The Vire-CMS to MOS interface
version 0.5

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

June 9, 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).

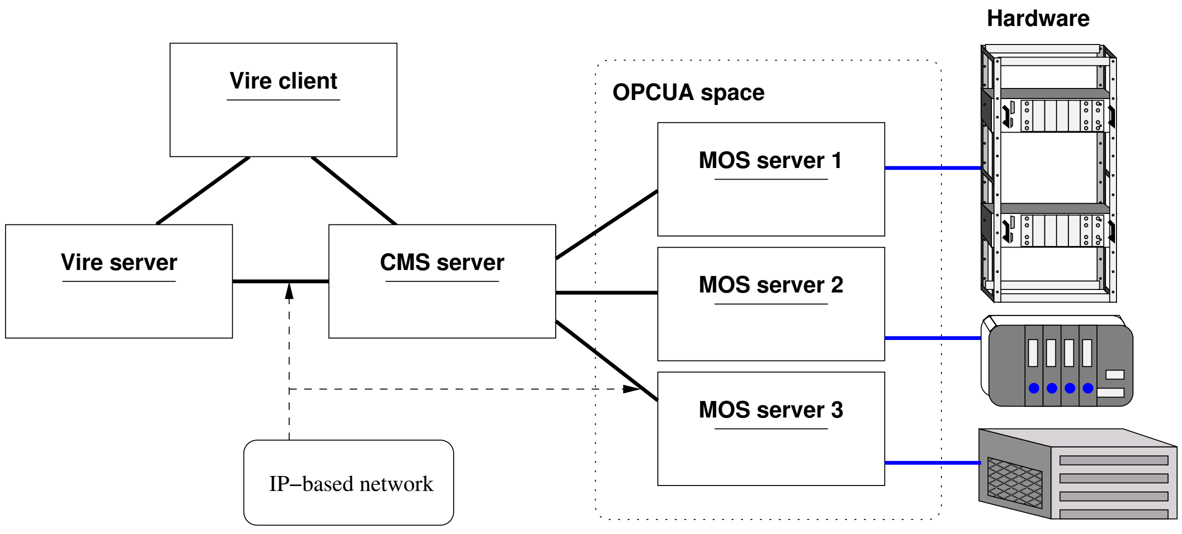


Figure 1: Overview of the Vire server, a Vire client, the CMS server and its connections to the hardware through dedicated MOS servers.

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_name` 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 short 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 Vire 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 strings are the identifiers of additional channels 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 `mos_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.cZqblZoOmY"
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 operations 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 (to be discussed). This channel uses a *synchronous* mode.

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

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

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

5. the *pubsub* channel (*asynchronous* mode) 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 service.

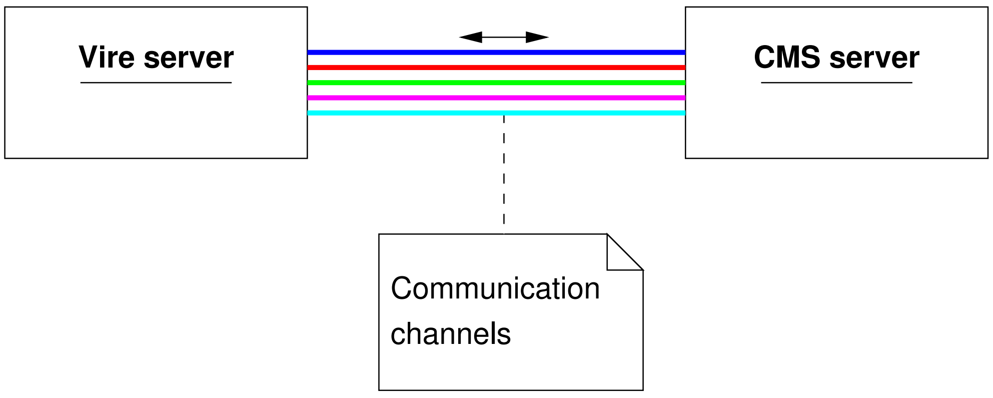


Figure 3: The communication channels between the CMS server and the Vire server.

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

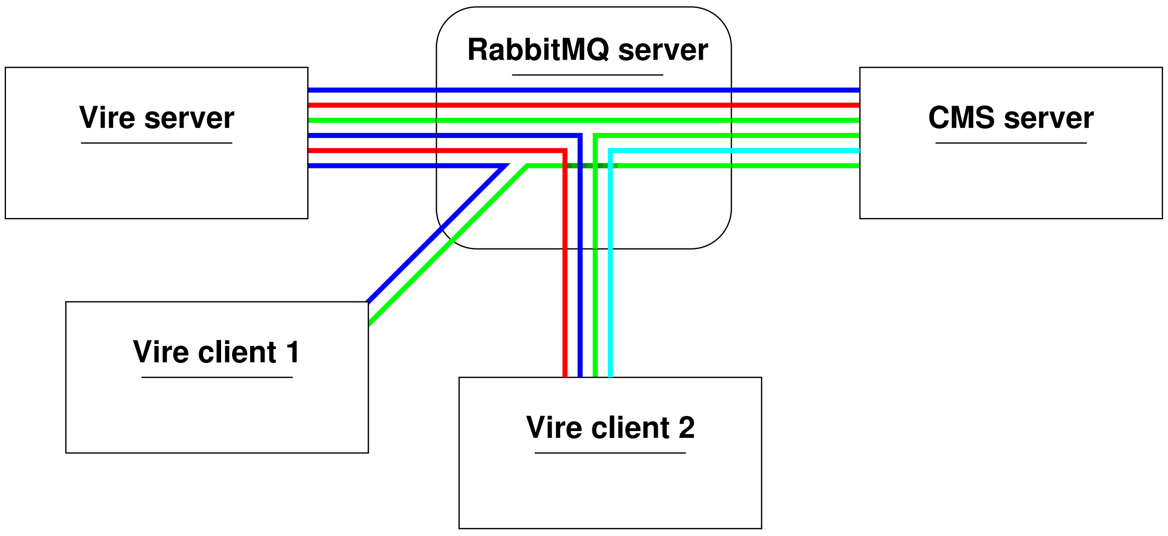


Figure 4: A RabbitMQ based communication framework shared by the CMS server, the Vire server and some Vire clients. Each component uses a set of dedicated channels (service, monitoring, control, event, pubsub).

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 non ambiguous 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_name`:

   The name of the setup is extracted from the main setup configuration file. It is composed of a character string that identifies the name of the experimental setup.
   Example: `"snemo"`.

3. `setup_version`:

   The version of the setup is extracted from the main setup configuration file. It is a character string which uses a standard version numbering scheme (`"1.0"`) or an arbitrary simple character string tag.
   Example: `"1.0.2"` or `"test_42"`

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.cZqblZoOmY"`.

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.5Hja55450z"`.

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*. Table 1 shows a possible set of such channels, each associated to a given channel type. Only one channel of one given type can exists at the same time within the context of a given Vire to CMS connection. The main configuration file defines conventionally the identifiers of these channels.

   Each channel can be dynamically enriched with some specific bindings dedicated to specific operations, services, clients... (to be discussed).

```
# 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.

Channel type	Sync/Async	Channel name
`vire::amqp::service`	sync	`"snemo.amqp.viresvr.cZqblZoOmY"`
`vire::amqp::control`	sync	`"snemo.amqp.viresvr.IKA8b7SfK2"`
`vire::amqp::monitoring`	sync	`"snemo.amqp.viresvr.n25VDOX7wG"`
`vire::amqp::event`	async	`"snemo.amqp.viresvr.5Hja5545Oz"`
`vire::amqp::pubsub`	async	`"snemo.amqp.viresvr.WAqq7ERzsl"`

Table 1: Conventional channels with their specific types defined for communication between the Vire server, the CMS server and Vire clients. The channels' names shown here are arbitrary.

.

1.5 Service 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* object to the CMS server through the service channel (object type id: `"vire::mosinterface::connection_request"`). This object has the following parameters:

   - `setup_id` : the identifier of the setup consists in two records (Fig. 6):
     - `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"`

     It is implemented in the Vire API as an object of type `vire::utility::instance_identifier`. A sample JSON formated stream is shown on figure 7.

C/C++
```
struct instance_identifier
{
  // Attributes:
  std::string name;    // The name of the setup
  std::string version; // The version of the setup (optional)
};
```

Figure 6: Structure of the `vire::utility::instance_identifier` object type.

The setup identifier is expected to match the identifier stored in the CMS server (fetched from the main configuration file).

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

Figure 7: JSON formated `setup_identifier` object.

- 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"
  ```

Figure 8 shows the structure of *connection request* object.

```
C/C++
struct connection_request
{
  // Attributes:
  instance_identifier      setup_id;            // The experimental setup identifier.
  std::vector<std::string> requested_resources; // The list of resources requested by
                                                // the Vire server.
};
```

Figure 8: The structure of a *connection request* object emitted by the Vire server to the CMS server.

A typical JSON formatted *connection request* object is shown on Fig. 9. The exact format of the encapsulating message structure is discussed in appendix C. The detailed description of the *connection request* object is documented in appendix D.

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 (after the `devices_launch.conf` startup file). In that case, the Vire/CMS connection is established in *failsoft* mode, for example for debugging purpose.

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 its path and hosting device) by the CMS server, but effectively missing because of some MOS device connection

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 9: JSON formatted `vire::mosinterface::connection_request` object emitted by the Vire server to the CMS server.

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.

   The CMS server then transmits a *connection request success response* object (object type id: `"vire::mosinterface::connection_success_response"`) object to the Vire server, using the service channel. The message contains the list of resource status records (class `vire::resource::resource_status_record`, Fig. 10).

   The structure of the *connection success response* object is shown on Fig. 11. It consists in the exhaustive list of status records associated to all requested resources.

   A typical JSON format of a *connection success response* message is shown on Fig. **??**.

C/C++
```
struct resource_status_record
{
  // Attributes:
  std::string path;   // Resource path
  uint16_t    status; // Resource dynamic status bitset
};
```

Figure 10: The structure of a *resource status record* object.

C/C++

```
struct connection_success_response
{
  // Attributes:
  std::vector<resource_status_record> resources_snapshot;
     // The status snapshot of all resources requested by the Vire server
};
```

Figure 11: The structure of a *connection success response* object emitted by the CMS server to the Vire server.

JSON

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

Figure 12: A JSON formatted *connection success response* object emitted by the CMS server to the Vire server. The meaning of the `status` associated to each resource is 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 *connection failure response* object is built (object identifier: `"vire::mosinterface::connection_failure_response"`, Fig. 13) and encoded in a JSON formatted message (Fig. 14). The failure response message is able to encode three types of errors:

- `invalid_context_error` (error type id : `"invalid_context_error"`) : if the context is not suitable to establish a connection with the Vire server (maintenance shutdown of critical MOS, server already connected...).
- `invalid_setup_id_error` (error type id : `"invalid_setup_id_error"`) : if the requested setup identifier does not match the setup addressed by the CMS server.
- `unknown_resources_error` (error type id : `"unknown_resources_error"`): 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 (see details in appendix D). Specific additional attributes may be available depending of the type of the error.

C/C++

```cpp
struct connection_failure_response
{
  struct invalid_context_error;   // Nested error class
  struct invalid_setup_id_error;  // Nested error class
  struct unknown_resources_error; // Nested error class
  // Attributes:
  error_identifier  error_type_id; // Error type identifier
  XXX_error         error;         // An instance of one of the nested error types above
}
```

Figure 13: Structure of a *connection failure response* object emitted by the CMS server to the Vire server.

JSON

```json
{
  "error_type_id" : {
    "name"    : "unknown_resources_error",
    "version" : "1.0"
  }
  "error" : {
    "code" : "100",
    "message" : "Connection to CMS server was refused because of unrecognized resource(s)",
    "unknown_paths" : [
      "SuperNEMO://Demonstrator/CMS/NotExistingDevice/start",
      "SuperNEMO://Demonstrator/CMS/NotExistingDevice/stop"
    ]
  }
}
```

Figure 14: Example of a JSON formatted *connection failure response* object sent by the CMS server to the Vire server.

Post-conditions:

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 (dynamic 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 computed a map of resources that permits to identify each requested resource and associate it to an item published by a given MOS: datapoint (DP), method.

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

1.5.2 Disconnect of the Vire server from the CMS server

Pre-condition:

- The Vire server must have an existing connection with the CMS server.
- The CMS server must have an existing connection with a Vire server.
- All current synchronous resource executions in write mode should be terminated. It is the responsibility of the Vire server to guarantee such condition. Typically a synchronous resource execution cannot be submitted to the CMS server from a Vire *session* if there is not enough time left to guarantee the completion of the resource execution (including an associated timeout).

Description:

1. The Vire server constructs a *disconnection request* object (object type ID: `"vire::mosinterface::disconnection_request"`) and sends it to the CMS server through the service channel.

   The structure of a *disconnection request* object is shown on Fig. 15.

C/C++

```
struct disconnection_request
{
};
```

Figure 15: Structure of the *disconnection request* object emitted by the Vire server to the CMS server.

A typical JSON formatted *disconnection request* object is shown on Fig. 16. The exact format of the message will be discussed in appendix C.

JSON

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

Figure 16: JSON formatted *disconnection request* object emitted by the Vire server to the CMS server.

2. The CMS server acknowledges the disconnection through a *disconnection request success response* object (object type ID : `"vire::mosinterface::disconnection_success_response"`, Fig. 17). It disposes of the Vire resource map associated to this connection.

C/C++

```
struct disconnection_success_response
{
};
```

Figure 17: Structure of the *disconnection success response* object emitted by the CMS server to the Vire server.

3. The CMS server rejects the disconnection through a *disconnection failure response* object (object type ID : `"vire::mosinterface::disconnection_failure_response"`, Fig. 18).

C/C++
```
struct disconnection_failure_response
{
  // Nested error types:
  struct invalid_context_error;

  // Attributes:
  type_identifier error_type_id; // Error type identifier
  XXX_error       error;         // Encapsulated error of one type above
};
```

Figure 18: Structure of the *disconnection failure response* object emitted by the CMS server to the Vire server.

Post-conditions:

- The connection between the Vire server and the CMS server is terminated.
- The Vire server marks all resources addressed through the CMS server as *missing*.
- 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 CMS server.

1.5.3 Restore a broken connection of the Vire server with the CMS server

NOTE (FM): To be discussed

2 Representation of Vire resources within 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, `DAQstart`, `DAQstop` ...).

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. It may be the cached last value recorded in a MOS server or the actual current value stored in the device itself.

- 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. Such resource generally implies the user has specific privilege.

- 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.

Depending of the size and compounding of the experimental apparatus, the Vire server may publish 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 and accessible 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:

[3]Interface Control Document

- 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 conventionally defined in the Vire API. 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 (get) and write (set) 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 associates any resource path to an 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. This is guaranted by the XML to Vire configuration files convertion utility. 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` datapoint monitoring functionality.

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 and which operations are possible. 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. Four *resource status flags* are defined:

- 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 associated to this deviced. It is the responsibility of the CMS server to detect such event and update all associated resources' missing flags.

  Example: Assume the MOS running at `192.168.1.15:4841` and hosting the `CMS.COIL_PS` device is disconnected because Naughty Zoot teared off the cable connecting the coil power supply device from the computer hosting the MOS server. Then all resources identified with a path starting with `"SuperNEMO://Demonstrator/CMS/Coil/PS/"` will be automatically tagged as *missing* by the CMS server.

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 significant 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+output parameters/failure+error record) 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 *failed* flag is set when a the device associated to resource or a single resource itself met some internal error/failure state.

  Example: none provided.

Given these four boolean flags, it is possible for the CMS server to build a 4-bits status word that reflects the current status of the resource at any time. Accessing this status bitset for each resource, any client of the CMS server (the Vire server application or some Vire client applications) can figure out the real time conditions to access resources and build a local image of the resources.

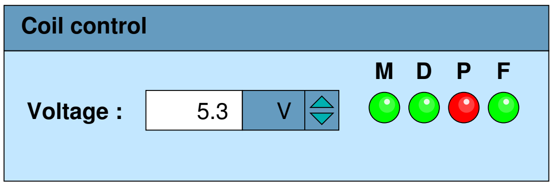


Figure 19: An example of a GUI component which displays the status bits associated to a resource, informing the user of the realtime conditions. Here the **M**issing flag is not set, reflecting that the resource is available to the system typically because the associated device is recognized by the MOS and is currently running with all its functionnalities. The **D**isabled 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 **F**ailed flag is not set, reflecting that no specific problem has been detected by the CMS server with this datapoint 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 (or client) is expected to request the execution of a resource with a specific `vire::resource::resource_exec_request` object.

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

Question: support for timeout.

Typical JSON formatted *resource execution request* objects are shown on figures 20, 21 and 22.

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

Figure 20: Typical JSON formatted *resource execution request* object sent by the Vire server to the CMS server. Here the resource corresponds to a method that takes no input arguments.

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

Figure 21: Typical JSON formatted *resource execution request* object sent by the Vire server to the CMS server. Here the resource corresponds to a write method that takes only one input argument.

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

Figure 22: Typical JSON formatted *resource execution request* object sent by the Vire server to the CMS server. Here the resource corresponds to a read method that takes no input argument.

The CMS server must decode a message which wraps the request object 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.

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

A typical JSON formatted *resource_ exec_ success_ response* object is shown on Fig. ?? and ??.

JSON
```
{
  "status" : {
    "path" : "SuperNEMO://Demonstrator/CMS/Acquisition/start",
    "flags" : "0100"
  },
  "reception_timestamp" : "20160203T175203.031743801", // TO BE CHECKED
  "completion_timestamp" : "20160203T175217.56143801", // TO BE CHECKED
  "output_arguments" : []
}
```

Figure 23: A JSON formatted *resource execution success response* object sent by the CMS server to the Vire server. Here the response does not contain any output arguments. Note that the CMS server also informs the Vire server about the updated resource status.

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

Figure 24: JSON formatted `resource_exec_success_response` object.

In case of failure, the CMS server builds a failure response object which describes the error (figure 25).

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

- `invalid_context_error` : the general context does not allow to operate the execution of a resource.
- `invalid_resource_error` : the resource path is not valid.
- `invalid_status_error` : the resource status is bad.
- `resource_exec_error` : the execution of the resource met an error: invalid context, invalid input argument name, invalid input argument value, timeout...

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

JSON

```
{
   "error_type_id" : {
      "name" : "invalid_status_error",
      "version" : "1.0"
   },
   "error" : {
      "code" : "101"
      "message" : "Operation failed because the DAQ device is already running."
      "path" : "SuperNEMO://Demonstrator/CMS/DAQ/start",
      "flag" : "disabled"
    }
  }
}
```

Figure 25: Example of JSON formatted *resource execution failure response* object 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.

within the timeout, an error response message is sent back to the Vire server.

Execute a resource in *non blocking* mode: A resource can be executed in *non blocking* mode through the specific `resource_exec_non_blocking_request` object. The content of this object is the same than in *blocking* mode.

At reception of the request object, the CMS server immediately sents back a response object of one of the following types:

- `resource_exec_non_blocking_success_response` object:

```
JSON
{
  "status" : {
    "path" : "SuperNEMO://Demonstrator/CMS/Coil/PS/Control/Current/__dp_read__",
    "flags" : "0010",
  }
}
```

Figure 26: JSON formatted `resource_exec_non_blocking_success_response` object.

- `resource_exec_non_blocking_failure_response` object:

  TO BE DONE.

The pending flag is set until the operation has been completed or failed.

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. 27 and 28 (for the response message).

struct

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

Figure 27: JSON formatted *resource fetch status* object sent by the Vire server to the CMS server.

```
JSON
{
  "status" : {
    "path" : "SuperNEMO://Demonstrator/CMS/Coil/PS/Control/Current/__dp_read__",
    "flags" : "0010"
  }
}
```

Figure 28: JSON formatted *resource fetch status* success response message sent back by the CMS server to the Vire server.

Activate pub/sub of a resource: The Vire server is able to request the dynamic activation/deactivation of the Pub/Sub service associated to a given resource. Three payload objects are thus defined to address this kind of transaction between the Vire and CS servers:

- `vire::resource::resource_pubsub_request` wraps a Vire server request to activate/deactivate the Pub/Sub service for a given resource managed through the CMS server.
- `vire::resource::resource_pubsub_success_response` wraps a successful response of the CMS server.
- `vire::resource::resource_pubsub_failure_response` wraps a failure response of the CMS server.

Specific utility objects are also defined:

- `vire::resource::amqp_pubsub_access` : describes the access to the Pub/Sub service for a given resource.

These objects are described in appendix D.

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 29 shows an example of a device embedded in a MOS server.

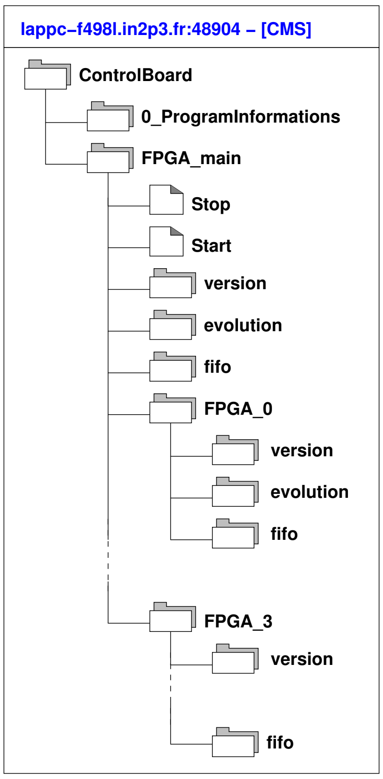


Figure 29: Example of a device managed through a MOS server. The root device is named `ControlBoard`. First level daughter devices are `0_ProgramInformations` and `FPGA_main`. Here the MOS/OPCUA server is labelled `CMS`.

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 30. 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.

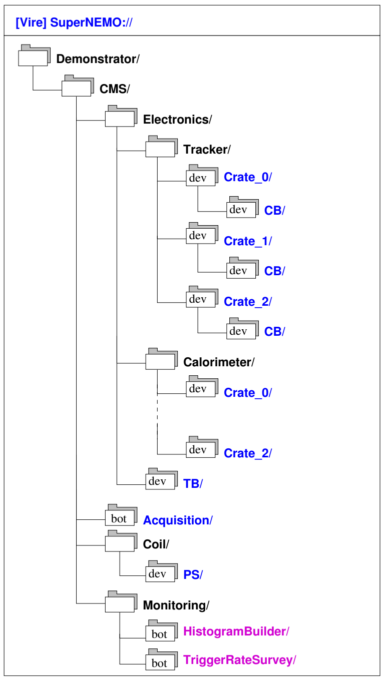


Figure 30: Example of a hierarchy of devices described by the Vire API. The root device is named `SuperNEMO:`. The top level (root) device is named `Demonstrator`. The devices colored in blue are managed through MOS/OPCUA. The devices colored in magenta are directly embedded in the Vire server. Devices 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 31 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.

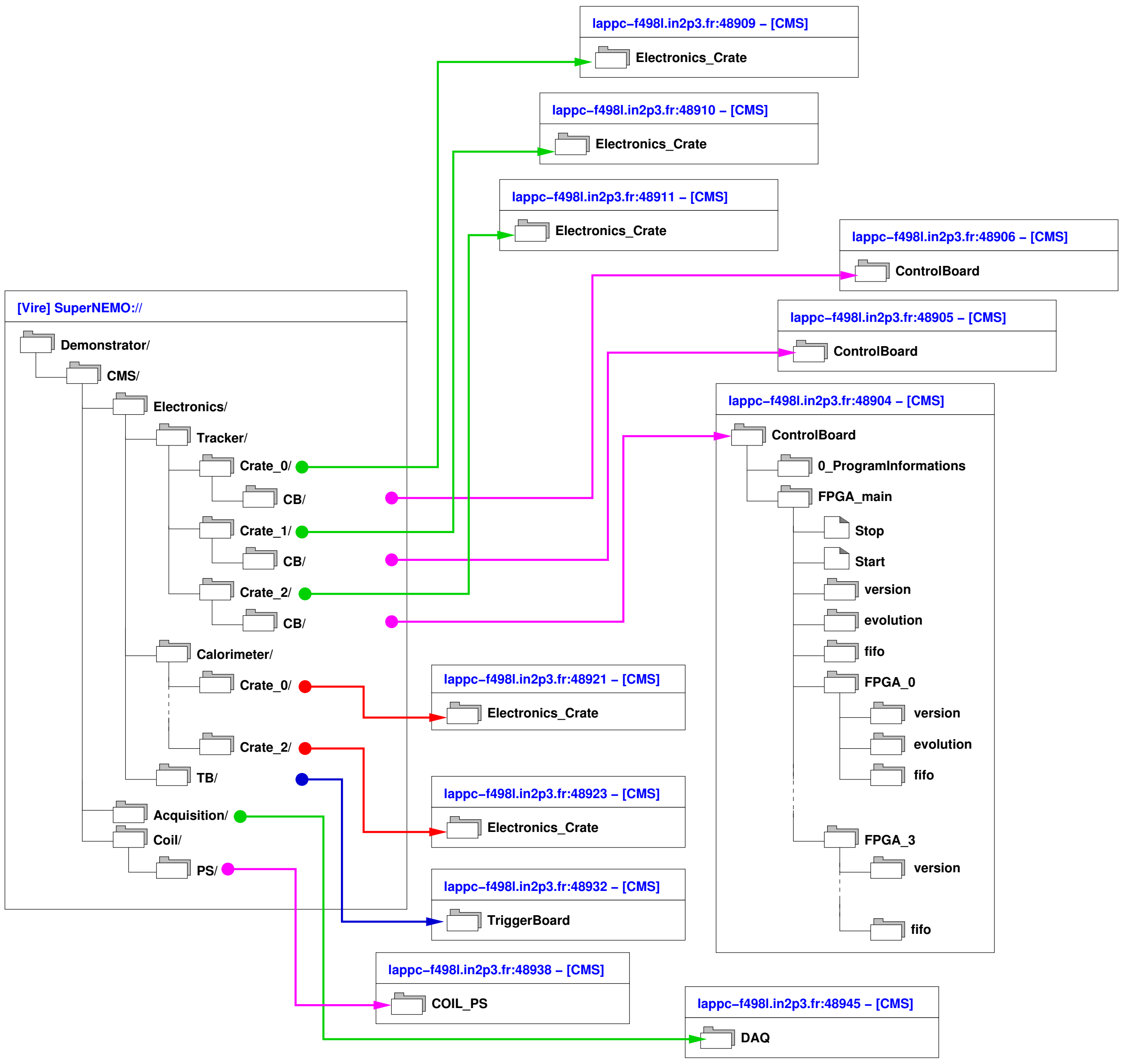


Figure 31: The mounting of many MOS device hierarchies in the Vire device hierarchy. Each OPCUA server runs a simple hardware device that is *mounted* from a specific node with its own path.

B.3 Example

Using the examples displayed in figure 31, we consider in detail the way one specific device managed by MOS is mounted in the Vire hierarchy. Figure 32 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−f4981.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.

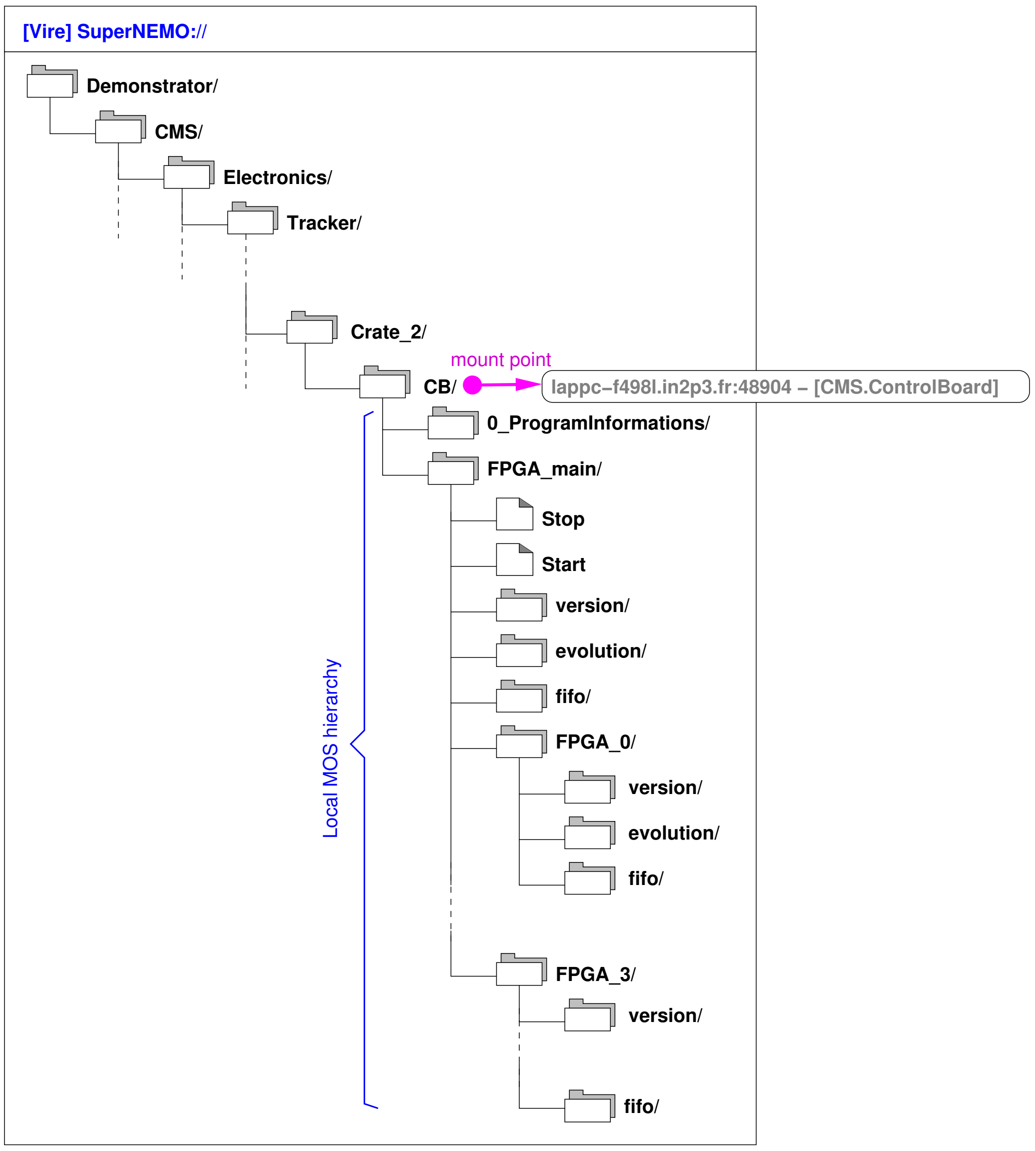


Figure 32: The mounting of one MOS device and its local hierarchy in the Vire device hierarchy.

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 33):

- 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 {
  msg_header header; // Header of the message
  msg_body   body;   // Body of the message
};
```

Figure 33: The structure of a message object (C++).

C.2 The message header

The header contains (figure 34):

- 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 35).
- 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 36).
- 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 msg_header {
  message_identifier message_id;  // Message identifier from the emitter.
  format_identifier  format_id;   // Format identifier.
  std::string        timestamp;   // Timestamp.
  message_identifier in_reply_to; // Message identifier of the associated request message.
};
```

Figure 34: The structure of a message header object (C++ class: `vire::message::msg_header`).

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 35: 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 36: The structure of a message format identifier (C++).

JSON

```
  {
    "header" : {
      "message_id" : {
        "emitter" : "vire.server",
        "number"  : "723"
      },
      "format_id"  : {
        "name" : "vire::message::format::basic",
        "version" : "1.0"
      },
      "timestamp" : "20160516T085346",
      "in_reply_to" : {
        "emitter" : "vire.client",
        "number"  : "843"
      }
    },
    "body" : {
      ... special encoding for the data ...
    }
  }
```

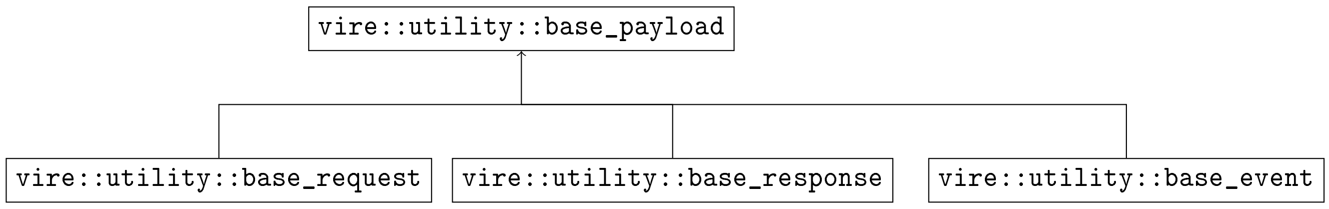
Figure 37: Example of the header of a JSON formatted message.

C.3 The message body

The default message format in Vire is named "vire::message::format::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 a *payload object* that has to be transmitted between two components of the system. *Payload objects* are classified in one of three categories:

1. Request : describes a request submitted by one component to another component (generally during a synchronous exchange).
2. Response: describes the response to a former request (during a synchronous exchange).
3. Event: describes a arbitrary information record (alarm, exception, signal, asynchronous).

Vire implements the following class hierarchy:


vire::utility::base_payload
vire::utility::base_request
vire::utility::base_response
vire::utility::base_event

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 38):

1. The `object_type_id` specifies the type of the encoded object (figure 39). 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.

C++
```
struct msg_body {
  type_identifier      object_type_id; // Object type identifier.
  const base_payload * object;         // Handle to a payload object.
};
```

Figure 38: The structure of a message body object (C++).

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

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.

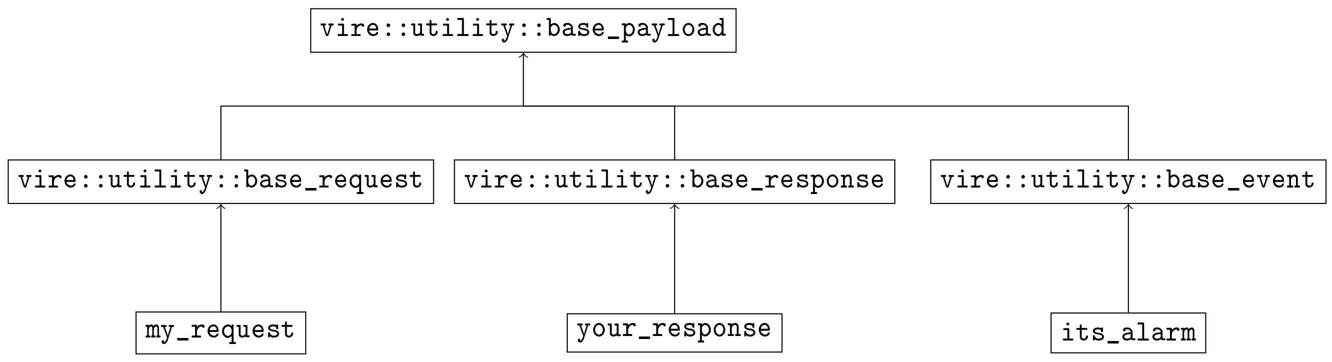
Figure 40 shows the formatted body of a message which encodes a connection request object from the Vire server to the CMS server.

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

Figure 40: Example of the body of a JSON formatted message wrapping a payload object of type `vire::cmsserver::connection_request`, inherited from the `vire::utility::base_request` type.

D Vire payload objects

As mentioned in appendix C, Vire messages are wrappers for *payload objects*. Each type of payload object must be (de)serializable through JSON. The following class hierarchy shows the base architecture used to define new payload objects.


vire::utility::base_payload
vire::utility::base_request
vire::utility::base_response
vire::utility::base_event
my_request
your_response
its_alarm

D.1 Utilities

Any payload object (request, response or event) generally contains some information records which are specific to the functionalities of the payload object they belong. These records are of arbitrary types. Of course they can be (de)serialized using JSON. Some of these types are general enough to be defined by the Vire core API itself because they are reused by various payload objects. These types are considered as *utilities*. Among them we find: error record types, identifier types, resource status records...We propose to describe them in this section. Other records are specific to their parent payload objects. Such objects will be described with the payload object they belong to.

D.1.1 Errors

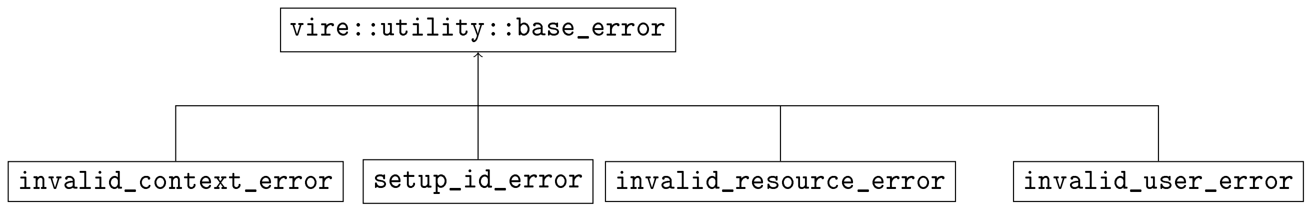
Some *response* or *event* payload objects contains a specific error record object. A *failure response* or an *exception event* object will always contain such an error record object.

Each *error record* is represented by an instance of a given error type. Each of the error types defined in Vire inherits the `vire::utility::base_error` base class (figure 41) which contains the following attributes:

- the error code: A non zero integer which is set to 1 by default (indicating a generic failure case). The error code can be conventionally set to any positive integer value to represent a specific error case, depending on the context.
- the error message: an optional human readable character string which documents the issue.

An example of JSON formatted basic error object is given in figure 42.

Several type of generic errors are defined in Vire:



C++

```
struct base_error
{
  // Attributes:
  int         code;    // Error code (>0).
  std::string message; // Error message (optional).
};
```

Figure 41: The structure of a `vire::utility::base_error` object (C++).

JSON

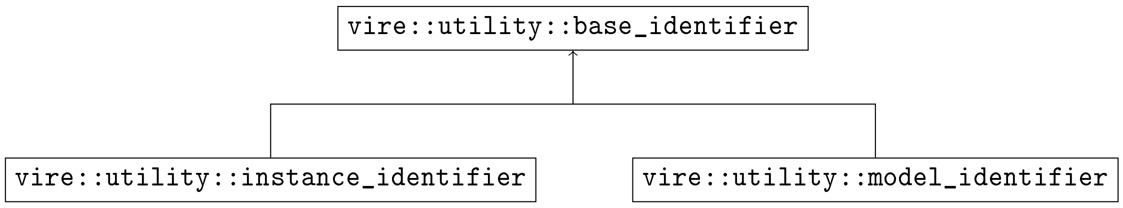
```
{
  "code" : "42",
  "message" : "Invalid AMQP server port=[2341]"
}
```

Figure 42: JSON formatted basic error object (class `vire::utility::base_error`.

- `vire::utility::invalid_context_error`
- `vire::utility::setup_id_error`
- `vire::resource::invalid_resource_error`
- `vire::user::invalid_user_error`

D.1.2 Object and type identifiers

Vire uses some dedicated classes to represent the identifier of various objects or instance as well as various types (or models) of components. Vire implements the following class hierarchy:


vire::utility::base_identifier
vire::utility::instance_identifier
vire::utility::model_identifier

The `vire::utility::base_identifier` (figure 43) class is a pure abstract class that cannot be instantiated. However it contains a mandatory name and an optional version description which are used by all inherited classes:

- The `vire::utility::instance_identifier` concrete class inherits `vire::utility::base_identifier` and is used to identify unique instances of objects known by the system.
- The `vire::utility::model_identifier` concrete class also inherits `vire::utility::base_identifier` and is used to identify types of objects registered in the system.

The only difference between these two classes is the validation scheme of the name attribute. Figure 44 shows an example of instance indentifier.

C++

```
struct base_identifier
{
  // Attributes:
  std::string name;    // The name uniquely identifying the object or the type of object.
  std::string version; // An optional character string representing the version
                       // of the object type.
};
```

Figure 43: The structure of the `vire::utility::base_identifier` class (C++).

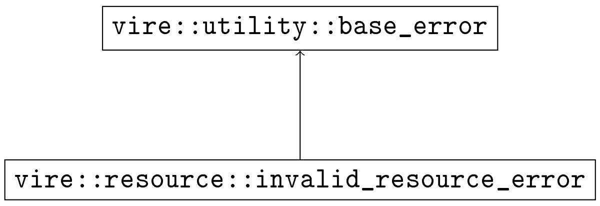
JSON

```
{
  "name" : "vire::resource::invalid_resource_error",
  "version" : "1.0"
}
```

Figure 44: JSON formatted class identifier object (class `vire::utility::model_identifier`). Here one identifies a specific error type.

D.1.3 Resource related objects

- Class `vire::resource::invalid_resource_error` (figure 45).


vire::utility::base_error
vire::resource::invalid_resource_error

C++

```
struct invalid_resource_error : public vire::utility::base_error
{
  // Attributes:
  int         code;         // Error code
  std::string message;      // Error message
  std::string invalid_path; // Invalid resource path
};
```

Figure 45: The structure of a invalid resource error object (C++).

JSON++

```
{
  "code" : "3",
  "message" : "Resource path 'Atlas://Calorimeter/HV/Crate1/stop' is invalid",
  "invalid_path" : "Atlas://Calorimeter/HV/Crate1/stop"
}
```

Figure 46: JSON formatted invalid resource error object.

- Class `vire::resource::resource_status_record` (figure 47).

C++

```
struct resource_status_record
{
  // Attributes:
  std::string path;  // Path of the requested resource
  uint16_t    flags; // Status bits (Missing/Disabled/Pending/Error)
};
```

Figure 47: The structure of a resource status record object (C++).

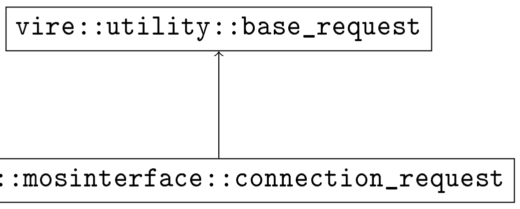
JSON++

```
{
  "path" : "SuperNEMO://Demonstrator/CMS/Coil/Control/Current/__dp_read__",
  "flags" : "0010"
}
```

Figure 48: JSON formatted resource status record object.

D.2 Connection of the Vire server to the CMS server

- The `vire::mosinterface::connection_request` class (version `1.0`) represents a connection request sent by the Vire server to the CMS server through the service channel.


vire::utility::base_request
vire::mosinterface::connection_request

  Class registration:

  - name: "`vire::mosinterface::connection_request`"
  - version: "1.0"

C++

```
struct connection_request : public vire::utility::base_request
{
  // Type alias:
  typedef vire::utility::instance_identifier setup_identifier;

  // Attributes:
  setup_identifier         setup_id;            // Identifier of the experimental setup
  std::vector<std::string> requested_resources; // The list of requested resources
                                                // addressed by path
};
```

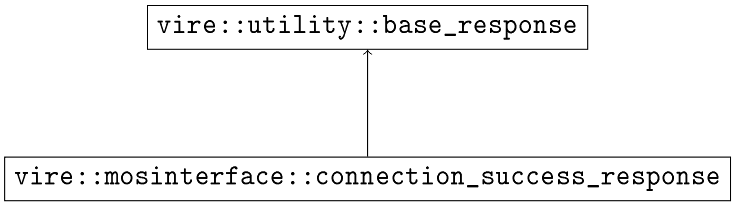
Figure 49: The structure of the connection request object to be emitted by the Vire server to the CMS server (C++).

C++

```
{
  "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/Acquisition/start",
    "SuperNEMO://Demonstrator/CMS/Acquisition/stop"
  ]
}
```

Figure 50: A JSON formatted connection request object sent by the Vire server to the CMS server (C++).

- The `vire::mosinterface::connection_success_response` class represents the response sent back to the Vire server by the CMS server through the service channel in case of success.


vire::utility::base_response
vire::mosinterface::connection_success_response

  Class registration:

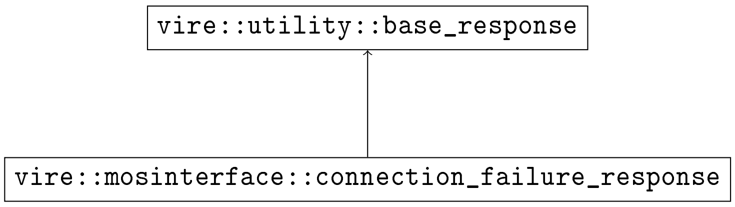
  - name: `"vire::mosinterface::connection_success_response"`
  - version: "1.0"

C++

```
struct connection_success_response
  : public vire::utility::base_response
{
  typedef vire::resource::resource_status_record resource_status_record; // Type alias

  // Attributes:
  std::vector<resource_status_record> resources_snapshot; // Requested resources snapshot
};
```

Figure 51: The structure of the connection success response emitted by the CMS server to the Vire server (C++).

- The `vire::mosinterface::connection_failure_response` class represents the response sent back to the Vire server by the CMS server through the service channel in case of failure.


vire::utility::base_response
vire::mosinterface::connection_failure_response

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

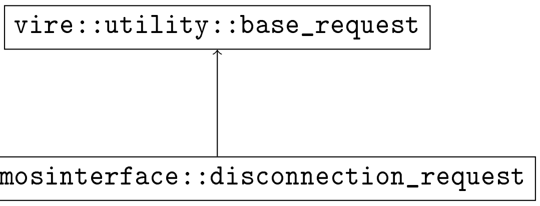
Figure 52: JSON formatted connection success response object (class `vire::mosinterface::connection_success_response`.

```cpp
struct connection_failure_response
  : public vire::utility::base_response
{
  // Nested type alias:
  typedef vire::utility::model_identifier error_identifier;

  // Nested error type aliases:
  typedef vire::utility::invalid_context_error invalid_context_error;
  typedef vire::utility::invalid_setup_id_error invalid_setup_id_error;

  // Nested error type:
  struct unknown_resources_error : public vire::utility::base_error {
    std::vector<std::string> unknown_paths; // List of unknown resources' paths
  };

  // Attributes:
  error_identifier error_id; // Error type identifier
  XXX_error        error;    // Embedded error record of one of the nested error type above
};
```

Figure 53: The structure of the connection failure response emitted by the CMS server to the Vire server (C++).

D.3 Disconnection of the Vire server from the CMS server

- The `vire::mosinterface::disconnection_request` class represents a disconnection request sent by the Vire server to the CMS server through the service channel.


vire::utility::base_request
vire::mosinterface::disconnection_request

  Class registration:

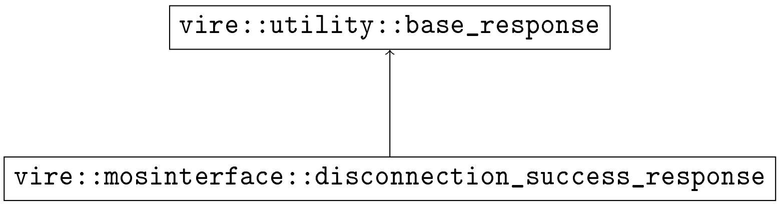
  - name: `"vire::mosinterface::disconnection_request"`
  - version: "1.0"

C++

```
struct disconnection_request : public vire::utility::base_request {
};
```

Figure 54: The structure of the disconnection request object to be emitted by the Vire server to the CMS server (C++).

- The `vire::mosinterface::disconnection_success_response` class represents the response sent back to the Vire server by the CMS server through the service channel in case of success.


vire::utility::base_response
vire::mosinterface::disconnection_success_response

  Class registration:

  - name: `"vire::mosinterface::disconnection_success_response"`
  - version: "1.0"

C++

```
struct disconnection_success_response
  : public vire::utility::base_response
{
};
```

Figure 55: The structure of the disconnection success response emitted by the CMS server to the Vire server (C++).

D.4 Resource related payload objects

D.4.1 Resource Pub/Sub service

- The `vire::resource::resource_pubsub_request` object is responsible of demanding the activation/deactivation of the Pub/Sub service associated to a given resource (fig. 56).

C++
```
struct resource_pubsub_request
  : public vire::utility::base_request
{
  // Attributes:
  std::string path;       // The resource path.
  bool        activation; // Pub/Sub service de/activation flag.
};
```

Figure 56: The structure of the `vire::resource::resource_pubsub_request` class (C++).

- The `vire::resource::resource_pubsub_success_response` object encapsulate a successfull response of the CMS server to the Vire server concerning the activation/deactivation of the Pub/Sub service associated to a given resource (fig. 57).

C++
```
struct resource_pubsub_success_response
  : public vire::utility::base_response
{
  // Type alias:
  typedef vire::utility::model_identifier pubsub_access_identifier;

  // Attributes:
  std::string              path;             // The resource path.
  bool                     activation;       // The effective activation flag.
  pubsub_access_identifier pubsub_access_id; // The type of Pub/Sub service access
  XXX_pubsub_access        pubsub_access;    // If activation is set, this describes the
                                             // access to the Pub/Sub service.
};
```

Figure 57: The structure of the `vire::resource::resource_pubsub_success_response` class (C++).

- The `vire::resource::amqp_pubsub_access` object describes the access to Pub/Sub service through RabbitMQ (fig. 58).
- The `vire::resource::resource_pubsub_failure_response` object describes a failure response concerning a request on Pub/Sub service associated to a given resource (fig. 59).

C++

```
struct amqp_pubsub_access
{
  // Attributes:
  std::string channel; // The RabbitMQ Pub/Sub channel.
  std::string binding; // The binding dedicated to this Pub/Sub service.
  std::string key;     // The Pub/Sub specific key (optional?).
};
```

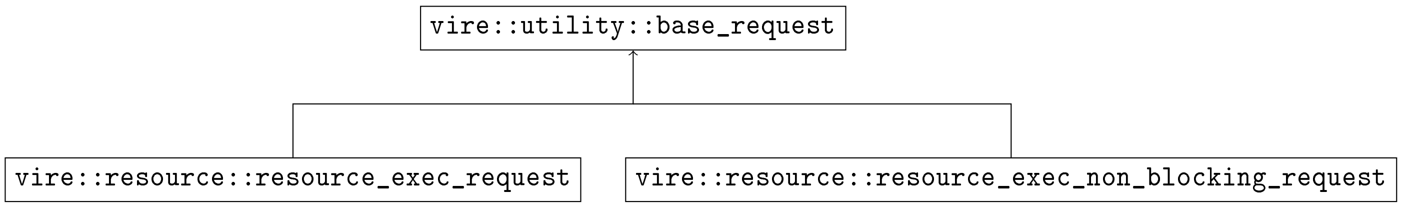
Figure 58: The structure of the `vire::resource::amqp_pubsub_access_type` class (C++).

C++

```
struct resource_pubsub_failure_response
  : public vire::utility::base_response
{
  // Nested type alias:
  typedef vire::utility::model_identifier error_identifier;

  // Nested error type aliases:
  typedef vire::utility::invalid_context_error  invalid_context_error;
  typedef vire::utility::invalid_resource_error invalid_resource_error;

  // Nested error type:
  struct no_pubsub_resource_error : public vire::utility::base_error {
    std::string path; // The path of the resource without Pub/Sub service support
  };

  // Attributes:
  error_identifier error_id; // Error type identifier
  XXX_error        error;    // Embedded error record of one of the nested error type above
};
```

Figure 59: The structure of the `vire::resource::resource_pubsub_failure_response` class (C++).

D.4.2 Resource execution


vire::utility::base_request
vire::resource::resource_exec_request
vire::resource::resource_exec_non_blocking_request

- The `vire::resource::resource_fetch_status_request` object demands to the CMS server an updated status record associated to a given resource (fig. 60).

C++
```
struct resource_fetch_status_request
  : public vire::utility::base_request
{
  // Attributes:
  std::string path; // Resource path.
};
```

Figure 60: The structure of a `vire::utility::resource_fetch_status_request` object (C++).

- The `vire::resource::resource_fetch_status_success_response` object transmits the updated/current status record associated to a given resource (fig. 61).

C++
```
struct resource_fetch_status_success_response
  : public vire::utility::base_response
{
  // Nested type alias:
  typedef vire::resource::resource_status_record resource_status_record;

  // Attributes:
  resource_status_record status; // The resource status record.
};
```

Figure 61: The structure of a `vire::utility::resource_fetch_status_success_response` object (C++).

- The `vire::resource::resource_fetch_status_failure_response` object describes a failure detected by the CMS server in response to a resource fetch status request.
- The `vire::resource::resource_exec_request` object represent a resource execution request in blocking (synchronous) mode.
- `vire::resource::resource_exec_success_response`
- `vire::resource::resource_exec_failure_response`
- `vire::resource::resource_exec_non_blocking_request`

C++

```
struct resource_fetch_status_failure_response
  : public vire::utility::base_response
{
  // Nested type alias:
  typedef vire::utility::model_identifier error_identifier;

  // Nested error type aliases:
  typedef vire::utility::invalid_context_error   invalid_context_error;
  typedef vire::resource::invalid_resource_error invalid_resource_error;

  // Attributes:
  error_identifier error_id; // Error type identifier
  XXX_error        error;    // Embedded error record of one of the nested error type above
};
```

Figure 62: The structure of a `vire::utility::resource_fetch_status_failure_response` object (C++).

C++

```
struct resource_exec_request
  : public vire::utility::base_request
{
  typedef vire::resource::method_argument method_argument;

  // Attributes:
  std::string                  path;            // Resource path.
  std::vector<method_argument> input_arguments; // Embedded error record of one of
                                                // the nested error type above.
};
```

Figure 63: The structure of a `vire::utility::resource_fetch_status_failure_response` object (C++).

- `vire::resource::resource_exec_non_blocking_success_response`
- `vire::resource::resource_exec_non_blocking_failure_response`