
```

Figure 15:

JSON

```
{
  "resource_exec_response": {
    "error_type" : "what_error",
    "what_error" : {
      "message" : "Operation failed because the DAQ device is already running."
    }
    "path" : "SuperNEMO://Demonstrator/CMS/Acquisition/start",
    "status" : "0100",
    "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.

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).

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.

3 Access to the CMS 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[5].

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

[5] `"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.

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.

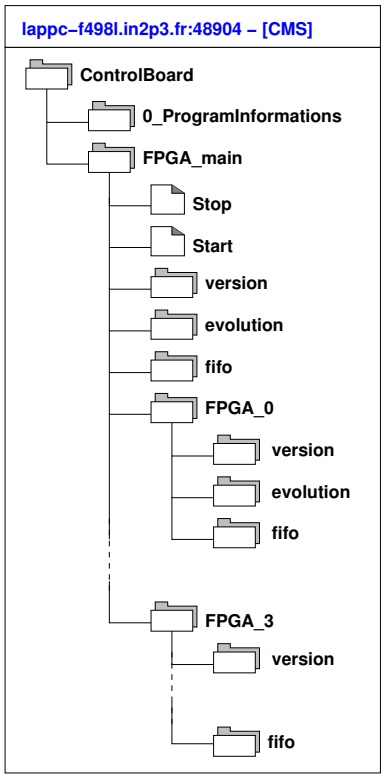

Figure 19: Example of a device managed through a MOS server. The root device is named `ControlBoard`. First level daughter devices are `0_ProgramInformations` and `FPGA_main`. Here the MOS/OPCUA server is labelled `CMS`.

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.

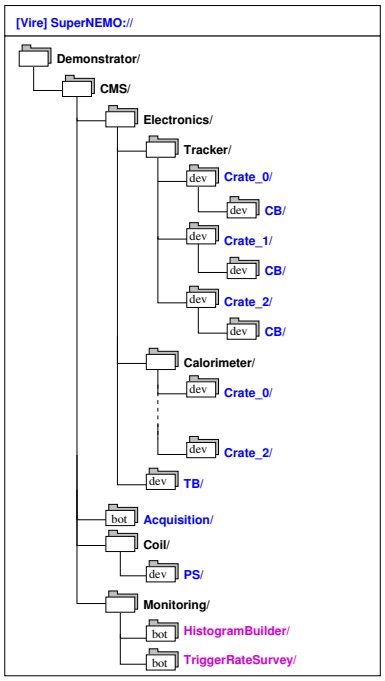

Figure 20: Example of a hierarchy of devices described by the Vire API. The root device is named `SuperNEMO:`. The top level (root) device is named `Demonstrator`. The devices colored in blue are managed through MOS/OPCUA. The devices colored in magenta are directly embedded in the Vire server. Devices 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.

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.

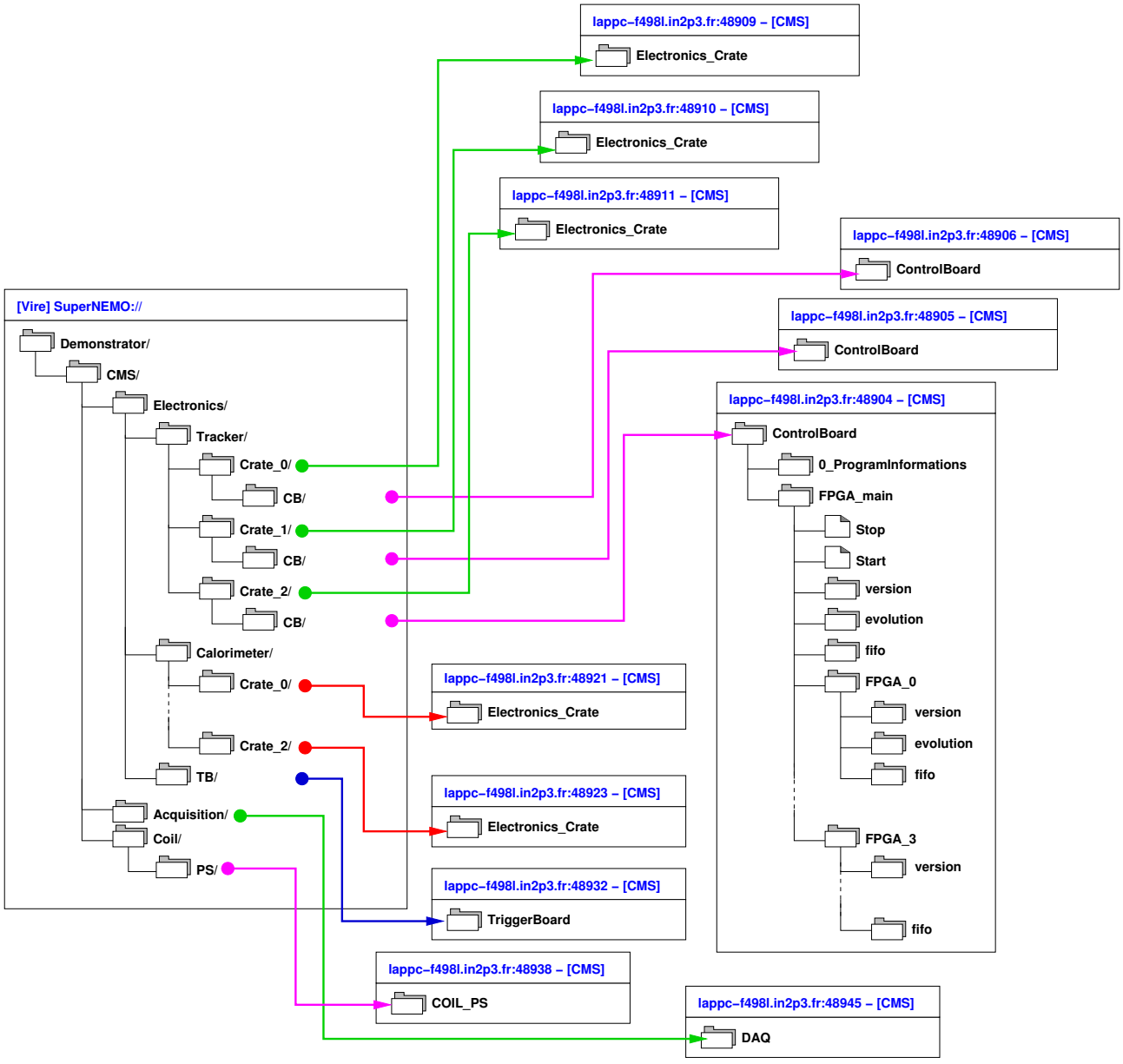

Figure 21: The mounting of many MOS device hierarchies in the Vire device hierarchy. Each OPCUA server runs a simple hardware device that is *mounted* from a specific node with its own path.

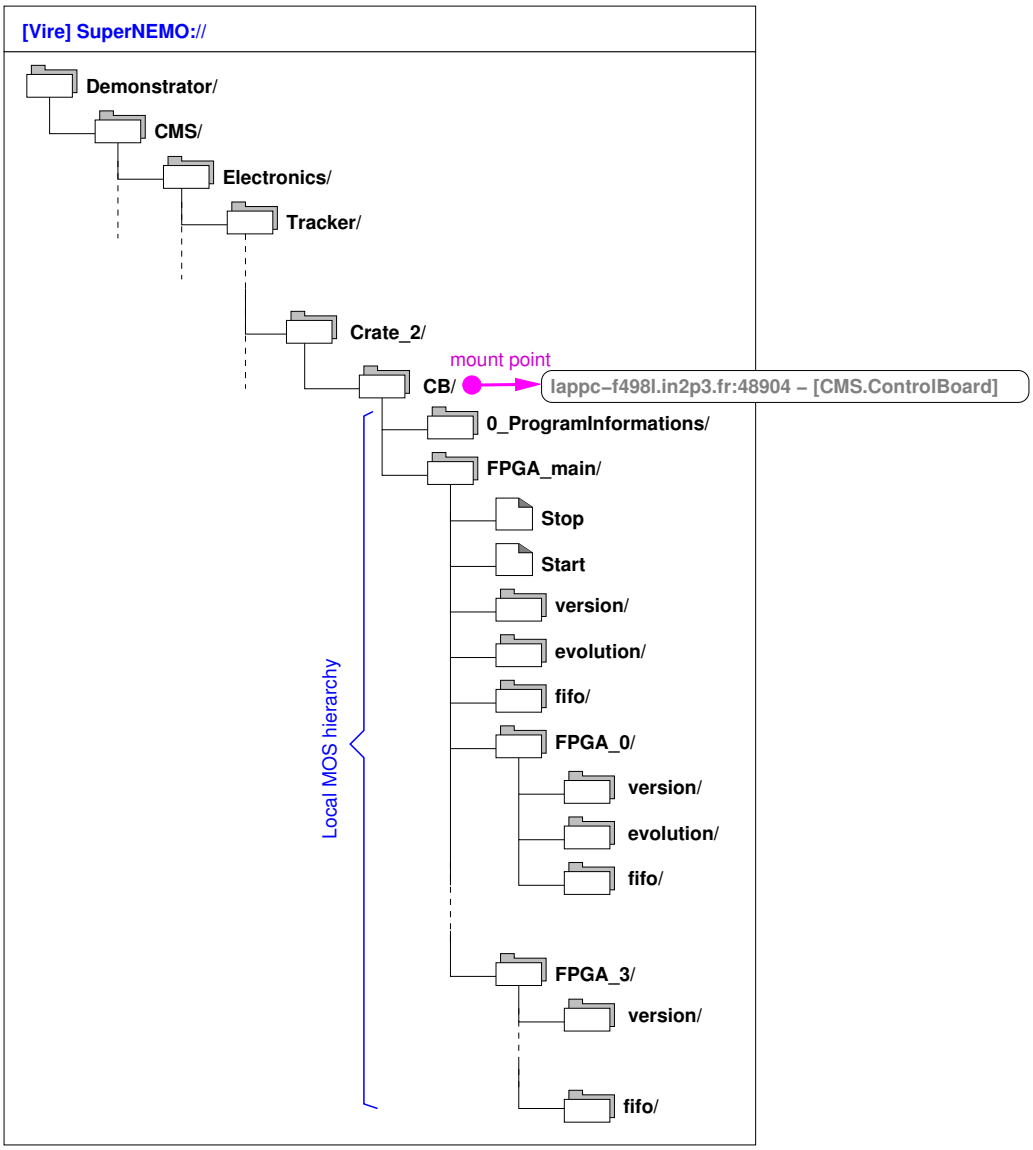

Figure 22: The mounting of one MOS device and its local hierarchy in the Vire device hierarchy.

C JSON formatting

C.1 Representation of error

Typically, an error is represented through a record with a given *error_ type*. Any response message must inform the receiver about the success/failure status of the operation. The message must contain an *error_ type* field. Depending on the value of this field, additional informations can be appended.

Four basic error types are defined:

- `"null"` (or `"none"`) : this corresponds to a successful operation with no error. No additional information is requested.
- `"code_error"` : this corresponds to an error documented through a single error code (a non null integer).

  A `"code_error"` record is thus expected:

JSON

```
{
  "error_type" : "code_error",
  "code_error" : {
    "code" : "12"
  }
}
```

- `"what_error"` : this corresponds to an error documented through a single error message (a non empty string).

  A `"what_error"` record is thus expected:

JSON

```
{
  "error_type" : "what_error",
  "what_error" : {
    "message" : "Device 'dev00234' is disconnected"
  }
}
```

- `"code_what_error"` : this corresponds to an error documented through an error code and an error message.

  A `"code_what_error"` record is thus expected:

JSON

```
{
  "error_type" : "code_what_error",
  "code_what_error" : {
    "code"    : "12"
    "message" : "Device 'dev00234' is disconnected"
  }
}
```