Supvisors

Release 0.18

Julien Le Cléach

May 02, 2024

CONTENTS

Supvisors is a Control System for Distributed Applications, based on multiple instances of Supervisor.

INTRODUCTION

1.1 Overview

Supvisors is a control system for distributed applications over multiple Supervisor instances.

The **Supvisors** software is born from a common need in embedded systems where applications are distributed over several nodes. The problematic comes with the following challenges:

- to have a status of the processes,

- to have a synthetic status of the applications based on the processes status,

- to have basic statistics about the resources taken by the applications,

- to have a basic status of the nodes,

- to control applications and processes dynamically,

- to distribute the same application over different platforms (developer machine, integration platform, etc),

- to deal with resources (CPU, memory, network, etc),

- to deal with failures:

    - missing node when starting,

    - crash of a process,

    - crash of a node.

As a bonus:

- it should be free, open source, not subject to export control issues,

- it shouldn't require specific administration rights (root).

Supervisor can handle a part of the requirements but it only works on a single UNIX-like operating system. The Supervisor website references some third parties dealing with multiple Supervisor instances but they only consist in dashboards and they focus on the nodes rather than on the applications and their possible distribution over nodes. Nevertheless, the extensibility of Supervisor makes it possible to implement the missing requirements.

Supvisors works as a Supervisor plugin and is intended for those who are already familiar with Supervisor or who have neither the time nor the resources to invest in a complex orchestration tool like Kubernetes.

1.2 Definitions

Here follows a few definitions of terms are used throughout the present documentation.

The term *Node* refers to an UNIX operating system having a dedicated host name and IP address.

A Supervisor *instance* is a Supervisor damon running on a *Node*, and with a distinct HTTP configuration.

A **Supvisors** *instance* refers to a Supervisor *instance* including a **Supvisors** extension.

Supvisors corresponds to the distributed software grouping all the **Supvisors** *instances* configured to work together.

Multiple **Supvisors** *instances* running on the same *Node* do not necessarily belong to the same **Supvisors**.

A Supervisor *Process* is a structure whose configuration is described in a Supervisor *Program* (or *Homogeneous Process Group*). It is managed by a single Supervisor *instance*. This Supervisor *instance* is able to spawn a UNIX process based on this structure and to control it.

A Supervisor *Group* (or *Heterogeneous Process Group*) is a collection of Supervisor *Programs*. It is also managed by a single Supervisor *instance*.

A **Supvisors** *Process* is the union of all Supervisor *Processes* sharing the name Supervisor *Group* name and *Process* name within **Supvisors**. Such a Supervisor *Process* is not necessarily defined in all Supervisor *instances*.

A **Supvisors** *Application* is the collection of all **Supvisors** *Processes* sharing the name Supervisor *Group* name within **Supvisors**.

The definitions of all **Supvisors** *Processes* and **Supvisors** *Applications* are shared across **Supvisors** and all **Supvisors** *instances* have control over them, even if the process definition is unknown to their respective hosting Supervisor *instance*.

1.3 Platform Requirements

Supvisors has been tested and is known to run on Linux (Rocky 8.5, RedHat 8.2 and Ubuntu 20.04 LTS).

Supvisors will not run at all under any version of Windows.

Supvisors works with `Python 3.6` to `Python 3.12`.

An old release of **Supvisors** (version 0.1, available on PyPi) works with `Python 2.7` (and previous versions of Supervisor, i.e. 3.3.0) but is not maintained anymore.

The CSS of the Dashboard has been written for Firefox ESR 91.3.0. The compatibility with other browsers or other versions of Firefox is unknown.

1.4 Installation

Supvisors has the following dependencies:

Package	Minimal release	Usage
Supervisor	4.2.4	Base software, extended by **Supvisors**
psutil	5.7.3	*(optional)* Information about system usage
matplotlib	3.3.3	*(optional)* Graphs for Dashboard
lxml	4.6.2	*(optional)* XSD validation of the XML rules file
Flask-RESTX	0.5.1 (py36) 1.1.0 (py37+)	*(optional)* Expose the Supervisor and Supvisors XML-RPC API through a REST API
PyZMQ	22.0.3	*(optional)* Alternative for the Supvisors Event interface
websockets	10.4	*(optional)* Alternative for the Supvisors Event interface (requires Python 3.7+)

1.4.1 With an Internet access

Supvisors can be installed with `pip install`:

```
# minimal install (including Supervisor)
[bash] > pip install supvisors

# install including all optional dependencies
[bash] > pip install supvisors[all]

# install for dashboard statistics and graphs only
# (includes psutil and matplotlib)
[bash] > pip install supvisors[statistics]

# install for XML validation only (includes lxml)
[bash] > pip install supvisors[xml_valid]

# install for the REST API (includes flask-restx)
[bash] > pip install supvisors[flask]

# install for the ZMQ event interface (includes PyZMQ)
[bash] > pip install supvisors[zmq]

# install for the Websockets event interface (includes websockets, requires Python 3.7+)
[bash] > pip install supvisors[ws]
```

1.4.2 Without an Internet access

All the dependencies have to be installed prior to **Supvisors**. Refer to the documentation of these dependencies.

Finally, get the latest release from Supvisors releases, unzip the archive and enter the directory **supvisors-{version}**.

Install **Supvisors** with the following command:

```
[bash] > python setup.py install
```

1.4.3 Additional commands

During the installation, a few additional commands are added to the BINDIR (directory that the **Python** installation has been configured with):

- **supvisorsctl** provides access to the extended **Supvisors** API when used with the option -s URL, which is missed from the Supervisor **supervisorctl** (refer to the *supervisorctl extension* part).

- **supvisorsflask** provides a Flask-RESTX application that exposes the Supervisor and **Supvisors** XML-RPC APIs through a REST API (refer to the *REST API* page).

> **Attention:** **supvisorsflask** uses the Flask's built-in server, which should not be an issue as it is unlikely that this interface ever needs to be scaled.

1.5 Running Supvisors

Supvisors runs as a plugin of Supervisor so it follows the same principle as Running Supervisor but using multiple UNIX-like operating systems.

Although **Supvisors** was originally designed to handle exactly one Supervisor instance per node, it can handle multiple Supervisor instances on each node since the version 0.11.

However, the Supervisor configuration file **MUST**:

- be configured with an internet socket (refer to the inet-http-server section settings) ;

- include the [rpcinterface:supvisors] and the [ctlplugin:supvisors] sections (refer to the *Configuration* page) ;

- be consistent on all considered nodes, more particularly attention must be paid to the list of declared **Supvisors** instances and the IP ports used.

> **Important:** A script may be required to start Supervisor on several nodes if not configured to run automatically at startup (**ssh** loop for example).

CONFIGURATION

2.1 Supervisor's Configuration File

This section explains how **Supvisors** uses and complements the Supervisor configuration.

As written in the introduction, all Supervisor instances **MUST** be configured with an internet socket. `username` and `password` can be used at the condition that the same values are used for all Supervisor instances.

```
[inet_http_server]
port=:60000
;username=lecleach
;password=p@$$w0rd
```

Apart from the `rpcinterface` and `ctlplugin` sections related to **Supvisors**, all Supervisor instances can have a completely different configuration, including the list of programs.

2.1.1 rpcinterface extension point

Supvisors extends the Supervisor's XML-RPC API.

```
[rpcinterface:supvisors]
supervisor.rpcinterface_factory = supvisors.plugin:make_supvisors_rpcinterface
```

The parameters of **Supvisors** are set in this section of the Supervisor configuration file. It is expected that some parameters are strictly identical for all **Supvisors** instances otherwise unpredictable behavior may happen. The present section details where it is applicable.

`supvisors_list`

> The list of expected **Supvisors** instances to handle, separated by commas. The elements of `supvisors_list` define how the **Supvisors** instances will share information between them and MUST be identical to all **Supvisors** instances or unpredictable behavior may happen.
>
> The exhaustive form of an element matches `<nick_identifier>host_name:http_port`, where:
>
> - `nick_identifier` is the optional but **unique** Supervisor identifier (it can be set in the Supervisor configuration or in the command line when starting the `supervisord` program) ;
>
> - `host_name` is the name of the node where the **Supvisors** instance is running ;
>
> - `http_port` is the port of the internet socket used to communicate with the Supervisor instance (obviously unique per node).

Default: the host name, as given by the `socket.gethostname()` function.

Required: No.

Identical: Yes.

Note: `host_name` can be the host name, the fully qualified domain name, one of the node aliases or IP addresses. **Supvisors** uses the `socket.gethostbyaddr(host_name)` function to identify the **Supvisors** instance declared.

Note: In user-related features (options, rules, XML-RPC) where a **Supvisors** identifier is requested, `nick_identifier` and `host_name:http_port` can both be used indifferently.

Attention: The chosen host name, alias or IP address must be known to every nodes in the `supvisors_list` on the network interface considered. If it's not the case, check the network configuration.

Hint: If the **Supvisors** is configured with at most one **Supvisors** instance per host, the `host_name` is a fully acceptable declaration.

Hint: `nick_identifier` can be seen as a nickname that may be more user-friendly than a `host_name` or a `host_name:http_port` when displayed in the **Supvisors** Web UI or used in the *Supvisors' Rules File*.

Attention: if `http_port` is not provided, the local **Supvisors** instance takes the assumption that the other **Supvisors** instance uses the same `http_port`. The `http_port` *MUST* be set if there multiple **Supvisors** instances running on the same node.

Important: As a general rule, **Supvisors** uses the form `host_name:http_port` to exchange information between the **Supvisors** instances. Unless a `nick_identifier` is provided, this form will also be used on all user interfaces (configuration files, Web UI, XML-RPC and logs).

`software_name`

An optional string that will be displayed at the top of the **Supvisors** Web UI. It is intended to correspond to the name of the user software.

Default: None.

Required: No.

Identical: No.

`software_icon`

An optional image that will be displayed at the top of the **Supvisors** Web UI. It is intended to correspond to the name of the user software.

Default: None.

Required: No.

Identical: No.

Hint: `software_icon` are `software_name` can both be set and will be displayed in that order if required.

`multicast_group`

The IP address and port number of the Multicast Group where the **Supvisors** instances will share their identity between them, separated by a colon (example: `239.0.0.1:1234`). This is an alternative to the `supvisors_list` option, that allows **Supvisors** to work in a discovery mode.

Default: None.

Required: No.

Identical: Yes.

Hint: Although it is an alternative, this option can yet be combined with `supvisors_list`. In this case, the **Supvisors** instances declared in the `supvisors_list` option will form an initial group that may grow when other unknown **Supvisors** instances declare themselves.

`multicast_interface`

The network interface where the **Supvisors** multicast group will be bound. If not set, `INADDR_ANY` will be applied so as to bind on all network interfaces.

Default: None.

Required: No.

Identical: Yes.

`multicast_ttl`

The time-to-live of a message sent on the **Supvisors** multicast interface.

Default: `1`.

Required: No.

Identical: Yes.

`stereotypes`

A list of names, separated by commas, that can be used to reference a kind of **Supvisors** instance in the rules files. The local **Supvisors** instance will be tagged using these names and will share this information with the other **Supvisors** instances. Although it has been designed to support the discovery mode, it is made available to the standard mode.

Default: None.

Required: No.

Identical: No.

`rules_files`

A space-separated sequence of file globs, in the same vein as supervisord include section. Instead of `ini` files, XML rules files are expected here. Their content is described in *Supvisors' Rules File*. It is highly

---

recommended that this parameter is identical to all **Supvisors** instances or the startup sequence would be different depending on which **Supvisors** instance is given the *Master* role.

Default: None.

Required: No.

Identical: Yes.

auto_fence

When true, **Supvisors** will definitely disconnect a **Supvisors** instance that is inactive. This functionality is detailed in *Auto-Fencing*.

Default: `false`.

Required: No.

Identical: No.

event_link

The communication protocol type used to publish all **Supvisors** events (Instance, Application and Process events). Value in [`NONE` ; `ZMQ` ; `WS`]. Other protocols may be considered in the future. If set to `NONE`, the interface is not available. If set to `ZMQ`, events are published through a ZeroMQ TCP socket. If set to `WS`, events are published through websockets (requires a **Python** version 3.7 or later). The protocol of this interface is detailed in *Event interface*.

Default: NONE.

Required: No.

Identical: No.

event_port

The port number used to publish all **Supvisors** events (Instance, Application, Process and Statistics events). The protocol of this interface is detailed in *Event interface*.

Default: local Supervisor HTTP port + 1.

Required: No.

Identical: No.

synchro_options

The conditions applied by **Supvisors** to exit the `INITIALIZATION` state. Multiple values in [`LIST` ; `TIMEOUT` ; `CORE` ; `USER`], separated by commas. If STRICT is selected, **Supvisors** exits the `INITIALIZATION` state when all the **Supvisors** instances declared in the `supvisors_list` option are in the RUNNING state. If `LIST` is selected, **Supvisors** exits the `INITIALIZATION` state when all known **Supvisors** instances (including those declared in the `supvisors_list` option **AND** those discovered) are in the RUNNING state. If `TIMEOUT` is selected, **Supvisors** exits the `INITIALIZATION` state after the duration defined in the `synchro_timeout` option. If CORE is selected, **Supvisors** exits the `INITIALIZATION` state when all the **Supvisors** instances identified in the `core_identifiers` option are in a RUNNING state. If USER is selected, **Supvisors** exits the `INITIALIZATION` state as soon as the *Master* instance is set, which can be triggered upon user request (using the **Supvisors** Web UI, the end_sync XML-RPC or the end_sync command of `supervisorctl`). The use of this option is more detailed in *Synchronizing Supvisors instances*.

Default: `STRICT,TIMEOUT,CORE`.

Required: No.

Identical: No.

---

synchro_timeout

The time in seconds that **Supvisors** waits for all expected **Supvisors** instances to publish their TICK. Value in [15 ; 1200]. This option is taken into account only if TIMEOUT is selected in the synchro_options. The use of this option is more detailed in *Synchronizing Supvisors instances*.

Default: 15.

Required: No.

Identical: No.

inactivity_ticks

By default, a remote **Supvisors** instance is considered inactive when no tick has been received from it while 2 ticks have been received fom the local **Supvisors** instance, which may be a bit strict in a busy network. This delay is also used when starting / stopping a process, considering that process events are expected in due time. This option allows to loosen the constraint. Value in [2 ; 720].

Default: 2.

Required: No.

Identical: No.

Note: This option originates from a previous **Supvisors** design where the event publication mechanism did not allow to assess the consideration of the request by the remote **Supvisors** instance. In the current design, an inactive **Supvisors** instance is detected before a timeout is triggered on missing expected events. The option has thus less interest but it is kept anyway for robustness value.

---

core_identifiers

A list of names, separated by commas. These names can taken from the names deduced from the supvisors_list option and / or from the union of all stereotypes shared within **Supvisors**. If the **Supvisors** instances of this subset are all in a RUNNING state and the CORE value is set in the synchro_options option, this will put an end to the synchronization phase in **Supvisors**. Independently from the CORE option being used, **Supvisors** will preferably take a member of this list when selecting a *Master* instance. This parameter must be identical to all **Supvisors** instances or unpredictable behavior may happen.

Default: None.

Required: No.

Identical: Yes.

disabilities_file

The file path that will be used to persist the program disabilities. This option has been added in support of the Supervisor request #591 - New Feature: disable/enable. The persisted data will be serialized in a JSON string so a .json extension is recommended.

Default: None.

Required: No.

Identical: No.

Hint: Both absolute and relative paths are supported. User expansion is also allowed. It is expected that the folder tree exists. However **Supvisors** will try to create it if not, unless write permission is denied.

Note: If the file does not exist at startup, all processes are enabled by default and a first version of the file will be written down accordingly.

---

starting_strategy

The strategy used to start applications on **Supvisors** instances. Possible values are in { CONFIG, LESS_LOADED, MOST_LOADED, LOCAL , LESS_LOADED_NODE, MOST_LOADED_NODE}. The use of this option is detailed in *Starting strategy*. It is highly recommended that this parameter is identical to all **Supvisors** instances or the startup sequence would be different depending on which **Supvisors** instance is the *Master*.

Default: CONFIG.

Required: No.

Identical: Yes.

conciliation_strategy

The strategy used to solve conflicts upon detection that multiple instances of the same program are running. Possible values are in { SENICIDE, INFANTICIDE, USER, STOP, RESTART, RUNNING_FAILURE }. The use of this option is detailed in *Conciliation*. It is highly recommended that this parameter is identical to all **Supvisors** instances or the conciliation phase would behave differently depending on which **Supvisors** instance is the *Master*.

Default: USER.

Required: No.

Identical: Yes.

stats_enabled

By default, **Supvisors** can provide basic statistics on the node and the processes spawned by Supervisor on the **Supvisors** *Dashboard*, provided that the psutil module is installed. This option can be used to adjust or disable the collection of the host and/or process statistics. Possible values are in { OFF, HOST, PROCESS, ALL }. For backwards compatibility, boolean values true and false have been kept and are respectively equal to ALL and OFF.

Default: ALL.

Required: No.

Identical: No.

stats_collecting_period

This is the *minimum* duration between 2 statistics measurements on one entity (host or process). It is not a strict period. Value in [1 ; 3600] seconds.

Default: 10.

Required: No.

Identical: No.

Note: The statistics collection is deferred to a dedicated process of **Supvisors**. The *Collector* is mainly using the psutil module. Regardless of the number of processes to manage, the *Collector* will not take up more than one processor core. If there are more statistics requests than allowed by one processor core, the duration between 2 measurements on the same process will inevitably increase.

If there are *many* processes to deal with and if it is unsuitable to dedicate one whole processor core to process statistics, the `stats_collecting_period` should be increased or the process statistics may be deactivated using the the `stats_enabled` option.

> **Attention:** If there are multiple **Supvisors** instances on the same host, there will be multiple *Collector* processes too. The user should pay attention to set the `stats_collecting_period` option accordingly so that the CPU load remains within acceptable limits.

`stats_periods`

The list of periods for which the statistics will be provided in the **Supvisors** *Dashboard*, separated by commas. Up to 3 int or float values are allowed in [`1` ; `3600`] seconds.

Default: `10`.

Required: No.

Identical: No.

`stats_histo`

The depth of the statistics history. Value in [`10` ; `1500`].

Default: `200`.

Required: No.

Identical: No.

`stats_irix_mode`

The way of presenting process CPU values. If `true`, values are displayed in 'IRIX' mode. If `false`, values are displayed in 'Solaris' mode.

Default: `false`.

Required: No.

Identical: No.

`tail_limit`

In its Web UI, Supervisor provides a page that enables to display the 1024 latest bytes of the process logs. The page is made available by clicking on the process name in the process table. A button is added to refresh it. The size of the logs can be updated through the URL by updating the `limit` attribute. The same function is provided in the **Supvisors** Web UI. This option has been added to enable a default size different than 1024 bytes. It applies to processes logs and Supervisor logs.

Default: `1KB`.

Required: No.

Identical: No.

`tailf_limit`

In its Web UI, Supervisor provides a page that enables to display the 1024 latest bytes of the process logs and that auto-refreshes the page in a `tail -f` manner. The page is made available by clicking on the `Tail -f` button in the process table. The initial size of the logs cannot be updated. The same function is provided in the **Supvisors** Web UI. This option has been added to enable a default size different than 1024 bytes. It applies to processes logs and Supervisor logs.

Default: `1KB`.

Required: No.

Identical: No.

> **Attention:** Setting the `tail_limit` and `tailf_limit` options with very big values may block the web browser. Moderation should be considered.

The logging options are strictly identical to Supervisor's. By the way, it is the same logger that is used. These options are more detailed in supervisord Section values.

`logfile`

The path to the **Supvisors** activity log of the `supervisord` process. This option can include the value `%(here)s`, which expands to the directory in which the Supervisor configuration file was found. If `logfile` is unset or set to `AUTO`, **Supvisors** will use the same logger as Supervisor. It makes it easier to understand what happens when both Supervisor and **Supvisors** log in the same file.

Default: `AUTO`.

Required: No.

Identical: No.

`logfile_maxbytes`

The maximum number of bytes that may be consumed by the **Supvisors** activity log file before it is rotated (suffix multipliers like `KB`, `MB`, and `GB` can be used in the value). Set this value to `0` to indicate an unlimited log size. No effect if `logfile` is unset or set to `AUTO`.

Default: `50MB`.

Required: No.

Identical: No.

`logfile_backups`

The number of backups to keep around resulting from **Supvisors** activity log file rotation. If set to `0`, no backups will be kept. No effect if `logfile` is unset or set to `AUTO`.

Default: `10`.

Required: No.

Identical: No.

`loglevel`

The logging level, dictating what is written to the **Supvisors** activity log. One of [`critical`, `error`, `warn`, `info`, `debug`, `trace`, `blather`]. See also: supervisord Activity Log Levels. No effect if `logfile` is unset or set to `AUTO`.

Default: `info`.

Required: No.

Identical: No.

2.1.2 ctlplugin extension point

Supvisors also extends supervisorctl. This feature is not described in Supervisor documentation.

```
[ctlplugin:supvisors]
supervisor.ctl_factory = supvisors.supvisorsctl:make_supvisors_controller_plugin
```

2.1.3 Configuration File Example

```
[inet_http_server]
port=:60000
;username=lecleach
;password=p@$$w0rd

[supervisord]
logfile=./log/supervisord.log
loglevel=info
pidfile=/tmp/supervisord.pid

[rpcinterface:supervisor]
supervisor.rpcinterface_factory = supervisor.rpcinterface:make_main_rpcinterface

[supervisorctl]
serverurl=http://localhost:60000

[include]
files = common/*/*.ini %(host_node_name)s/*.ini  %(host_node_name)s/*/*.ini

[rpcinterface:supvisors]
supervisor.rpcinterface_factory = supvisors.plugin:make_supvisors_rpcinterface
software_name = Supvisors Tests
software_icon = ../ui/img/icon.png
supvisors_list = cliche81,<cliche82>192.168.1.49,cliche83:60000:60001,cliche84
rules_files = ./etc/my_movies*.xml
stereotypes = @TEST
auto_fence = false
event_link = WS
event_port = 60002
synchro_timeout = 20
inactivity_ticks = 3
core_identifiers = cliche81,cliche82
disabilities_file = /tmp/disabilities.json
starting_strategy = CONFIG
conciliation_strategy = USER
stats_enabled = true
stats_periods = 5,60,600
stats_histo = 100
stats_irix_mode = false
tail_limit = 50MB
tailf_limit = 50MB
logfile = ./log/supvisors.log
logfile_maxbytes = 50MB
```

(continues on next page)

```
logfile_backups = 10
loglevel = debug

[ctlplugin:supvisors]
supervisor.ctl_factory = supvisors.supvisorsctl:make_supvisors_controller_plugin
```

2.2 Supvisors' Rules File

This part describes the contents of the XML rules files declared in the `rules_files` option.

Basically, a rules file contains rules that define how applications and programs should be started and stopped, and the quality of service expected. It relies on the Supervisor group and program definitions.

Important: *About the declaration of Supervisor groups/processes in a rules file*

It is important to notice that all applications declared in this file will be considered as *Managed* by **Supvisors**. The main consequence is that **Supvisors** will try to ensure that one single instance of the program is running over all the **Supvisors** instances considered. If two instances of the same program are running in two different **Supvisors** instances, **Supvisors** will consider this as a conflict. Only the *Managed* applications have an entry in the navigation menu of the **Supvisors** Web UI.

The groups declared in Supervisor configuration files and not declared in a rules file will thus be considered as *Unmanaged* by **Supvisors**. So they have no entry in the navigation menu of the **Supvisors** web page. There can be as many running instances of the same program as Supervisor allows over the available **Supvisors** instances.

If the lxml package is available on the system, **Supvisors** uses it to validate the XML rules files before they are used.

Hint: It is still possible to validate the XML rules files manually. The XSD file `rules.xsd` used to validate the XML can be found in the **Supvisors** package. Just use **xmllint** to validate:

```
[bash] > xmllint --noout --schema rules.xsd user_rules.xml
```

2.2.1 `<application>` rules

Here follows the definition of the attributes and rules applicable to an `application` element.

name

> This attribute gives the name of the application. The name MUST match a Supervisor group name.
>
> *Default*: None.
>
> *Required*: Yes, unless a `pattern` attribute is provided.

pattern

> A regex matching one or more Supervisor group names is expected in this attribute. Refer to *Using patterns* for more details.
>
> *Default*: None.
>
> *Required*: Yes, unless a `name` attribute is provided.

Note: The options below can be declared in any order in the `application` section.

`distribution`

> In the introduction, it is written that the aim of **Supvisors** is to manage distributed applications. However, it may happen that some applications are not designed to be distributed (for example due to inter-process communication design) and thus distributing the application processes over multiple nodes would just make the application non operational. If set to `ALL_INSTANCES`, **Supvisors** will distribute the application processes over the applicable **Supvisors** instances. If set to `SINGLE_INSTANCE`, **Supvisors** will start all the application processes in the same **Supvisors** instance. If set to `SINGLE_NODE`, **Supvisors** will distribute all the application processes over a set of **Supvisors** instances running on the same node.
>
> *Default*: `ALL_INSTANCES`.
>
> *Required*: No.

Note: When a single **Supvisors** instance is running on each node, `SINGLE_INSTANCE` and `SINGLE_NODE` are strictly equivalent.

`identifiers`

> This element is only used when `distribution` is set to `SINGLE_INSTANCE` or `SINGLE_NODE` and gives the list of **Supvisors** instances where the application programs can be started.
>
> The names are to be taken either the names deduced from the `supvisors_list` option defined in *rpcinterface extension point*, and / or from the declared *Instance aliases*, and / or or from a stereotype as provided in the `stereotypes` option, and separated by commas.
>
> Special values can be used. The *wildcard symbol* `*` stands for all names deduced from `supvisors_list`. Any name list including a `*` is strictly equivalent to `*` alone.
>
> The *hashtag symbol* `#` can be used with a `pattern` definition and eventually complemented by a list of deduced names. The aim is to assign the Nth deduced name of `supvisors_list` or the Nth name of the subsequent list (made of names deduced from `supvisors_list`) to the Nth instance of the application, **assuming that 'N' is provided at the end of the application name, preceded by a dash or an underscore**. Yeah, a bit tricky to explain... Examples will be given in *Using patterns and signs*.
>
> *Default*: `*`.
>
> *Required*: No.

Attention: When the distribution of the application is restricted (`distribution` not set to `ALL_INSTANCES`), the rule `identifiers` of the application programs is not considered.

`start_sequence`

> This element gives the starting rank of the application in the `DISTRIBUTION` state, when applications are started automatically. When `<= 0`, the application is not started. When `> 0`, the application is started in the given order.
>
> *Default*: `0`.
>
> *Required*: No.

`stop_sequence`

This element gives the stopping rank of the application when all applications are stopped just before **Supvisors** is restarted or shut down. This value must be positive. If not set, it is defaulted to the `start_sequence` value. **Supvisors** stops the applications sequentially from the greatest rank to the lowest.

Default: `start_sequence` value.

Required: No.

> **Attention:** The `stop_sequence` is **not** taken into account:
>
> - when calling Supervisor's `restart` or `shutdown` XML-RPC,
>
> - when stopping the **supervisord** daemon.
>
> It only works when calling **Supvisors**' `restart` or `shutdown` XML-RPC.

starting_strategy

The strategy used to start applications on **Supvisors** instances. Possible values are in { `CONFIG`, `LESS_LOADED`, `MOST_LOADED`, `LOCAL` }. The use of this option is detailed in *Starting strategy*.

Default: the value set in the *rpcinterface extension point* of the Supervisor configuration file.

Required: No.

starting_failure_strategy

This element gives the strategy applied upon a major failure, i.e. happening on a required process, in the starting phase of an application. The possible values are { `ABORT`, `STOP`, `CONTINUE` } and are detailed in *Starting Failure strategy*.

Default: `ABORT`.

Required: No.

running_failure_strategy

This element gives the strategy applied when the application loses running processes due to a **Supvisors** instance that becomes silent (crash, power down, network failure, etc). This value can be superseded by the value set at program level. The possible values are { `CONTINUE`, `RESTART_PROCESS`, `STOP_APPLICATION`, `RESTART_APPLICATION`, `SHUTDOWN`, `RESTART` } and are detailed in *Running Failure strategy*.

Default: `CONTINUE`.

Required: No.

operational_status

This element contains the formula that will be used to evaluate the operational status of the application, displayed in the **Supvisors** Web UI, and has no other impact on any **Supvisors** function. The formula will be parsed using the Python module `AST`. The exhaustive list of operators and functions supported by **Supvisors** is: `and`, `or`, `not`, `any` and `all`. Parenthesis can also be used. The operands must be string values, between quotes or double-quotes, corresponding to a program name of the application, or a pattern matching one or multiple program names. Multiple program names must be used as an argument of `any` or `all`. When set, the `required` value of the `programs` elements is not considered.

Default: None.

Required: No.

programs

This element is the grouping section of all `program` rules that are applicable to the application. Obviously, the `programs` element of an application can include multiple `program` elements.

Default: None.

Required: No.

program

In a `programs` section, this element defines the rules that are applicable to the program whose name matches the `name` or `pattern` attribute of the element. The `name` must match exactly a program name in the program list of the Supervisor group definition for the application considered here.

Default: None.

Required: No.

2.2.2 `<program>` rules

The `program` element defines the rules applicable to at least one program. This element must be included in an `programs` element. Here follows the definition of the attributes and rules applicable to this element.

Note: The options below can be declared in any order in the `program` section.

---

name

This attribute MUST match exactly the name of a program as defined in Supervisor program settings.

Default: None.

Required: Yes, unless an attribute `pattern` is provided.

pattern

A regex matching one or more Supervisor program names is expected in this attribute. Refer to the *Using patterns* for more details.

Default: None.

Required: Yes, unless an attribute `name` is provided.

identifiers

This element gives the list of **Supvisors** instances where the program can be started.

The names are separated by commas and have to be taken from:

- either the names declared in the `supvisors_list` option defined in the *rpcinterface extension point*,
- and / or the declared *Instance aliases*,
- and / or a stereotype provided in the `stereotypes` option.

Special values can be applied. The *wildcard symbol* `*` stands for all names deduced from `supvisors_list`. Any name list including a `*` is strictly equivalent to `*` alone.

The *hashtag symbol* `#` and the *at symbol* `@` can be used with a `pattern` definition and eventually complemented by a list of deduced names. The aim is to assign the Nth deduced name of `supvisors_list` or the Nth name of the subsequent list (made of names deduced from `supvisors_list`) to the Nth instance of the program in a homogeneous process group. Examples will be given in *Using patterns and signs*.

Default: `*`.

Required: No.

---

required

This element gives the importance of the program for the application. If `true` (resp. `false`), a failure of the program is considered major (resp. minor). When the `operational_status` element of the `application` element is set, this element is ignored. This information is mainly used to give the operational status of the application in the Web UI and has no other impact on any **Supvisors** function.

Default: `false`.

Required: No.

start_sequence

This element gives the starting rank of the program when the application is starting. When `<= 0`, the program is not started automatically. When `> 0`, the program is started automatically in the given order.

Default: `0`.

Required: No.

stop_sequence

This element gives the stopping rank of the program when the application is stopping. This value must be positive. If not set, it is defaulted to the `start_sequence` value. **Supvisors** stops the processes sequentially from the greatest rank to the lowest.

Default: `start_sequence` value.

Required: No.

wait_exit

If the value of this element is set to `true`, **Supvisors** waits for the process to exit before starting the next sequence. This may be particularly useful for scripts used to load a database, to mount disks, to prepare the application working directory, etc.

Default: `false`.

Required: No.

expected_loading

This element gives the expected percent usage of *resources*. The value is a estimation and the meaning in terms of resources (CPU, memory, network) is in the user's hands. When multiple **Supvisors** instances are available, **Supvisors** uses the `expected_loading` value to distribute the processes over the available **Supvisors** instances, so that the system remains safe.

Default: `0`.

Required: No.

Note: *About the choice of an user estimation*

Although **Supvisors** may be taking measurements on each node where it is running, it has been chosen not to use these figures for the loading purpose. Indeed, the resources consumption of a process may be very variable in time and is not foreseeable. It is recommended to give a value based on an average usage of the resources in the worst case configuration and to add a margin corresponding to the standard deviation.

---

starting_failure_strategy

This element gives the strategy applied upon a major failure, i.e. happening on a required process, in the starting phase of an application. This value supersedes the value eventually set at application level. The possible values are { `ABORT`, `STOP`, `CONTINUE` } and are detailed in *Starting Failure strategy*.

Default: `ABORT`.

Required: No.

`running_failure_strategy`

This element gives the strategy applied when the process is running in a **Supvisors** instance that becomes silent (crash, power down, network failure, etc). This value supersedes the value eventually set at application level. The possible values are { `CONTINUE`, `RESTART_PROCESS`, `STOP_APPLICATION`, `RESTART_APPLICATION`, `SHUTDOWN`, `RESTART` } and their impact is detailed in *Running Failure strategy*.

Default: `CONTINUE`.

Required: No.

`reference`

This element gives the name of an applicable `model`, as defined in *<model> rules*.

Default: None.

Required: No.

Note: *About referencing models*

The `reference` element can be combined with all the other elements described above. The rules got from the referenced model are loaded first and then eventually superseded by any other rule defined in the same program section. A model can reference another model. In order to prevent infinite loops and to keep a reasonable complexity, the maximum chain starting from the `program` section has been set to 3. As a consequence, any rule may be superseded twice at a maximum.

Here follows an example of a `program` definition:

```
<program name="prg_00">
    <identifiers>cliche01,cliche03,cliche02</identifiers>
    <required>true</required>
    <start_sequence>1</start_sequence>
    <stop_sequence>1</stop_sequence>
    <wait_exit>false</wait_exit>
    <expected_loading>3</expected_loading>
    <running_failure_strategy>RESTART_PROCESS</running_failure_strategy>
</program>
```

2.2.3 Using patterns

It may be quite tedious to give all this information to every programs, especially if multiple programs use a common set of rules. So two mechanisms are put in place to help.

The first one is the `pattern` attribute that may be used instead of the `name` attribute in a `program` element. It can be used to configure a set of programs in a more flexible way than just considering homogeneous programs, like Supervisor does.

The same `program` options are applicable, whatever a `name` attribute or a `pattern` attribute is used. For a `pattern` attribute, a regex (or a simple substring) matching one Supervisor program name or more is expected.

```
<program pattern="prg_">
    <identifiers>cliche01,cliche03,cliche02</identifiers>
```

```
    <start_sequence>2</start_sequence>
    <required>true</required>
</program>
```

> **Attention:** *About the pattern names*.
>
> Precautions must be taken when using a `pattern` definition. In the previous example, the rules are applicable to every program names containing the "`prg_`" substring, so that it matches **prg_00**, **prg_dummy**, but also **dummy_prg_2**.
>
> As a general rule when looking for program rules, **Supvisors** always searches for a `program` definition having the exact program name set in the `name` attribute, and only if not found, **Supvisors** tries to find a corresponding `program` definition with a matching `pattern`.
>
> It also may happen that multiple patterns match the same program name. In this case, **Supvisors** chooses the pattern with the greatest matching, or arbitrarily the first of them if such a rule does not discriminate enough. So considering the program **prg_00** and the two matching patterns `prg` and `prg_`, **Supvisors** will apply the rules related to `prg_`.

The `pattern` attribute can be applied to `application` elements too. The same logic as per `program` elements applies. This is particularly useful in a context where many users over multiple nodes need to have their own application.

> **Note:** Supervisor does not provide support for *homogeneous* groups of *heterogeneous* programs. So in order to have N running instances of the same application, the only possible solution is to define N times the Supervisor group using a variation in the group name (e.g. an index suffix). It is however possible to include the same Supervisor program definitions into different groups.
>
> Unfortunately, using *homogeneous* program groups with `numprocs` set to N cannot help in the present case because Supervisor considers the program name in the group and not the `process_name`.

2.2.4 Using patterns and signs

The *hashtag symbol #* and *at symbol* in the program `identifiers` are designed for a program that is meant to be started on every **Supvisors** instances available, or on a subset of them.

As an example, based on the following simplified Supervisor configuration:

```
[rpcinterface:supvisors]
supervisor.rpcinterface_factory = supvisors.plugin:make_supvisors_rpcinterface
supvisors_list = cliche01,cliche02,cliche03,cliche04,cliche05

[program:prg]
process_name=prg_%(process_num)02d
numprocs=5
numprocs_start=1
```

Without this option, it is necessary to define rules for all instances of the program.

```
<program name="prg_01">
    <identifiers>cliche01</identifiers>
</program>
```

```
<!-- similar definitions for prg_02, prg_03, prg_04 -->

<program name="prg_05">
    <identifiers>cliche05</identifiers>
</program>
```

Now with this option, the rule becomes more simple.

```
<program pattern="prg_\d+">
    <identifiers>#</identifiers>
</program>
```

It is also possible to give a subset of deduced names.

```
<program pattern="prg_\d+">
    <identifiers>#,cliche04,cliche02</identifiers>
</program>
```

Note: **Supvisors** instances are chosen in accordance with the sequence given in `supvisors_list` or in the subsequent list. In the second example above, **prg_01** will be assigned to `cliche04` and **prg_02** to `cliche02`.

When using the **Supvisors** discovery mode is activated, the **Supvisors** instances are chosen in accordance with their arrival in the system, which is random but fixed when established.

The start index defined in `numprocs_start` has no consequence.

Important: In the initial **Supvisors** design, it was expected that the `numprocs` value set in the program configuration file would exactly match the number of **Supvisors** instances.

However, if the number of **Supvisors** instances is greater than the `numprocs` value, processes will be assigned to the `numprocs` first **Supvisors** instances, with both # and @.

On the other side, if the number of **Supvisors** instances is lower than the `numprocs` value:

- when using @, one process will be assigned to each **Supvisors** instance, leaving the processes in excess unassigned ;
- when using #, all the processes will be equally assigned on the **Supvisors** instances.

Attention: As pointed out just before, **Supvisors** takes the information from the program configuration. So this function will definitely NOT work if the program is unknown to the local Supervisor, which is a relevant use case. As written before, the Supervisor configuration can be different for all **Supvisors** instances, including the definition of groups and programs.

Important: *Convention for application names when using patterns and signs*

When the hashtag is used for the application `identifiers`, **Supvisors** cannot rely on the Supervisor configuration to map the application instances to the **Supvisors** instances.

By convention, the application name MUST end with `-N` or `_N`. The Nth application will be mapped to the Nth deduced name of the list, i.e. the name at index `N-1` in the list.

N must be strictly positive. Zero-padding is allowed, as long as N can be converted into an integer.

2.2.5 `<model>` rules

The second mechanism is the `model` definition. The `program` rules definition is extended to a generic model, that can be defined outside of the application scope, so that the same rules definition can be applied to multiple programs, in any application.

The same options are applicable, **including** the `reference` option (recursion is yet limited to a depth of 2). There is no particular expectation for the `name` attribute of a `model`.

Here follows an example of model:

```
<model name="X11_model">
    <identifiers>cliche01,cliche02,cliche03</identifiers>
    <start_sequence>1</start_sequence>
    <required>false</required>
    <wait_exit>false</wait_exit>
</model>
```

Here follows examples of `program` definitions referencing a model:

```
<program name="xclock">
    <reference>X11_model</reference>
</program>

<program pattern="prg">
    <reference>X11_model</reference>
    <!-- prg-like programs have the same rules as X11_model, but with required=true-->
    <required>true</required>
</program>
```

2.2.6 Instance aliases

When dealing with long lists of **Supvisors** instances, the content of application or program `identifiers` options may impair the readability of the rules file. It is possible to declare instance aliases and to use the alias names in place of the deduced names in the `identifiers` option.

Here follows a few usage examples:

```
<alias name="consoles">console01,console02,console03</alias>
<alias name="servers">server01,server02</alias>

<!-- working alias reference -->
<alias name="all_ok">servers,consoles</alias>

<model name="hci">
    <identifiers>consoles</identifiers>
</model>

<model name="service">
```

(continues on next page)

```
    <identifiers>servers,consoles</identifiers>
</model>
```

Hint: *About aliases referencing other aliases*

Based on the previous example, an alias referencing other aliases will only work if it is placed *before* the aliases referenced.

At some point, the resulting names are checked against the names deduced from the supvisors_list parameter of the *rpcinterface extension point* so any unknown name or remaining alias will simply be discarded.

```
<!-- Correct alias reference -->
<alias name="all_ok">servers,consoles</alias>

<alias name="consoles">console01,console02,console03</alias>
<alias name="servers">server01,server02</alias>

<!-- Wrong alias reference -->
<alias name="all_ko">servers,consoles</alias>
```

2.2.7 Rules File Example

Here follows a complete example of a rules file. It is used in **Supvisors** self tests.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<root>

    <!-- aliases -->
    <alias name="distribute_sublist">#,cliche82,cliche83:60000,cliche84</alias>
    <alias name="consoles">cliche82,cliche81</alias>

    <!-- models -->
    <model name="disk_01">
        <identifiers>cliche81</identifiers>
        <expected_loading>5</expected_loading>
    </model>

    <model name="disk_02">
        <reference>disk_01</reference>
        <identifiers>cliche82</identifiers>
    </model>

    <model name="disk_03">
        <reference>disk_01</reference>
        <identifiers>cliche83:60000</identifiers>
    </model>

    <model name="converter">
        <identifiers>*</identifiers>
        <expected_loading>25</expected_loading>
```

```xml
    </model>

    <!-- import application -->
    <application name="import_database">
        <start_sequence>2</start_sequence>
        <starting_failure_strategy>STOP</starting_failure_strategy>
        <operational_status>all('.*')</operational_status>

        <programs>
            <program pattern="mount_disk_">
                <identifiers>distribute_sublist</identifiers>
                <start_sequence>1</start_sequence>
                <required>true</required>
                <expected_loading>0</expected_loading>
            </program>

            <program name="copy_error">
                <identifiers>cliche81</identifiers>
                <start_sequence>2</start_sequence>
                <required>true</required>
                <wait_exit>true</wait_exit>
                <expected_loading>25</expected_loading>
            </program>
        </programs>

    </application>

    <!-- movies_database application -->
    <application name="database">
        <start_sequence>3</start_sequence>
        <operational_status>all("register.*") and any('movie.*')</operational_status>

        <programs>
            <program pattern="movie_server_">
                <identifiers>#</identifiers>
                <start_sequence>1</start_sequence>
                <expected_loading>5</expected_loading>
                <running_failure_strategy>CONTINUE</running_failure_strategy>
            </program>

            <program pattern="register_movies_">
                <identifiers>#,cliche81,cliche83:60000</identifiers>
                <start_sequence>2</start_sequence>
                <wait_exit>true</wait_exit>
                <expected_loading>25</expected_loading>
            </program>
        </programs>

    </application>

    <!-- my_movies application -->
    <application name="my_movies">
```

```xml
        <start_sequence>4</start_sequence>
        <starting_strategy>CONFIG</starting_strategy>
        <starting_failure_strategy>CONTINUE</starting_failure_strategy>

        <programs>
            <program name="manager">
                <identifiers>*</identifiers>
                <start_sequence>1</start_sequence>
                <stop_sequence>3</stop_sequence>
                <required>true</required>
                <expected_loading>5</expected_loading>
                <running_failure_strategy>RESTART_APPLICATION</running_failure_strategy>
            </program>

            <program name="web_server">
                <identifiers>cliche84</identifiers>
                <start_sequence>2</start_sequence>
                <required>true</required>
                <expected_loading>3</expected_loading>
            </program>

            <program name="hmi">
                <identifiers>consoles</identifiers>
                <start_sequence>3</start_sequence>
                <stop_sequence>1</stop_sequence>
                <expected_loading>10</expected_loading>
                <running_failure_strategy>STOP_APPLICATION</running_failure_strategy>
            </program>

            <program pattern="disk_01_">
                <reference>disk_01</reference>
            </program>

            <program pattern="disk_02_">
                <reference>disk_02</reference>
            </program>

            <program pattern="disk_03_">
                <reference>disk_03</reference>
            </program>

            <program pattern="error_disk_">
                <reference>disk_01</reference>
                <identifiers>*</identifiers>
            </program>

            <program name="converter_04">
                <reference>converter</reference>
                <identifiers>cliche83:60000,cliche81,cliche82</identifiers>
            </program>

            <program name="converter_07">
```

```xml
                <reference>converter</reference>
                <identifiers>cliche81,cliche83:60000,cliche82</identifiers>
            </program>

            <program pattern="converter_">
                <reference>converter</reference>
            </program>
        <programs>

    </application>

<!-- player application -->
<application name="player">
    <distribution>SINGLE_INSTANCE</distribution>
    <identifiers>cliche81,cliche83:60000</identifiers>
    <start_sequence>5</start_sequence>
    <starting_strategy>MOST_LOADED</starting_strategy>
    <starting_failure_strategy>ABORT</starting_failure_strategy>

    <programs>
        <program name="test_reader">
            <start_sequence>1</start_sequence>
            <required>true</required>
            <wait_exit>true</wait_exit>
            <expected_loading>2</expected_loading>
        </program>

        <program name="movie_player">
            <start_sequence>2</start_sequence>
            <expected_loading>13</expected_loading>
        </program>
    </programs>

</application>

<!-- web_movies application -->
<application pattern="web_">
    <start_sequence>6</start_sequence>
    <stop_sequence>2</stop_sequence>
    <starting_strategy>LESS_LOADED</starting_strategy>

    <programs>
        <program name="web_browser">
            <identifiers>*</identifiers>
            <start_sequence>1</start_sequence>
            <expected_loading>4</expected_loading>
            <running_failure_strategy>RESTART_PROCESS</running_failure_strategy>
        </program>
    </programs>

</application>
```

```xml
    <!-- disk_reader_81 application -->
    <application name="disk_reader_81">
        <start_sequence>1</start_sequence>
    </application>

</root>
```

DASHBOARD

Each Supervisor instance provides a Web Server and the **Supvisors** extension provides its own Web User Interface, as a replacement of the Supervisor one but using the same infrastructure.

Note: The information displayed in the Web User Interface is a synthesis of the information provided by all **Supvisors** instances and as perceived by the **Supvisors** instance that displays the web pages.

Important: *About the browser compliance.*

The CSS of the web pages has been written for Firefox ESR 91.3.0. The compatibility with other browsers or other versions of Firefox is unknown.

All pages are divided into 3 parts:

- the *Common Menu* on the left side ;
- a header on the top right ;
- the content itself on the lower right.

3.1 Common Menu

3.1.1 Supvisors

Generally, clicking on the 'Supvisors' title brings the *Main page* back.

There is an exception when conflicts are detected. In this case, a red light is blinking next to the **Supvisors** logo and clicking on it displays the *Conciliation page*.

The version of **Supvisors** is displayed underneath.

Fig. 1: Common Menu
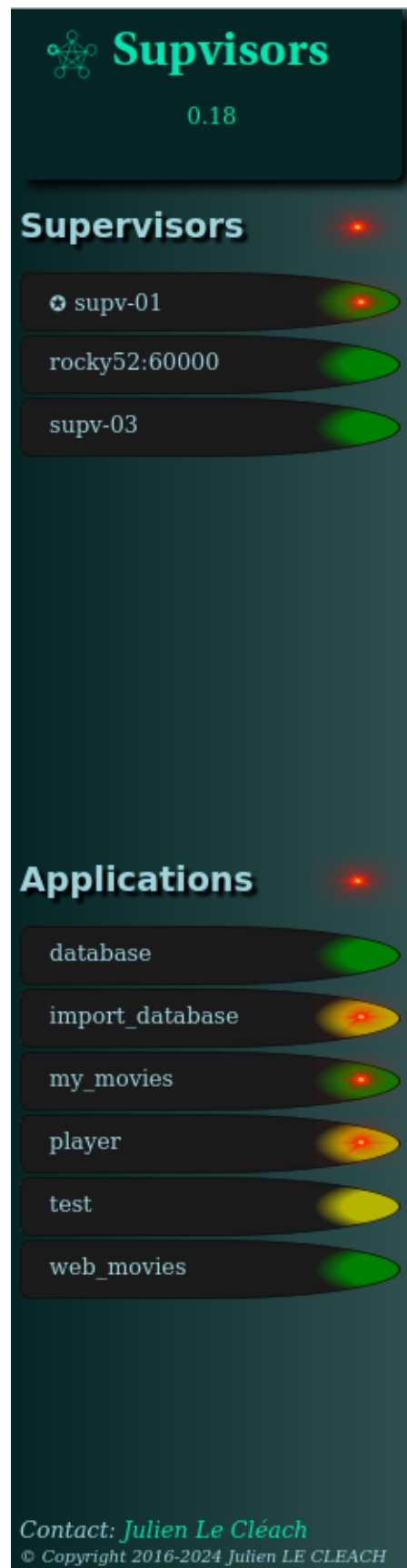
3.1.2 Supervisors

Below the **Supvisors** logo is the **Supervisors** part that lists all the **Supvisors** instances defined in the *rpcinterface extension point* of the Supervisor configuration file. The color gives the state of the **Supvisors** instance:

- grey for UNKNOWN ;

- grey-to-green gradient for CHECKING ;

- yellow for SILENT ;

- green for CHECKED and RUNNING ;

- red for ISOLATED.

The **Supvisors** instance is blinking when it is handling starting or stopping jobs.

An additional red light is displayed in the event where a process is in a crash state (FATAL or unexpectedly EXITED).

Only the hyperlinks of the CHECKED and RUNNING **Supvisors** instances are active. The browser is redirected to the *Supervisor page* of the targeted **Supvisors** instance.

The **Supvisors** instance playing the role of *Master* is pointed out with the symbol.

3.1.3 Applications

Below is the **Applications** part that lists all the *Managed* applications defined through the group sections of the Supervisor configuration file and also declared in the **Supvisors** *Supvisors' Rules File*. The color gives the state of the Application, as seen by the **Supvisors** instance that is displaying this page:

- yellow for STOPPED ;

- yellow-to-green gradient for STARTING ;

- green-to-yellow gradient for STOPPING ;

- green for RUNNING.

The application is blinking when it is part of the starting or stopping jobs managed by the local **Supvisors** instance.

An additional red light is displayed in the event where a failure has been raised on the application.

All hyperlinks are active. The browser is redirected to the corresponding *Application page* on the local Web Server.

3.1.4 Copyright

The bottom part of the menu contains a contact link to the **Supvisors** owner and maintainer and a copyright information.

3.2 Common footer

The bottom part of all pages displays two information areas:

- the acknowledgement area, used to print the result of the actions requested from the buttons of the Web UI ;

- the time when the page has been generated, and the **Supvisors** instance that provided the page.

Depending on the result, the acknowledgement area may have a different background color:

- grey by default, when no action is pending ;

- blue for a successful result ;

- amber when an action could not be performed but when the result is already as expected (e.g. a process is already started) ;

- amber too as an acknowledgement of an action having a major impact (e.g. a shutdown or a restart) ;

- red in the event of an error (e.g. start / stop failed).

3.3 Main Page

The Main Page shows a synoptic of the **Supvisors** status.
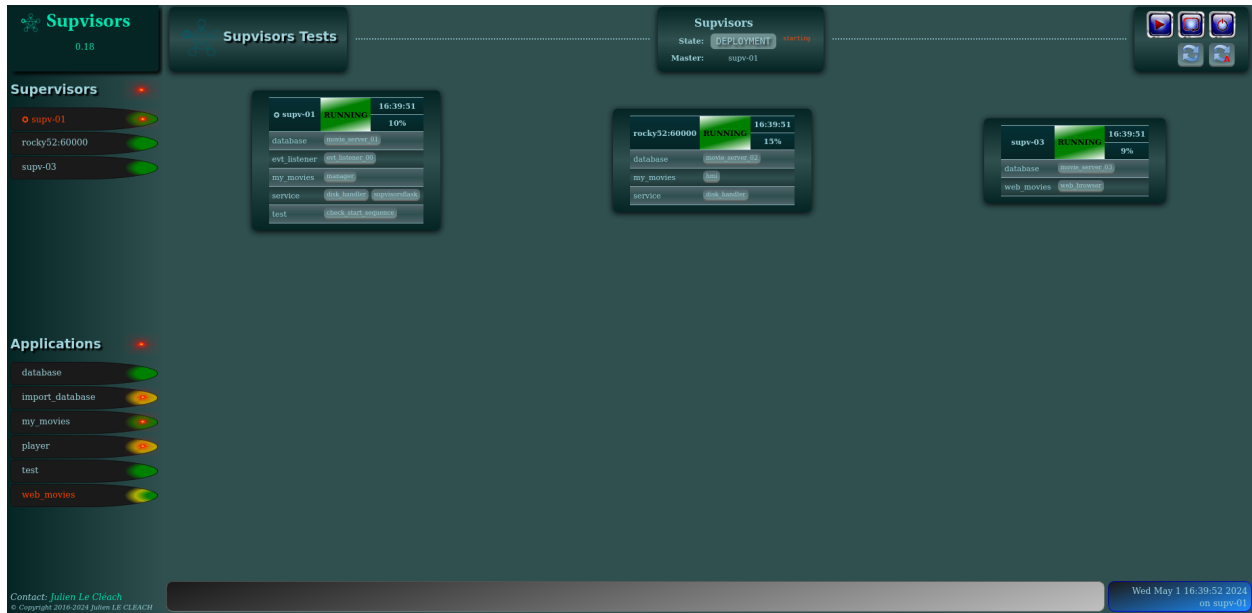
Fig. 2: Supvisors Main page

3.3.1 Main Page Header

A first card on the left of the header may be displayed with the user software name and icon, if those are set in the **Supvisors** section of the Supervisor configuration file.

The next card provides the **Supvisors** state and is displayed at the center of the header:

OFF

> This is the **Supvisors** starting phase. It is very transient and corresponds to the phase between the moment when Supervisor is started and the moment when it sends the running event to its listeners.

INITIALIZATION

> In this state, **Supvisors** waits for all **Supvisors** instances to connect themselves. Refer to the *Synchronizing Supvisors instances* section for more details.

> In this state, the **Supvisors** *XML-RPC API* is restricted so that only version, master and **Supvisors** instance information are available.

DISTRIBUTION

In this state, **Supvisors** is automatically starting applications. Refer to the *Starting strategy* section for more details.

The whole *Status* part and the *Supvisors Control* part of the **Supvisors** *XML-RPC API* are available from this state.

OPERATION

In this state, **Supvisors** is mainly:

- listening to Supervisor events ;
- publishing the events on its *Event interface* ;
- checking the activity of all remote **Supvisors** instances ;
- detecting eventual multiple running instances of the same program ;
- providing statistics to its Dashboard.

The whole **Supvisors** *XML-RPC API* is available in this state.

CONCILIATION

This state is reached when **Supvisors** has detected multiple running instances of the same program. **Supvisors** is either solving conflicts itself or waiting for the user to do it. Refer to the *Conciliation* section for more details.

The **Supvisors** *XML-RPC API* is restricted in this state. It is possible to stop applications and processes but the start requests are rejected.

RESTARTING

Supvisors is stopping all processes before commanding its own restart, i.e. the restart of all **Supvisors** instances including a restart of their related Supervisor. Refer to the *Stopping strategy* section for more details.

The **Supvisors** *XML-RPC API* is NOT available in this state.

SHUTTING_DOWN

Supvisors is stopping all processes before commanding its own shutdown, i.e. the shutdown of all **Supvisors** instances including a restart of their related Supervisor. Refer to the *Stopping strategy* section for more details.

The **Supvisors** *XML-RPC API* is NOT available in this state.

FINAL

This is the final state of **Supvisors**, in which it remains inactive and waits for the Supervisor stopping event. This state is very transient too.

The **Supvisors** *XML-RPC API* is NOT available in this state.

The **Supvisors** modes are displayed alongside the state if activated:

starting

This mode is visible and blinking when the Starter of any of the **Supvisors** instances has jobs in progress.

stopping

This mode is visible and blinking when the Stopper of any of the **Supvisors** instances has jobs in progress.

On the right side, 5 buttons are available:

- ends the **Supvisors** synchronization phase (only when **Supvisors** is in INITIALIZATION state and USER is set in the `synchro_options` option) ;

- restarts **Supvisors** through all **Supvisors** instances ;

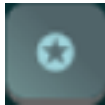- shuts down **Supvisors** through all **Supvisors** instances ;

- refreshes the current page ;

- refreshes the current page and sets a periodic 5s refresh to the page.

3.3.2 Main Page Contents

For every **Supvisors** instances, a card is displayed in the contents of the **Supvisors** Main Page. Each box contains:

- (on condition) a   star button allowing the user to ends the **Supvisors** synchronization phase and forcing the corresponding **Supvisors** instance as *Master* (only when **Supvisors** is in INITIALIZATION state and USER is set in the `synchro_options` option);

- the **Supvisors** instance nick name, as a hyperlink to the corresponding *Supervisor Page* if the **Supvisors** instance is in the CHECKED or the RUNNING state ;

- the **Supvisors** instance state, colored with the same rules used in the *Common Menu* ;

- the **Supvisors** instance process loading ;

- the list of all processes that are running in this **Supvisors** instance, whatever they belong to a *Managed* application or not.

3.4 Conciliation Page

If the page is refreshed when **Supvisors** is in CONCILIATION state, a red light is blinking next to the 'Supvisors' title in the top left of the *Common Menu*.

This situation is unlikely to happen if the `conciliation_strategy` chosen in the *rpcinterface extension point* of the Supervisor configuration file is different from USER, as the other values will trigger an immediate and automatic conciliation of the conflicts.

The Conciliation Page can be reached by clicking on the 'Supvisors' title when the red blinking light is displayed.

Fig. 3: Supvisors Main page with USER sync

Fig. 4: Supvisors Conciliation page

3.4.1 Conciliation Page Header

The header of the Conciliation Page has exactly the same contents as the header of the *Main page*.

3.4.2 Conciliation Page Contents

On the right side of the page, the list of process conflicts is displayed into a table. A process conflict is raised when the same program is running in multiple **Supvisors** instances.

So the table lists, for each conflict:

- the name of the program incriminated ;
- the list of **Supvisors** instances where it is running ;
- the uptime of the corresponding process in each **Supvisors** instance ;
- for each process, a list of actions helping to the solving of this conflict:
    - Stop the process ;
    - Keep this process (and stop all others) ;
- for each process, a list of automatic strategies (refer to *Conciliation*) helping to the solving of this conflict.

The left side of the page contains a simple box that enables the user to perform a global conciliation on all conflicts, using one of the automatic strategies proposed by **Supvisors**.

3.5 Supervisor Page

The *Supervisor* Page of **Supvisors** is the page that most closely resembles the legacy Supervisor page, hence its name, although it is a bit less "sparse" than the web page provided by Supervisor. It shows the status of the **Supvisors** instance, as seen by the **Supvisors** instance itself as this page is always re-directed accordingly. It also enables the user to command the processes declared in this **Supvisors** instance and provides statistics that may be useful at software integration time.

3.5.1 Supervisor Page Header

A first card on the left of the header may be displayed with the user software name and icon, if those are set in the **Supvisors** section of the Supervisor configuration file.

The status of the **Supvisors** instance is then displayed:

- the **Supvisors** instance deduced name, marked with the sign if it is the *Master* ;
- the current loading of the processes running in this **Supvisors** instance ;
- the **Supvisors** instance state and modes (discovery, starting, stopping).

Note: The discovery mode is displayed when the **Supvisors** multicast interface is activated.

Note: The **Supvisors** instance modes are visible and blinking when the `Starter` or the `Stopper` of the considered **Supvisors** instance has jobs in progress. It doesn't mean that a process is starting or stopping in the local Supervisor. It

---

means that the **Supvisors** instance is managing a start or a stop sequence, which could lead to processes being started or stopped on any other Supervisor instance managed by **Supvisors**.

The 'Statistics View' card enables the user to choose the information presented on this page. By default, the *Processes Section* is displayed. The other choice is the *Host Section*. The *Host* button is named after the name of the node hosting the **Supvisors** instance. The periods can be updated in the *rpcinterface extension point* of the Supervisor configuration file.

Next to it, the 'Statistics Period' box enables the user to choose the period used for the statistics of this page. The periods can be updated in the *rpcinterface extension point* of the Supervisor configuration file.

Note: These two boxes are not displayed if the optional module psutil is not installed or if the statistics are disabled through the `stats_enabled` option of the *rpcinterface extension point* of the Supervisor configuration file.

On the right side, 5 buttons are available:

- stops all the processes handled by Supervisor in this **Supvisors** instance ;

- restarts this **Supvisors** instance, including Supervisor ;

- shuts down this **Supvisors** instance, including Supervisor ;

- refreshes the current page ;

- refreshes the current page and sets a periodic 5s refresh to the page.

3.5.2 Processes Section

The **Processes Section** looks like the page provided by Supervisor. Indeed, it lists the programs that are configured in Supervisor, it presents their current state with an associated description and enables the user to perform some actions on them:

- Log tail (with a refresh button, click on the program name itself) ;
- Start ;
- Stop ;
- Restart ;
- Clear log ;
- Tail stdout log (auto-refreshed) ;
- Tail stderr log (auto-refreshed).

The activation of the Start, Stop and Restart buttons is depending on the process state. In addition to that, a stopped process cannot be started if the the corresponding program has been disabled.

Fig. 5: Supvisors Processes page

The activation of the Clear, Stdout and Stderr buttons is depending on the configuration of the `stdout_logfile` and `stderr_logfile` options of the Supervisor program configuration.

Supvisors shows additional information for each process, such as:

- the loading declared for the process in the rules file ;

- the CPU usage of the process during the last period (only if the process is `RUNNING`) ;

- the instant memory (Resident Set Size) occupation of the process at the last period tick (only if the process is `RUNNING`).

Note: CPU usage and memory are available only if the optional module psutil is installed and if the statistics are not disabled through the `stats_enabled` option of the *rpcinterface extension point* of the Supervisor configuration file.

Here is the color code used for process states:

- grey if the process state is `UNKNOWN` or if the process is disabled ;

- yellow if the process is `STOPPED` or expectedly `EXITED` ;

- yellow-green gradient if the process is `STARTING` or `BACKOFF` ;

- green if the process is `RUNNING` ;

- green-yellow gradient if the process is `STOPPING` ;

- red if the process is `FATAL` or unexpectedly `EXITED`.

Note: For `RUNNING` processes, the color code used is a bit different if the process has ever crashed since **Supvisors** has been started. The aim is to inform that process logs should be consulted.

'standard' `RUNNING` process	`RUNNING` process with a crash history
RUNNING	RUNNING

All processes are grouped by their application name and **Supvisors** provides expand / shrink actions per application to enable the user to show / hide blocks of processes. Global expand / shrink actions are provided too in the top left cell of the table.

Considering the application processes that are running in this **Supvisors** instance, the application line displays:

- the sum of their expected loading ;
- the sum of their CPU usage ;
- the sum of their instant memory occupation.

The following actions are also provided and apply to all application processes:

- Start (equivalent to **supervisorctl start group:***) ;
- Stop (equivalent to **supervisorctl stop group:***);
- Restart (a multicall chaining **stop group:*** and **start group:***).

Hint: These actions are an implementation of the following Supervisor unresolved issue:

- #1504 - Web interface: Add stop group Action

A click on the CPU or RAM measures shows detailed statistics about the process. This is not active on the application values. More particularly, **Supvisors** displays on the right side of the page a table showing for both CPU and Memory:

- the last measure ;
- the mean value ;
- the value of the linear regression slope (unit is percent per period) ;
- the value of the standard deviation.

A color and a sign are associated to the last value, so that:

- green and point out an increase of the value since the last measure ;
- red and point out a decrease of the value since the last measure ;
- blue and point out the stability of the value since the last measure.

Underneath, **Supvisors** shows two graphs (CPU and Memory) built from the series of measures taken from the selected process:

- the history of the values with a plain line ;
- the mean value with a dashed line and value in the top right corner ;
- the linear regression with a straight dotted line ;
- the standard deviation with a colored area around the mean value.

3.5.3 Host Section

Fig. 6: Supvisors Host page

The Host Section contains CPU, Memory and Network statistics for the considered node.

The CPU table shows statistics about the CPU on each core of the processor and about the average CPU of the processor.

The Memory table shows statistics about the amount of used (and not available) memory.

The Network table shows statistics about the receive and sent flows on each network interface.

Note: A space is left for future growth and is intended to display statistics about disk usage.

Clicking on a button associated to the resource displays detailed statistics (graph and table), similarly to the process buttons.

3.6 Application Page

The Application Page of **Supvisors**:

- shows the status of the *managed* application, as seen by the considered **Supvisors** instance ;
- enables the user to command the application and its processes ;
- provides statistics that may be useful at software integration time.
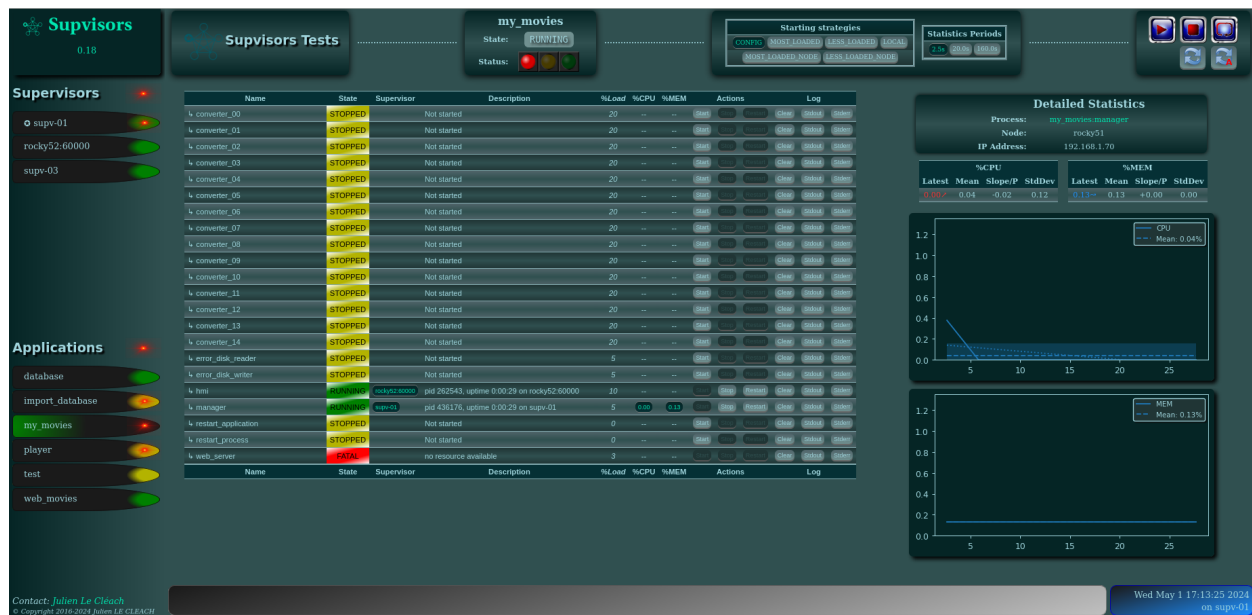
Fig. 7: Supvisors Application page

3.6.1 Application Page Header

As per the other pages, a first card on the left of the header may be displayed with the user software name and icon.

The status of the Application is displayed on the left side of the header, including:

- the name of the application ;

- the state of the application ;

- a led corresponding to the operational status of the application:

   - empty if not RUNNING ;

   - red if RUNNING and at least one major failure is detected ;

   - orange if RUNNING and at least one minor failure is detected, and no major failure ;

   - green if RUNNING and no failure is detected.

The Application operational status is evaluated against the status_formula or the required options eventually provided in the rules file.

The next part of the header is the 'Starting strategy' box that enables the user to choose the strategy to start the application programs listed below.

Strategies are detailed in *Starting strategy*.

Then follows the 'Statistics Period' box that enables the user to choose the period used for the statistics of this page. The periods can be updated in the *rpcinterface extension point* of the Supervisor configuration file.

On the right side, 4 buttons are available:

- starts the application ;

- stops the application ;

- restarts the application ;

- refreshes the current page ;

- refreshes the current page and sets a periodic 5s refresh to the page.

3.6.2 Application Page Contents

The table lists all the programs belonging to the application, and it shows:

- the 'synthetic' state of the process (refer to this note for details about the synthesis) ;

- the **Supvisors** instances where it runs, if appropriate ;

- the description (after initialization from Supervisor, the nick name of the corresponding **Supvisors** instance is added depending on the state) ;

- the loading declared for the process in the rules file ;

- the CPU usage of the process during the last period (only if the process is RUNNING) ;

- the instant memory (Resident Set Size) occupation of the process at the last period tick (only if the process is RUNNING).

Like the *Supervisor page*, the Application page enables the user to perform some actions on programs:

- Start ;

- Stop ;

- Restart ;

- Clear log ;

- Tail stdout log (auto-refreshed) ;

- Tail stderr log (auto-refreshed).

The difference is that the process is not started necessarily in the **Supvisors** instance that displays this page. Indeed, **Supvisors** uses the rules of the program (as defined in the rules file) and the starting strategy selected in the header part to choose a relevant **Supvisors** instance. If no rule is defined for the program, the Start button will be disabled.

The availability of the logs is not tested in this page.

As previously, a click on the CPU or Memory measures shows detailed statistics about the process. And unlike the *Supervisor page*, statistics information are not hidden in this page because they may have been collected on the other nodes (due to a different configuration) and thus can be made available here.

XML-RPC API

The **Supvisors** XML-RPC API is an extension of the Supervisor XML-RPC API. Detailed information can be found in the Supervisor XML-RPC API Documentation.

The `supvisors` namespace has been added to the `supervisor` XML-RPC interface.

The XML-RPC **system.listMethods** provides the list of methods supported for both Supervisor and **Supvisors**.

```
server.supvisors.getState()
```

Important: In the following, the namespec refers to the full name of the program, including the group name, as defined in Supervisor. For example: in **X11:xclock**, X11 is the name of a Supervisor group and `xclock` is the name of a Supervisor program that is referenced in the group. In some cases, it can also refer to all the programs of the group (**X11:***).

4.1 Status

class supvisors.rpcinterface.**RPCInterface**(*supvisors: Any*)

This class holds the XML-RPC extension provided by **Supvisors**.

get_api_version()

Return the version of the RPC API used by **Supvisors**.

Returns
the **Supvisors** version.

Return type
str

get_supvisors_state()

Return the state and modes of **Supvisors**. The **Supvisors** state is the FSM state and is a reflection of the **Supvisors** *Master* instance state. The **Supvisors** modes provides the identifiers of the **Supvisors** instances having starting or stopping jobs in progress.

Returns
the state and modes of **Supvisors**.

Return type
dict[str, Any]

Key	Type	Description
'fsm_stat	`int`	The **Supvisors** state, in [0;9].
'fsm_stat	`str`	The **Supvisors** state as string, in `['OFF', 'INITIALIZATION', 'DISTRIBUTION', 'OPERATION', 'CONCILIATION', 'RESTARTING', `'SHUTTING_DOWN', 'FINAL']`.
'dis- cov- ery_mod	`bool`	True if the **Supvisors** discovery mode is activated.
'start- ing_jobs'	`list(`	The list of **Supvisors** instances having starting jobs in progress.
'stop- ping_job:	`list(`	The list of **Supvisors** instances having stopping jobs in progress.

get_master_identifier()

Get the identification of the **Supvisors** instance elected as **Supvisors** *Master*.

> **Returns**
>> the identifier of the **Supvisors** *Master* instance.
>
> **Return type**
>> str

get_strategies()

Get the default strategies applied by **Supvisors**:
- auto-fencing: Supvisors instance isolation if it becomes inactive ;
- starting: used in the `DISTRIBUTION` state to start applications ;
- conciliation: used in the `CONCILIATION` state to conciliate conflicts.

> **Returns**
>> a structure containing information about the strategies applied.
>
> **Return type**
>> dict[str, Any]

Key	Type	Description
'auto- fencing'	`bool`	The application status of the auto-fencing in **Supvisors**.
'concilia- tion'	`str`	The conciliation strategy applied when **Supvisors** is in the `CONCILIATION` state.
'starting'	`str`	The starting strategy applied when **Supvisors** is in the `DISTRIBUTION` state.

get_instance_info(*instance*)

Get information about the **Supvisors** instances identified by `identifier` (Supvisors identifier, Supervisor identifier or Supvisors stereotype).

This method can return multiple results if a **Supvisors** stereotype is used as parameter.

> **Parameters**
>> **identifier** (`str`) – the identifier of the Supvisors instance where the Supervisor daemon is running.
>
> **Returns**
>> a structure containing information about the **Supvisors** instance.
>
> **Return type**
>> list[dict[str, Any]].

> **Raises**
>   **RPCError** – with code `Faults.BAD_NAME` if `identifier` is unknown to **Supvisors**.

Key	Type	Description
'identifier'	str	The **Supvisors** instance identifier (`host:http_port`).
'nick_identifi	str	The Supervisor instance identifier, or a copy of the **Supvisors** identifier if not set.
'node_name'	str	The name of the node where the **Supvisors** instance is running.
'port'	int	The HTTP port of the **Supvisors** instance.
'statecode'	int	The **Supvisors** instance state, in [0;6].
'statename'	str	The **Supvisors** instance state as string, in [`'UNKNOWN'`, `'CHECKING'`, *'CHECKED'* `, `'RUNNING'`, `'SILENT'`, `'ISOLATED'`].
'discovery_mode'	bool	True if the discovery mode is activated in the **Supvisors** instance.
'remote_sequen	int	The remote TICK counter, i.e. the number of TICK events received since the remote **Supvisors** instance is running.
'remote_mtime'	floa	The monotonic time received in the last heartbeat sent by the remote **Supvisors** instance, in seconds since the remote host started.
'remote_time'	floa	The POSIX time received in the last heartbeat sent by the remote **Supvisors** instance, in seconds and in the remote reference time.
'local_sequence	int	The local TICK counter when the latest TICK was received from the remote **Supvisors** instance.
'local_mtime'	floa	The monotonic time when the latest TICK was received from the remote **Supvisors** instance, in seconds since the local host started.
'local_time'	floa	The POSIX time when the latest TICK was received from the remote **Supvisors** instance, in seconds and in the local reference time.
'loading'	int	The sum of the expected loading of the processes running on the **Supvisors** instance, in [0;100]%.
'process_failure'	bool	True if one of the local processes has crashed or has exited unexpectedly.
'fsm_statecod	int	The **Supvisors** state as seen by the **Supvisors** instance, in [0;9].
'fsm_statena	str	The **Supvisors** state as string, in [`'OFF'`, `'INITIALIZATION'`, `'DISTRIBUTION'`, `'OPERATION'`, `'CONCILIATION'`, `'RESTARTING'`, `'SHUTTING_DOWN'`, `'FINAL'`].
'starting_jobs'	bool	True if the **Supvisors** instance has starting jobs in progress.
'stopping_jobs'	bool	True if the **Supvisors** instance has stopping jobs in progress.

get_all_instances_info()

>   Get information about all **Supvisors** instances.
>   **Returns**
>   a list of structures containing information about all **Supvisors** instances.
>   **Return type**
>   list[dict[str, Any]]

get_application_info(*application_name*)

>   Get information about an application named `application_name`.
>   **Parameters**
>   **application_name** (`str`) – the name of the application.
>   **Returns**
>   a structure containing information about the application.
>   **Return type**

dict[str, Any]

Raises

RPCError – with code: SupvisorsFaults.BAD_SUPVISORS_STATE if **Supvisors**
is still in INITIALIZATION state ; Faults.BAD_NAME if application_name is un-
known to **Supvisors**.

Key	Type	Description
'applica-tion_name'	str	The Application name.
'statecode'	int	The Application state, in [0;4].
'statename'	str	The Application state as string, in ['UNKNOWN', 'STOPPED', 'STARTING', 'STOPPING', 'RUNNING'].
'ma-jor_failure'	bool	True if at least one required process is not started.
'mi-nor_failure'	bool	True if at least one optional process could not be started.

get_all_applications_info()

Get information about all applications managed in **Supvisors**.

Returns

a list of structures containing information about all applications.

Return type

list[dict[str, Any]]

Raises

RPCError – with code SupvisorsFaults.BAD_SUPVISORS_STATE if **Supvisors**
is still in INITIALIZATION state.

get_process_info(*namespec*)

Get synthetic information about a process named namespec. It gives a synthetic status, based
on the process information coming from all running **Supvisors** instances.

Parameters

namespec (*str*) – the process namespec (name, group:name, or group:*).

Returns

a list of structures containing information about the processes.

Return type

list[dict[str, Any]]

Raises

RPCError – with code: SupvisorsFaults.BAD_SUPVISORS_STATE if **Supvisors**
is still in INITIALIZATION state ; Faults.BAD_NAME if namespec is unknown to
Supvisors.

Key	Type	Description
'application_name'	str	The Application name the process belongs to.
'process_name'	str	The Process name.
'state-code'	int	The Process state, in {0, 10, 20, 30, 40, 100, 200, 1000}.
'state-name'	str	The Process state as string, in ['STOPPED', 'STARTING', 'RUNNING', 'BACKOFF', 'STOPPING', 'EXITED', 'FATAL', 'UNKNOWN'].
'expected_exit	bool	A status telling if the process has exited expectedly.
'last_event_	float	The local monotonic time of the last event received for this process, in seconds.
'identifiers'	list(s	The deduced names of all **Supvisors** instances where the process is running.
'extra_args'	str	The extra arguments used in the command line of the process.

Hint: The 'expected_exit' status is an answer to the following Supervisor request:

- #763 - unexpected exit not easy to read in status or getProcessInfo

---

Note: If there is more than one element in the 'identifiers' list, a conflict is in progress.

---

get_all_process_info()

    Get synthetic information about all processes.

        Returns

            a list of structures containing information about the processes.

        Return type

            list[dict[str, Any]]

        Raises

            RPCError – with code SupvisorsFaults.BAD_SUPVISORS_STATE if **Supvisors** is still in INITIALIZATION state.

get_local_process_info(*namespec*)

    Get local information about a process named namespec. It is a subset of supervisor. getProcessInfo, used by **Supvisors** in INITIALIZATION state, and giving the extra arguments of the process.

        Parameters

            namespec (str) – the process namespec (name, group:name).

        Returns

            a structure containing information about the process.

        Return type

            dict[str, Any]

        Raises

            RPCError – with code Faults.BAD_NAME if namespec is unknown to **Supvisors**.

Key	Type	Description
'group'	str	The Application name the process belongs to.
'name'	str	The Process name.
'state'	int	The Process state, in {0, 10, 20, 30, 40, 100, 200, 1000}.
'start'	int	The Process start date.
'now'	float	The Process current date.
'pid'	int	The UNIX process identifier.
'startsecs'	int	The configured duration between process STARTING and RUN-NING.
'stopwait-secs'	int	The configured duration between process STOPPING and STOPPED.
'pid'	int	The UNIX process identifier.
'extra_args'	str	The extra arguments used in the command line of the process.
'disabled'	bool	A status telling if the process is disabled.

get_all_local_process_info()

Get information about all processes located on this Supvisors instance. It is a subset of `supervisor.getProcessInfo`, used by **Supvisors** in `INITIALIZATION` state, and giving the extra arguments of the process.

> **Returns**
>> a list of structures containing information about the processes.
>
> **Return type**
>> list[dict[str, Any]]

get_inner_process_info()

Get Supvisors internal information related to the processes corresponding to namespec and declared on the Supvisors instance. Mainly used for debug purpose.

> **Parameters**
>> - **identifier** (`str`) – the identifier of the Supvisors instance where the Supervisor daemon is running.
>> - **namespec** (`str`) – the process namespec (`name`, `group:name`).
>
> **Returns**
>> a structure containing information about the process.
>
> **Return type**
>> list[dict[str, Any]]
>
> **Raises**
>> - **RPCError** – with code `Faults.BAD_NAME` if `identifier` is unknown to **Supvisors**.
>> - **RPCError** – with code `Faults.BAD_NAME` if `namespec` is unknown to **Supvisors**.
>> - **RPCError** – with code `Faults.FAILED` if no handshake has been done with `identifier`.

get_all_inner_process_info()

Get Supvisors internal information related to the processes declared on the Supvisors instance. Mainly used for debug purpose.

> **Parameters**
>> **identifier** (`str`) – the identifier of the Supvisors instance where the Supervisor daemon is running.
>
> **Returns**
>> a list of structures containing information about the processes.
>
> **Return type**
>> list[dict[str, Any]]

> **Raises**
>> **RPCError** – with code `Faults.BAD_NAME` if `identifier` is unknown to **Supvisors**.

get_application_rules(*application_name*)

> Get the rules used to start / stop the application named `application_name`.
>> **Parameters**
>>> **application_name** (`str`) – the name of the application.
>>
>> **Returns**
>>> a structure containing the rules.
>>
>> **Return type**
>>> dict[str, Any]
>>
>> **Raises**
>>> **RPCError** – with code: `SupvisorsFaults.BAD_SUPVISORS_STATE` if **Supvisors** is still in `INITIALIZATION` state ; `Faults.BAD_NAME` if `application_name` is unknown to **Supvisors**.

Key	Type	Description
'application_name'	str	The Application name.
'managed'	bool	The Application managed status in **Supvisors**. When `False`, the following attributes are not provided.
'distribution'	str	The distribution rule of the Application, in `['ALL_INSTANCES', 'SINGLE_INSTANCE', 'SINGLE_NODE']`.
'identifiers'	list(	The deduced names of all **Supvisors** instances where the non-fully distributed Application processes can be started, provided only if `distribution` is not `ALL_INSTANCES`.
'start_seque	int	The Application starting rank when starting all applications, in [0;127].
'stop_seque	int	The Application stopping rank when stopping all applications, in [0;127].
'starting_strateg	str	The strategy applied when starting Application automatically, in `['CONFIG', 'LESS_LOADED', 'MOST_LOADED', 'LOCAL', 'LESS_LOADED_NODE', 'MOST_LOADED_NODE']`.
'starting_failure	str	The strategy applied when a process crashes in a starting Application, in `['ABORT', 'STOP', 'CONTINUE']`.
'running_failur	str	The strategy applied when a process crashes in a running Application, in `['CONTINUE', 'RESTART_PROCESS', 'STOP_APPLICATION', 'RESTART_APPLICATION', 'SHUTDOWN', 'RESTART']`.
'status_formula	str	The `operational_status` formula set in the Application rule.

get_process_rules(*namespec*)

> Get the rules used to start / stop the process named `namespec`.
>> **Parameters**
>>> **namespec** (`str`) – the process namespec (`name`, `group:name`, or `group:*`).
>>
>> **Returns**
>>> a list of structures containing the rules.
>>
>> **Return type**
>>> list[dict[str, Any]]
>>
>> **Raises**
>>> **RPCError** – with code: `SupvisorsFaults.BAD_SUPVISORS_STATE` if **Supvisors** is still in `INITIALIZATION` state ; `Faults.BAD_NAME` if `namespec` is unknown to **Supvisors**.

Key	Type	Description
'application_name'	str	The Application name the process belongs to.
'process_name'	str	The Process name.
'identifiers'	list(	The deduced names of all **Supvisors** instances where the process can be started.
'start_sequ(	int	The Process starting rank when starting the related application, in [0;127].
'stop_sequ(	int	The Process stopping rank when stopping the related application, in [0;127].
're-quired'	bool	The importance of the process in the application.
'wait_exit'	bool	True if **Supvisors** has to wait for the process to exit before triggering the next starting phase.
'loading'	int	The Process expected loading when RUNNING, in [0;100]%.
'running_failur	str	The strategy applied when a process crashes in a running application, in `['CONTINUE', 'RESTART_PROCESS', 'STOP_APPLICATION', 'RESTART_APPLICATION', 'SHUTDOWN', 'RESTART']`.

get_conflicts()

Get the conflicting processes among the managed applications.

Returns

a list of structures containing information about the conflicting processes.

Return type

list[dict[str, Any]]

Raises

RPCError – with code SupvisorsFaults.BAD_SUPVISORS_STATE if **Supvisors** is still in INITIALIZATION state,

The returned structure has the same format as `get_process_info(namespec)`.

4.2 Supvisors Control

class supvisors.rpcinterface.**RPCInterface**(*supvisors: Any*)

This class holds the XML-RPC extension provided by **Supvisors**.

change_log_level(*level_param*)

Change the logger level for the local **Supvisors** instance. If **Supvisors** logger is configured as AUTO, this will impact the Supervisor logger too.

Parameters

level_param (`Union[str, int]`) – the new logger level, as a string or as a value.

Returns

always True unless error.

Return type

bool

Raises

RPCError – with code Faults.INCORRECT_PARAMETERS if level_param is unknown to **Supervisor**.

conciliate(*strategy*)

>Apply the conciliation strategy only if **Supvisors** is in `CONCILIATION` state and if the default conciliation strategy is `USER` (using other strategies would trigger an automatic behavior that wouldn't give a chance to this XML-RPC).
>
>>**Parameters**
>>
>>>**strategy** (`ConciliationStrategies`) – the strategy used to conciliate, as a string or as a value.
>>
>>**Returns**
>>
>>>`True` if conciliation is triggered, `False` when the conciliation strategy is `USER`.
>>
>>**Return type**
>>
>>>bool
>>
>>**Raises**
>>
>>>`RPCError` – with code: `SupvisorsFaults.BAD_SUPVISORS_STATE` if **Supvisors** is not in state `CONCILIATION` ; `Faults.INCORRECT_PARAMETERS` if `strategy` is unknown to **Supvisors**.

restart_sequence()

>Triggers the whole starting sequence by going back to the DISTRIBUTION state.
>
>>**Parameters**
>>
>>>**wait** (`bool`) – if `True`, wait for **Supvisors** to reach the OPERATION state.
>>
>>**Returns**
>>
>>>always `True` unless error.
>>
>>**Return type**
>>
>>>bool
>>
>>**Raises**
>>
>>>`RPCError` – with code `SupvisorsFaults.BAD_SUPVISORS_STATE` if **Supvisors** is not in `OPERATION` state.

restart()

>Stops all applications and restart **Supvisors** through all Supervisor daemons.
>
>>**Returns**
>>
>>>always `True` unless error.
>>
>>**Return type**
>>
>>>bool
>>
>>**Raises**
>>
>>>`RPCError` – with code `` `SupvisorsFaults.BAD_SUPVISORS_STATE `` if **Supvisors** is still in state `INITIALIZATION` or has no Master instance to perform the request.

shutdown()

>Stops all applications and shut down **Supvisors** through all Supervisor daemons.
>
>>**Returns**
>>
>>>always `True` unless error.
>>
>>**Return type**
>>
>>>bool
>>
>>**Raises**
>>
>>>`RPCError` – with code `` `SupvisorsFaults.BAD_SUPVISORS_STATE `` if **Supvisors** is still in state `INITIALIZATION` or has no Master instance to perform the request.

4.3 Supvisors Statistics Status and Control

class supvisors.rpcinterface.**RPCInterface**(*supvisors: Any*)

This class holds the XML-RPC extension provided by **Supvisors**.

get_statistics_status()

Get information about the statistics collection status in **Supvisors**:
- host_stats: True if the host statistics are collected ;
- process_stats: True if the process statistics are collected ;
- collecting_period: the minimum interval between 2 samples of the same statistics type.

Returns

a structure containing information about the statistics collected.

Return type

dict[str, Any]

Key	Type	Description
'host_stats'	bool	The status of Host Statistics collection in **Supvisors**.
'process_stats'	bool	The status of Process Statistics collection in **Supvisors**.
'collect-ing_period'	float	The minimum interval between 2 samples of the same statistics type.

enable_host_statistics(*enable_host*)

Override the host statistics option for the local **Supvisors** instance.

Parameters

enable_host (*bool*) – if True/False and psutil installed, enable/disable host statistics collection.

Returns

always True unless error.

Return type

bool

Raises

RPCError – with code SupvisorsFaults.NOT_INSTALLED if psutil is not installed.

enable_process_statistics(*enable_process*)

Override the process statistics option for the local **Supvisors** instance.

Parameters

enable_process (*bool*) – if True/False and psutil installed, enable/disable process statistics collection.

Returns

always True unless error.

Return type

bool

Raises

RPCError – with code SupvisorsFaults.NOT_INSTALLED if psutil is not installed.

update_collecting_period(*collecting_period*)

Override the statistics period option for the local **Supvisors** instance.

Parameters
> collecting_period (*float*) – the minimum interval between 2 samples of the same statistics type.

Returns
> always True unless error.

Return type
> bool

Raises
> RPCError – with code SupvisorsFaults.NOT_INSTALLED if psutil is not installed.

4.4 Application Control

class supvisors.rpcinterface.**RPCInterface**(*supvisors: Any*)
> This class holds the XML-RPC extension provided by **Supvisors**.

> start_application(*strategy*, *application_name*, *wait=True*)
>> Start the *Managed* application named application_name iaw the strategy and the rules file. To start *Unmanaged* applications, use supervisor.start('group:*').
>>
>> **Parameters**
>> - strategy (*StartingStrategies*) – the strategy used to choose a **Supvisors** instance, as a string or as a value.
>> - application_name (*str*) – the name of the application.
>> - wait (*bool*) – if True, wait for the application to be fully started before returning.
>>
>> **Returns**
>>> always True unless error or nothing to start.
>>
>> **Return type**
>>> bool
>>
>> **Raises**
>>> RPCError – with code: SupvisorsFaults.BAD_SUPVISORS_STATE if **Supvisors** is not in state OPERATION ; Faults.INCORRECT_PARAMETERS if strategy is unknown to **Supvisors** ; Faults.BAD_NAME if application_name is unknown to **Supvisors** ; SupvisorsFaults.NOT_MANAGED if the application is not *Managed* in **Supvisors** ; Faults.ALREADY_STARTED if the application is STARTING, STOPPING or RUNNING ; Faults.ABNORMAL_TERMINATION if the internal start request failed ; Faults.NOT_RUNNING if application_name could not be started.

> stop_application(*application_name*, *wait=True*)
>> Stop the *Managed* application named application_name. To stop *Unmanaged* applications, use supervisor.stop('group:*').
>>
>> **Parameters**
>> - application_name (*str*) – the name of the application.
>> - wait (*bool*) – if True, wait for the application to be fully stopped.
>>
>> **Returns**
>>> always True unless error.
>>
>> **Return type**
>>> bool
>>
>> **Raises**
>>> RPCError – with code: SupvisorsFaults.BAD_SUPVISORS_STATE if **Supvisors** is not in state OPERATION or CONCILIATION ; Faults.BAD_NAME if application_name is unknown to **Supvisors** ;

SupvisorsFaults.NOT_MANAGED if the application is not *Managed* in **Supvisors** ; Faults.NOT_RUNNING if application_name is already stopped ; Faults.STILL_RUNNING if application_name could not be stopped.

restart_application(*strategy*, *application_name*, *wait=True*)

Restart the application named application_name iaw the strategy and the rules file. To restart *Unmanaged* applications, use supervisor.stop('group:*'), then supervisor.start('group:*').

> **Parameters**
> - **strategy** (*StartingStrategies*) – the strategy used to choose a **Supvisors** instance, as a string or as a value.
> - **application_name** (*str*) – the name of the application.
> - **wait** (*bool*) – if True, wait for the application to be fully restarted.
>
> **Returns**
> always True unless error.
>
> **Return type**
> bool
>
> **Raises**
> **RPCError** – with code SupvisorsFaults.BAD_SUPVISORS_STATE if **Supvisors** is not in state OPERATION ; Faults.INCORRECT_PARAMETERS if strategy is unknown to **Supvisors** ; Faults.BAD_NAME if application_name is unknown to **Supvisors** ; SupvisorsFaults.NOT_MANAGED if the application is not *Managed* in **Supvisors** ; Faults.ABNORMAL_TERMINATION if the internal restart request failed ; Faults.NOT_RUNNING if *application_name*` could not be restarted.

test_start_application(*strategy*, *application_name*)

Return a distribution prediction for a start of the *Managed* application named application_name iaw the strategy and the rules file.

> **Parameters**
> - **strategy** (*StartingStrategies*) – the strategy used to choose a **Supvisors** instance, as a string or as a value.
> - **application_name** (*str*) – the name of the application.
>
> **Returns**
> a list of structures with the predicted distribution of the application processes.
>
> **Return type**
> list[dict[str, Any]]
>
> **Raises**
> **RPCError** – with code: SupvisorsFaults.BAD_SUPVISORS_STATE if **Supvisors** is not in state OPERATION ; Faults.INCORRECT_PARAMETERS if strategy is unknown to **Supvisors** ; Faults.BAD_NAME if application_name is unknown to **Supvisors** ; SupvisorsFaults.NOT_MANAGED if the application is not *Managed* in **Supvisors** ; Faults.ALREADY_STARTED if the application is STARTING, STOPPING or RUNNING.

4.5 Process Control

class supvisors.rpcinterface.**RPCInterface**(*supvisors: Any*)

This class holds the XML-RPC extension provided by **Supvisors**.

start_args(*namespec*, *extra_args=''*, *wait=True*)

Start the process named `namespec` on the local Supvisors instance. The behaviour is different from `supervisor.startProcess` as it sets the process state to `FATAL` instead of throwing an exception to the RPC client. This RPC makes it also possible to pass extra arguments to the program command line.

> **Parameters**
> - **namespec** (`str`) – the process namespec.
> - **extra_args** (`str`) – the extra arguments to be passed to the command line of the program.
> - **wait** (`bool`) – if `True`, wait for the process to be fully started.
>
> **Returns**
> always `True` unless error.
>
> **Return type**
> bool
>
> **Raises**
> **RPCError** – with code: `Faults.BAD_NAME` if `namespec` is unknown to Supvisors or Supervisor ; `SupvisorsFaults.DISABLED` if process is *disabled* ; `Faults.ALREADY_STARTED` if process is `RUNNING` ; `Faults.ABNORMAL_TERMINATION` if process could not be started.

start_process(*strategy*, *namespec*, *extra_args=''*, *wait=True*)

Start a process named `namespec` iaw the strategy and the rules file. WARN: the 'wait_exit' rule is not considered here.

> **Parameters**
> - **strategy** (`StartingStrategies`) – the strategy used to choose a **Supvisors** instance, as a string or as a value.
> - **namespec** (`str`) – the process namespec (name,``group:name``, or `group:*`).
> - **extra_args** (`str`) – the optional extra arguments to be passed to command line.
> - **wait** (`bool`) – if `True`, wait for the process to be fully started.
>
> **Returns**
> always `True` unless error.
>
> **Return type**
> bool
>
> **Raises**
> **RPCError** – with code: `SupvisorsFaults.BAD_SUPVISORS_STATE` if **Supvisors** is not in state `OPERATION` ; `Faults.INCORRECT_PARAMETERS` if `strategy` is unknown to **Supvisors** ; `Faults.BAD_NAME` if `namespec` is unknown to **Supvisors** ; `Faults.ALREADY_STARTED` if process is in a running state ; `Faults.ABNORMAL_TERMINATION` if the internal start request failed ; `Faults.NOT_RUNNING` if `namespec` could not be started.

start_any_process(*strategy*, *regex*, *extra_args=''*, *wait=True*)

Start one process among those matching the `regex` iaw the strategy and the rules file. WARN: the 'wait_exit' rule is not considered here.

> **Parameters**
> - **strategy** (`StartingStrategies`) – the strategy used to choose a

Supvisors instance, as a string or as a value.
- **regex** (*str*) – a regular expression to match process namespecs.
- **extra_args** (*str*) – the optional extra arguments to be passed to command line.
- **wait** (*bool*) – if True, wait for the process to be fully started.

Returns

the namespec of the process started unless error.

Return type

str

Raises

RPCError – with code: SupvisorsFaults.BAD_SUPVISORS_STATE if **Supvisors** is not in state OPERATION ; Faults. INCORRECT_PARAMETERS if strategy is unknown to **Supvisors** ; Faults.FAILED if no stopped process found matching regex in **Supvisors** ; Faults.ABNORMAL_TERMINATION if the internal start request failed ; Faults.NOT_RUNNING if namespec could not be started.

stop_process(*namespec*, *wait=True*)

Stop the process named namespec on the **Supvisors** instance where it is running.

Parameters
- **namespec** (*str*) – the process namespec (name, group:name, or group:*).
- **wait** (*bool*) – if True, wait for process to be fully stopped.

Returns

always True unless error.

Return type

bool

Raises

RPCError – with code: SupvisorsFaults.BAD_SUPVISORS_STATE if **Supvisors** is not in state OPERATION or CONCILIATION ; Faults.BAD_NAME if namespec is unknown to **Supvisors** ; Faults.NOT_RUNNING if namespec is already stopped ; Faults. STILL_RUNNING if namespec could not be stopped.

restart_process(*strategy*, *namespec*, *extra_args=''*, *wait=True*)

Restart the process named namespec iaw the strategy and the rules defined in the rules file. Note that the process will not necessarily start in the same **Supvisors** instance as the starting context will be re-evaluated. WARN: the 'wait_exit' rule is not considered here.

Parameters
- **strategy** (*StartingStrategies*) – the strategy used to choose a **Supvisors** instance, as a string or as a value.
- **namespec** (*str*) – the process namespec (name, group:name, or group:*).
- **extra_args** (*str*) – the extra arguments to be passed to the command line.
- **wait** (*bool*) – if True, wait for process to be fully restarted.

Returns

always True unless error.

Return type

bool

Raises

RPCError – with code: SupvisorsFaults.BAD_SUPVISORS_STATE if **Supvisors** is not in state OPERATION ; Faults. INCORRECT_PARAMETERS if strategy is unknown to **Supvisors** ;

Faults.BAD_NAME if namespec is unknown to **Supvisors** ; Faults. ABNORMAL_TERMINATION if the internal restart request failed ; Faults.NOT_RUNNING if namespec could not be restarted.

test_start_process(*strategy*, *namespec*)

Return a distribution prediction for starting the processes corresponding to the namspec iaw the strategy and the rules file.

> **Parameters**
> - **strategy** (*StartingStrategies*) – the strategy used to choose a **Supvisors** instance, as a string or as a value.
> - **namespec** (*str*) – the process namespec (name,``group:name``, or group:*).
>
> **Returns**
> a list of structures with the predicted distribution of the processes.
>
> **Return type**
> list[dict[str, Any]].
>
> **Raises**
> **RPCError** – with code: SupvisorsFaults.BAD_SUPVISORS_STATE if **Supvisors** is not in state OPERATION ; Faults. INCORRECT_PARAMETERS if strategy is unknown to **Supvisors** ; Faults.BAD_NAME if namespec is unknown to **Supvisors** ; Faults. ALREADY_STARTED if process is in a running state.

update_numprocs(*program_name*, *numprocs*, *wait=True*, *lazy=False*)

Update dynamically the numprocs of the program. Implementation of Supervisor issue #177 - Dynamic numproc change.

When the number of processes decreases:
- the processes in excess are immediately stopped if lazy is False ;
- the processes in excess are kept in Supervisor as long as they're still running if lazy is True.

> **Parameters**
> - **program_name** (*str*) – the program name, as found in the section of the Supervisor configuration files. Programs, FastCGI programs and event listeners are supported.
> - **numprocs** (*int*) – the new numprocs value (must be strictly positive).
> - **wait** (*bool*) – if True and the numprocs value decreases, wait for the processes in excess to be stopped.
> - **lazy** (*bool*) – if True, use the lazy mode when decreasing the program numprocs.
>
> **Returns**
> always True unless error.
>
> **Return type**
> bool
>
> **Raises**
> **RPCError** – with code: SupvisorsFaults.BAD_SUPVISORS_STATE if **Supvisors** is not in state OPERATION ; Faults.BAD_NAME if program_name is unknown to **Supvisors** ; Faults. INCORRECT_PARAMETERS if numprocs is not a strictly positive integer ; SupvisorsFaults.NOT_APPLICABLE if the program configuration does not support numprocs ; Faults.STILL_RUNNING if one process corresponding to program_name cannot be stopped.

Hint: This XML-RPC is the implementation of the following Supervisor request:
- #177 - Dynamic numproc change

---

enable(*program_name*, *wait=True*)

Enable the process, i.e. remove the disabled flag on the corresponding processes if set. This information is persisted on disk so that it is taken into account on Supervisor restart. Implementation of Supervisor issue #591 - New Feature: disable/enable.

> **Parameters**
> - **program_name** (`str`) – the name of the program
> - **wait** (`bool`) – if `True`, wait for the corresponding processes to be fully stopped.
>
> **Returns**
> always `True` unless error.
>
> **Return type**
> bool
>
> **Raises**
> **RPCError** – with code: `SupvisorsFaults.BAD_SUPVISORS_STATE` if **Supvisors** is not in state `OPERATION` ; `Faults.BAD_NAME` if `program_name` is unknown to **Supvisors**.

Hint: This XML-RPC is a part of the implementation of the following Supervisor request:

- #591 - New Feature: disable/enable

disable(*program_name*, *wait=True*)

Disable the program, i.e. stop the corresponding processes if necessary and prevent them to start. This information is persisted on disk so that it is taken into account on Supervisor restart. Implementation of Supervisor issue #591 - New Feature: disable/enable.

> **Parameters**
> - **program_name** (`str`) – the name of the program
> - **wait** (`bool`) – if `True`, wait for the corresponding processes to be fully stopped.
>
> **Returns**
> always `True` unless error.
>
> **Return type**
> bool
>
> **Raises**
> **RPCError** – with code: `SupvisorsFaults.BAD_SUPVISORS_STATE` if **Supvisors** is not in state `OPERATION` ; `Faults.BAD_NAME` if `program_name` is unknown to **Supvisors** ; `Faults.STILL_RUNNING` if at least one process corresponding to `program_name` cannot be stopped.

Hint: This XML-RPC is a part of the implementation of the following Supervisor request:

- #591 - New Feature: disable/enable

4.6 XML-RPC Clients

This section explains how to use the XML-RPC API from a **Python** or **JAVA** client.

4.6.1 `Python` Client

To perform an XML-RPC from a **Python** client, Supervisor provides the `getRPCInterface` function of the **supervisor.childutils** module.

The parameter requires a dictionary with the following variables set:

- `SUPERVISOR_SERVER_URL`: the url of the Supervisor HTTP server (ex: `http://localhost:60000`),

- `SUPERVISOR_USERNAME`: the user name for the HTTP authentication (may be void),

- `SUPERVISOR_PASSWORD`: the password for the HTTP authentication (may be void).

If the **Python** client has been spawned by Supervisor, the environment already contains these parameters but they are configured to communicate with the local Supervisor instance.

```
>>> import os
>>> from supervisor.childutils import getRPCInterface
>>> proxy = getRPCInterface(os.environ)
>>> proxy.supvisors.get_instance_info('cliche81')
{'identifier': 'cliche81', 'node_name': 'cliche81', 'port': 60000, 'statecode': 2,
→'statename': 'RUNNING',
'sequence_counter': 885, 'remote_time': 1645285505, 'local_time': 1645285505, 'loading':␣
→24,
'fsm_statecode': 3, 'fsm_statename': 'OPERATION', 'starting_jobs': False, 'stopping_jobs
→': False}
```

If the **Python** client has to communicate with another Supervisor instance, the parameters must be set accordingly.

The `ServerProxy` of the `xmlrpc` module can also be used.

```
>>> from xmlrpc.client import ServerProxy
>>> proxy = ServerProxy('http://cliche81:60000')
>>> proxy.supvisors.get_supvisors_state()
{'fsm_statecode': 3, 'fsm_statename': 'OPERATION', 'starting_jobs': [], 'stopping_jobs':␣
→[]}
```

4.6.2 `JAVA` Client

There is **JAVA** client *supervisord4j* referenced in the Supervisor documentation. However, it comes with the following drawbacks, taken from the `README.md` of supervisord4j:

- some XML-RPC are not implemented,

- some implemented XML-RPC are not tested,

- of course, it doesn't include the **Supvisors** XML-RPC API.

The **Supvisors** release comes with a `JAR` file including a **JAVA** client. It can be downloaded from the Supvisors releases.

The package `org.supvisors.rpc` implements all XML-RPC of all interfaces (`system`, `supervisor` and `supvisors`).

This package requires the following additional dependency:

- Apache XML-RPC.

The binary JAR of **Apache XML-RPC 3.1.3** is available in the Apache MAVEN repository.

```java
import org.supvisors.rpc.*;

// create proxy
SupervisorXmlRpcClient client = new SupervisorXmlRpcClient("10.0.0.1", 60000, "toto", "p@
→$$w0rd");

// Supervisor XML-RPC
SupervisorXmlRpc supervisor = new SupervisorXmlRpc(client);
System.out.println(supervisor.getState());

// Supvisors XML-RPC
SupvisorsXmlRpc supvisors = new SupvisorsXmlRpc(client);
System.out.println(supvisors.getSupvisorsState());
```

REST API

supvisorsflask is a **Supvisors** Flask-RESTX application that is added to the `BINDIR`. It exposes the Supervisor and **Supvisors** XML-RPC API through a REST API.

Note: An exception however: the Supervisor `system.multicall` XML-RPC has not been implemented.

5.1 Starting the *Flask-RESTX* application

The program **supvisorsflask** requires 2 main information to work:

- the URL of the Supervisor instance to address the XML-RPCs,

- the URL of the Flask web server to which the REST API will be exposed.

If **supvisorsflask** is spawned by Supervisor, it naturally gets the URL of the Supervisor instance through the `SUPERVISOR_SERVER_URL` environment variable. Otherwise, this URL must be passed using the `-u SUPERVISOR_URL` option.

Default values for `HOST` and `PORT` are the Flask default values, i.e. the application will run the web server on `http:/ /127.0.0.1:5000`.

```
[bash] > supvisorsflask --help
usage: supvisorsflask [--help] -u SUPERVISOR_URL [-h HOST] [-p PORT] [-d]

Start a Flask application to interact with Supvisors

optional arguments:
  --help                show this help message and exit
  -u SUPERVISOR_URL, --supervisor_url SUPERVISOR_URL
                        the Supervisor URL, required if supvisorsflask is not
                        spawned by Supervisor
  -h HOST, --host HOST  the Flask server IP address
  -p PORT, --port PORT  the Flask server port number
  -d, --debug           the Flask Debug mode
```

5.2 Using the REST API

The aim of the present documentation is not to be a REST API tutorial. So here follows just a few usage examples with **curl** and **python**. Of course, many other programming languages will provide an API to perform such requests.

5.2.1 `curl` commands

A first possibility is to use **curl** commands in a shell.

```
[bash] > curl -X 'GET' 'http://localhost:5000/supvisors/supvisors_state' -H 'accept:␣
↪application/json'
{"fsm_statecode": 3, "fsm_statename": "OPERATION", "starting_jobs": [], "stopping_jobs":␣
↪[]}

[bash] > curl -X 'POST' \
    'http://127.0.0.1:5000/supvisors/start_process/CONFIG/my_movies%3Aconverter_00?extra_
↪args=-x%202&wait=false' \
    -H 'accept: application/json'
true
```

Supervisor XML-RPC exceptions will return a payload including the fault message and code.

```
[bash] > curl -X 'GET' 'http://localhost:5000/supvisors/application_info/dummy' -H
↪'accept: application/json'
{"message": "BAD_NAME: application dummy unknown to Supvisors", "code": 10}

[bash] > curl -X 'POST' \
    'http://127.0.0.1:5000/supvisors/start_process/CONFIG/my_movies%3Aconverter_00?extra_
↪args=-x%202&wait=false' \
    -H 'accept: application/json'
{"message": "ALREADY_STARTED: my_movies:converter_00", "code": 60}
```

5.2.2 Python requests

Here is a possibility using the **Python** module Requests. All results are a **JSON** string.

```
>>> import json, requests
>>> res = requests.get('http://localhost:5000/supvisors/supvisors_state')
>>> print(res.text)
{"fsm_statecode": 3, "fsm_statename": "OPERATION", "starting_jobs": [], "stopping_jobs":␣
↪[]}
>>> print(json.loads(res.text))
{'fsm_statecode': 3, 'fsm_statename': 'OPERATION', 'starting_jobs': [], 'stopping_jobs':␣
↪[]}
>>> res = requests.post(f'http://localhost:5000/supvisors/start_process/LESS_LOADED/my_
↪movies%3Aconverter_01?extra_args=-x%201&wait=true')
>>> print(json.loads(res.text))
{'message': 'ABNORMAL_TERMINATION: my_movies:converter_01', 'code': 40}
```

5.3 Using the Swagger UI

An interest in using Flask-RESTX over Flask is to benefit from a documented Web UI when connecting a browser to the URL defined above.

The Web UI allows to test the REST API proposed.

POST /supvisors/start_process/{strategy}/{namespec} ^

Start a process named `namespec` iaw the strategy and the rules file. WARN: the 'wait_exit' rule is not considered here.

Parameters   Cancel

Name	Description
strategy * required string (path)	the starting strategy in {CONFIG, LESS_LOADED, MOST_LOADED, LOCAL, LESS_LOADED_NODE, MOST_LOADED_NODE} LESS_LOADED
namespec * required string (path)	the namespec of the process to start my_movies:converter_00
extra_args string (query)	the extra arguments to be passed to the command line of the program -x 2
wait boolean (query)	if true, wait until completion of the request false ⌄

Execute	Clear

Responses   Response content type   application/json ⌄

Curl

```
curl -X 'POST' \
  'http://127.0.0.1:5000/supvisors/start_process/LESS_LOADED/my_movies%3Aconverter_00?extra_args=-x%202&wait=false' \
  -H 'accept: application/json' \
  -d '' \
```

Request URL

```
http://127.0.0.1:5000/supvisors/start_process/LESS_LOADED/my_movies%3Aconverter_00?extra_args=-x%202&wait=false
```

Server response

Code	Details

200	**Response body**
	
	```true```
	    Download
	
	Response headers
	
	```content-length: 5```
	```content-type: application/json```
	```date: Sat,19 Feb 2022 11:21:34 GMT```
	```server: Werkzeug/2.0.3 Python/3.6.8```

Responses

Code	Description
200	Success

SUPERVISORCTL EXTENSION

This is an extension of the existing **supervisorctl** API. The additional commands provided by **Supvisors** are available by typing **help** at the prompt.

Important: When **supervisorctl** is used with the option -s URL, Supervisor does not provide access to the extended API. This is tracked through Supervisor #1455.

Supvisors alleviates the problem by providing the command **supvisorsctl** that works with all options. The use of **supvisorsctl** is thus preferred to avoid issues, although **supervisorctl** is suitable when used - explicitly or not - with a configuration file.

In the same vein, the implementation of Supervisor #591 has introduced a new RPCError exception code (Faults. DISABLED) that can be raised from Supervisor startProcess XML-RPC. Again, using **supervisorctl** with the option -s URL will raise an unknown result code where **supvisorsctl** will handle it properly.

```
[bash] > supvisorsctl help

default commands (type help <topic>):
=====================================
add     exit      open  reload  restart   start   tail
avail   fg        pid   remove  shutdown  status  update
clear   maintail  quit  reread  signal    stop    version


supvisors commands (type help <topic>):
=======================================
all_start            local_status          start_application
all_start_args       loglevel              start_args
application_info      master                start_process
application_rules    process_rules         start_process_args
conciliate           restart_application   stats_period
conflicts            restart_process       stats_status
disable              restart_sequence      stop_application
disable_stats        sreload               stop_process
enable               sshutdown             strategies
enable_stats         sstate                sversion
end_sync             sstatus               test_start_application
instance_status      start_any_process     test_start_process
lazy_update_numprocs start_any_process_args update_numprocs
```

6.1 Status

`sversion`

> Get the API version of **Supvisors**.

`sstate`

> Get the **Supvisors** state.

`master`

> Get the deduced name of the **Supvisors** *Master* instance.

`strategies`

> Get the strategies applied in **Supvisors**.

`instance_status`

> Get the status of all **Supvisors** instances.

`instance_status identifier`

> Get the status of the **Supvisors** instance identified by its deduced name.

`instance_status identifier1 identifier2`

> Get the status for multiple Supervisor instances identified by their deduced name.

`application_info`

> Get the status of all applications.

`application_info appli`

> Get the status of application named appli.

`application_info appli1 appli2`

> Get the status for multiple named applications.

`sstatus`

> Get the status of all processes.

`sstatus proc`

> Get the status of the process named proc.

`sstatus appli:*`

> Get the status of all processes in the application named appli.

`sstatus proc1 proc2`

> Get the status for multiple named processes.

`local_status`

> Get the local status (subset of Supervisor status, with extra arguments) of all processes.

`local_status proc`

> Get the local status of the process named proc.

`local_status appli:*`

> Get the local status of all processes in the application named appli.

```
local_status proc1 proc2
```

>   Get the local status for multiple named processes.

```
application_rules
```

>   Get the rules of all processes.

```
application_rules appli
```

>   Get the rules of the applications named appli.

```
application_rules appli1 appli2
```

>   Get the rules for multiple named applications.

```
application_rules
```

>   Get the rules of all applications.

```
process_rules proc
```

>   Get the rules of the process named proc.

```
process_rules appli:*
```

>   Get the rules of all processes in the application named appli.

```
process_rules proc1 proc2
```

>   Get the rules for multiple named processes.

```
conflicts
```

>   Get the **Supvisors** conflicts among the *managed* applications.

6.2 Supvisors Control

```
loglevel level
```

>   Change the level of the **Supvisors** logger.

```
conciliate strategy
```

>   Conciliate the conflicts detected by **Supvisors** if default strategy is USER and **Supvisors** is in
>   CONCILIATION` state.

```
restart_sequence
```

>   Triggers the whole **Supvisors** start sequence.

```
sreload
```

>   Restart all **Supvisors** instances.

```
sshutdown
```

>   Shutdown all **Supvisors** instances.

6.3 Statistics Control

`stats_status`

> Get the **Supvisors** statistics collection status.

`enable_stats process`

> Enable process statistics collection

`enable_stats host process`

> Enable host and process statistics collection.

`disable_stats host`

> Disable host statistics collection.

`disable_stats all`

> Disable host and process statistics collection.

`stats_period period`

> Update the host and process statistics collection period.

6.4 Application Control

From this part, a starting strategy may be required in the command lines. It can take values among { `CONFIG`, `LESS_LOADED`, `MOST_LOADED`, `LOCAL`, `LESS_LOADED_NODE`, `MOST_LOADED_NODE` }.

`start_application strategy`

> Start all managed applications with a starting strategy.

`start_application strategy appli`

> Start the managed application named appli with a starting strategy.

`start_application strategy appli1 appli2`

> Start multiple named managed applications with a starting strategy.

`stop_application`

> Stop all managed applications.

`stop_application appli`

> Stop the managed application named appli.

`stop_application appli1 appli2`

> Stop multiple named mnaged applications.

`restart_application strategy`

> Restart all managed applications with a starting strategy.

`restart_application strategy appli`

> Restart the managed application named appli with a starting strategy.

`restart_application strategy appli1 appli2`

> Restart multiple named managed applications with a starting strategy.

```
test_start_application strategy appli
```

> Return a prediction of the distribution of the managed application named appli with a starting strategy.

6.5 Process Control

```
start_process strategy
```

> Start all processes with a starting strategy.

```
start_process strategy proc
```

> Start the process named proc with a starting strategy.

```
start_process strategy proc1 proc2
```

> Start multiple named processes with a starting strategy.

```
start_any_process strategy regex
```

> Start a process whose namespec matches the regular expression and with a starting strategy.

```
start_any_process strategy regex1 regex2
```

> Start multiple processes whose namespec matches the regular expressions and with a starting strategy.

```
start_args proc arg_list
```

> Start the process named proc in the local **Supvisors** instance and with the additional arguments arg_list passed to the command line.

```
start_process_args strategy proc arg_list
```

> Start the process named proc with a starting strategy and with the additional arguments arg_list passed to the command line.

```
start_any_process_args strategy regex arg_list
```

> Start a process whose namespec matches the regular expression, using a starting strategy and additional arguments arg_list passed to the command line.

```
all_start proc
```

> Start the process named proc on all RUNNING **Supvisors** instances.

```
all_start_args proc arg_list
```

> Start the process named proc on all RUNNING **Supvisors** instances and with the additional arguments arg_list passed to the command line.

```
stop_process
```

> Stop all processes on all addresses.

```
stop_process proc
```

> Stop the process named proc.

```
stop_process proc1 proc2
```

> Stop multiple named processes.

```
restart_process strategy
```

> Restart all processes with a starting strategy.

```
restart_process strategy proc
```

Restart the process named proc with a starting strategy.

`restart_process strategy proc1 proc2`

Restart multiple named process with a starting strategy.

`test_start_process strategy proc`

Return a prediction of the distribution of the process named proc with a starting strategy.

`test_start_process strategy appli:*`

Return a prediction of the distribution of all appli application processes with a starting strategy.

`update_numprocs program_name numprocs`

Increase or decrease dynamically the program numprocs (including FastCGI programs and Event listeners), stopping the obsolete programs immediately.

`lazy_update_numprocs program_name numprocs`

Increase or decrease dynamically the program numprocs (including FastCGI programs and Event listeners), giving a chance to the obsolete programs to end by themselves.

`enable program_name`

Enable the processes corresponding to the program.

`disable program_name`

Stop and disable the processes corresponding to the program.

EVENT INTERFACE

7.1 Available Protocols

The **Supvisors** Event Interface can be created either using a ZeroMQ socket or using websockets.

All messages consist in a header and a body.

> **Attention:** The websockets implementation requires a **Python** version 3.7 or later.

7.2 Message header

This header is a unicode string that identifies the type of the event and that is defined as follows in the `supvisors.ttypes` module:

```python
class EventHeaders(Enum):
    """ Strings used as headers in messages between EventPublisher and Supvisors' Client.
    """
    SUPVISORS = 'supvisors'
    INSTANCE = 'instance'
    APPLICATION = 'application'
    PROCESS_EVENT = 'event'
    PROCESS_STATUS = 'process'
    HOST_STATISTICS = 'hstats'
    PROCESS_STATISTICS = 'pstats'
```

The header value is used to set the event subscriptions.

7.3 Message data

The second part of the message is a dictionary serialized in JSON. Of course, the contents depends on the message type.

7.3.1 Supvisors status

Key	Type	Value
'fsm_statec	`int`	The state of **Supvisors**, in [0;6].
'fsm_staten	`str`	The string state of **Supvisors**, among { `'INITIALIZATION'`, `'DISTRIBUTION'`, `'OPERATION'`, `'CONCILIATION'`, `'RESTARTING'`, `'SHUTTING_DOWN'`, `'FINAL'` }.
'discovery_mode'	`bool`	True if the **Supvisors** discovery mode is activated.
'starting_jobs'	`list(s`	The list of **Supvisors** instances having starting jobs in progress.
'stopping_jobs'	`list(s`	The list of **Supvisors** instances having stopping jobs in progress.

7.3.2 Supvisors instance status

Key	Type	Value
'identifier'	`str`	The deduced name of the **Supvisors** instance.
'node_name'	`str`	The name of the node where the **Supvisors** instance is running.
'port'	`int`	The HTTP port of the **Supvisors** instance.
'statecode'	`int`	The **Supvisors** instance state, in [0;6].
'statename'	`str`	The **Supvisors** instance state as string, among { `'UNKNOWN'`, `'CHECKING'`, *'CHECKED'*`, `'RUNNING'`, `'SILENT'`, `'FAILED'`, `'ISOLATED'` }.
'discovery_mode'	`bool`	True if the discovery mode is activated in the **Supvisors** instance.
'remote_sequence_co	`int`	The remote TICK counter, i.e. the number of TICK events received since the remote **Supvisors** instance is running.
'remote_mtime'	`floa`	The monotonic time received in the last heartbeat sent by the remote **Supvisors** instance, in seconds since the remote host started.
'remote_time'	`floa`	The POSIX time received in the last heartbeat sent by the remote **Supvisors** instance, in seconds and in the remote reference time.
'local_sequence_cou	`int`	The local TICK counter when the latest TICK was received from the remote **Supvisors** instance.
'local_mtime'	`floa`	The monotonic time when the latest TICK was received from the remote **Supvisors** instance, in seconds since the local host started.
'local_time'	`floa`	The POSIX time when the latest TICK was received from the remote **Supvisors** instance, in seconds and in the local reference time.
'loading'	`int`	The sum of the expected loading of the processes running on the node, in [0;100]%.
'process_failure'	`bool`	True if one of the local processes has crashed or has exited unexpectedly.

7.3.3 Application status

Key	Type	Value
'application_name'	str	The Application name.
'statecode'	int	The Application state, in [0;3].
'statename'	str	The Application state as string, among { `'STOPPED'`, `'STARTING'`, `'RUNNING'`, `'STOPPING'` }.
'major_failure'	bool	True if the application is running and at least one required process is not started.
'minor_failure'	bool	True if the application is running and at least one optional (not required) process is not started.

7.3.4 Process status

Key	Type	Value
'application_nan	str	The Application name.
'process_nan	str	The Process name.
'statecode'	int	The Process state, in {0, 10, 20, 30, 40, 100, 200, 1000}. A special value -1 means that the process has been deleted as a consequence of an XML-RPC `update_numprocs`.
'statename'	str	The Process state as string, among { `'STOPPED'`, `'STARTING'`, `'RUNNING'`, `'BACKOFF'`, `'STOPPING'`, `'EXITED'`, `'FATAL'`, `'UNKNOWN'` }. A special value `DELETED` means that the process has been deleted as a consequence of an XML-RPC `update_numprocs`.
'expected_e	bool	True if the exit status is expected (only when state is `'EXITED'`).
'last_eve	float	The date of the last process event received for this process, regardless of the originating **Supvisors** instance.
'identifiers'	list	The deduced names of the **Supvisors** instances where the process is running.
'extra_args'	str	The additional arguments passed to the command line of the process.

Hint: The `expected_exit` information of this event provides an answer to the following Supervisor request:

- #1150 - Why do event listeners not report the process exit status when stopped/crashed?

7.3.5 Process event

Key	Type	Value
'group'	str	The Application name.
'name'	str	The Process name.
'state'	int	The Process state, in {0, 10, 20, 30, 40, 100, 200, 1000}. A special value -1 means that the process has been deleted as a consequence of an XML-RPC `update_numprocs`.
'ex-pected'	bool	True if the exit status is expected (only when state is 100 - `EXITED`).
'now'	floa	The monotonic time of the event in the reference time of the host.
'pid'	int	The UNIX process ID (only when state is 20 - `RUNNING` or 40 - `STOPPING`).
'iden-tifier'	str	The deduced name of the **Supvisors** instance that sent the initial event.
'ex-tra_args'	str	The additional arguments passed to the command line of the process.
'dis-abled'	bool	True if the process is disabled on the **Supvisors** instance.

7.3.6 Host statistics

Key	Type	Value
'identi-fier'	str	The deduced name of the **Supvisors** instance.
'tar-get_period	float	The configured integration period.
'period'	list(float)	The start and end uptimes of the integration period, as a list of 2 values.
'cpu'	list(float)	The CPU (IRIX mode) on the node. The first element of the list is the average CPU. The following elements correspond to the CPU on every processor core.
'mem'	float	The memory occupation on the node.
'io'	dict(str, list(float)	The Process namespec.

7.3.7 Process statistics

Key	Type	Value
'namespec'	str	The Process namespec.
'identifier'	str	The deduced name of the **Supvisors** instance.
'target_period'	float	The configured integration period.
'period'	list(float)	The start and end uptimes of the integration period, as a list of 2 values.
'cpu'	float	The CPU (IRIX mode) of the process on the node.
'mem'	float	The memory occupation of the process on the node.

7.4 ZeroMQ Implementation

The ZeroMQ implementation relies on a PUB-SUB pattern provided by PyZMQ (**Python** binding of ZeroMQ).

When the `event_link` option is set to `ZMQ`, **Supvisors** binds a PUBLISH PyZMQ socket on all interfaces using the `event_port` option defined in the *rpcinterface extension point* of the Supervisor configuration file.

Supvisors publishes the events in multi-parts messages. The first part is the message header, as a unicode string. The body follows, encoded in JSON.

To receive the **Supvisors** events, the client application must connect a SUBSCRIBE PyZMQ socket to the address defined by the node name and the port number where the **Supvisors** PUBLISH PyZMQ socket is bound.

PyZMQ makes it possible to filter the messages received on the client side by subscribing to a part of them. To receive all messages, just subscribe using an empty string.

For example, the following **Python** instructions configure the PyZMQ socket so as to receive only the *Supvisors Status* and *Process Status* events:

```
socket.setsockopt(zmq.SUBSCRIBE, EventHeaders.SUPVISORS.value.encode('utf-8'))
socket.setsockopt(zmq.SUBSCRIBE, EventHeaders.PROCESS_STATUS.value.encode('utf-8'))
```

7.4.1 Python Client

Supvisors provides a **Python** implementation of the ZeroMQ client subscriber. The *SupvisorsZmqEventInterface* is designed to receive the **Supvisors** events from a **Supvisors** instance. It requires PyZMQ to be installed.

class supvisors.client.zmqsubscriber.**SupvisorsZmqEventInterface**(*zmq_context: Context*, *node_name: str*, *event_port: int*, *logger: Logger*)

> The SupvisorsZmqEventInterface connects to Supvisors and receives the events published using the PyZMQ interface.
>
> **The SupvisorsEventInterface requires:**
>
> > - an asynchronous PyZMQ context,
> > - the node name where the Supvisors instance is running and publishing its events,
> > - the event port number used by the Supvisors instance to publish its events,
> > - a logger.
>
> This event port number MUST correspond to the event_port value set in the [supvisors] section of the Supervisor configuration file.
>
> The default behaviour is to print the messages received. For any other behaviour, just specialize the methods on_xxx_status.
>
> WARN: Notifications are received in the context of the client thread.
>
> **Example:**
>
> > import zmq.asyncio from supvisors.client.zmqsubscriber import SupvisorsZmqEventInterface
> >
> > intf = SupvisorsZmqEventInterface(zmq.asyncio.Context.instance(), 'localhost', 9003, logger) intf.subscribe_all() intf.start() # ... receive notifications ... intf.stop()
>
> **on_supvisors_status**(*data*)
>
> > Receive and log the contents of the Supvisors Status message.

> **Parameters**
> > **data** – the latest Supvisors status
>
> **Returns**
> > None

on_instance_status(*data*)

> Receive and log the contents of the Supvisors Instance Status message.
>
> > **Parameters**
> > > **data** – the latest status about a given Supvisors instance
> >
> > **Returns**
> > > None

on_application_status(*data*)

> Receive and log the contents of the Application Status message.
>
> > **Parameters**
> > > **data** – the latest status about a given Application
> >
> > **Returns**
> > > None

on_process_status(*data*)

> Receive and log the contents of the Process Status message.
>
> > **Parameters**
> > > **data** – the latest status about a given Process
> >
> > **Returns**
> > > None

on_process_event(*data*)

> Receive and log the contents of the Process Event message.
>
> > **Parameters**
> > > **data** – the latest event about a given Process
> >
> > **Returns**
> > > None

on_host_statistics(*data*)

> Receive and log the contents of the Host Statistics message.
>
> > **Parameters**
> > > **data** – the latest statistics about a given host where Supvisors is running
> >
> > **Returns**
> > > None

on_process_statistics(*data*)

> Receive and log the contents of the Process Statistics message.
>
> > **Parameters**
> > > **data** – the latest statistics about a given process running in Supvisors
> >
> > **Returns**
> > > None

```
import zmq.asyncio
from supvisors.client.clientutils import create_logger
from supvisors.client.zmqsubscriber import SupvisorsZmqEventInterface

# create the subscriber thread
subscriber = SupvisorsZmqEventInterface(zmq.asyncio.Context.instance(), 'localhost',
→9003, create_logger())
# subscribe to all messages
subscriber.subscribe_all()
# start the thread
subscriber.start()
```

7.4.2 JAVA Client

A **JAVA** implementation of the ZeroMQ client subscriber is made available with each **Supvisors** release through a JAR file. This file can be downloaded from the Supvisors releases.

The *SupvisorsEventSubscriber* of the `org.supvisors.event package` is designed to receive the **Supvisors** events from the local **Supvisors** instance. A *SupvisorsEventListener* with a specialization of the methods `onXxxStatus` must be attached to the *SupvisorsEventSubscriber* instance to receive the notifications.

It requires the following additional dependencies:

  - JeroMQ.

  - Gson.

The binary JAR of **JeroMQ 0.5.2** is available in the JeroMQ MAVEN repository.

The binary JAR of **Google Gson 2.8.6** is available in the Gson MAVEN repository.

```java
import org.supvisors.event.*;

// create ZeroMQ context
Context context = ZMQ.context(1);

// create and configure the subscriber
SupvisorsEventSubscriber subscriber = new SupvisorsEventSubscriber(9003, context);
subscriber.subscribeToAll();
subscriber.setListener(new SupvisorsEventListener() {

    @Override
    public void onSupvisorsStatus(final SupvisorsStatus status) {
        System.out.println(status);
    }

    @Override
    public void onInstanceStatus(final SupvisorsInstanceInfo status) {
        System.out.println(status);
    }

    @Override
    public void onApplicationStatus(final SupvisorsApplicationInfo status) {
        System.out.println(status);
```

(continues on next page)

```java
    }

    @Override
    public void onProcessStatus(final SupvisorsProcessInfo status) {
        System.out.println(status);
    }

    @Override
    public void onProcessEvent(final SupvisorsProcessEvent event) {
        System.out.println(event);
    }

    @Override
    public void onHostStatistics(final SupvisorsHostStatistics status) {
        System.out.println(status);
    }

    @Override
    public void onProcessStatistics(final SupvisorsProcessStatistics status) {
        System.out.println(status);
    }
});

// start subscriber in thread
Thread t = new Thread(subscriber);
t.start();
```

7.5 websockets Implementation

> **Attention:** The websockets implementation requires a **Python** version 3.7 or later.

When the `event_link` option is set to `WS`, **Supvisors** creates a websockets server that binds on all interfaces using the `event_port` option defined in the *rpcinterface extension point* of the Supervisor configuration file.

Supvisors publishes the event messages as a tuple of header, as a unicode string, and body, encoded in JSON.

To receive the **Supvisors** events, the client application must create a websockets client that connects to the address defined by the node name and the port number where the **Supvisors** websockets server has bound.

Filtering the messages is performed by adding headers to the path of the URI. To receive all messages, just add `all` to the path of the URI.

For example, the following **Python** instructions configure the websockets client so as to receive only the *Supvisors Status* and *Process Status* events:

```python
import websockets

uri = 'ws://localhost:9003/supvisors/process'
async with websockets.connect(uri) as ws:
    ...
```

7.5.1 Python Client

Supvisors provides a **Python** implementation of the websockets client. The *SupvisorsWsEventInterface* is designed to receive the **Supvisors** events from a **Supvisors** instance. It requires websockets to be installed.

class supvisors.client.wssubscriber.**SupvisorsWsEventInterface**(*node_name: str*, *event_port: int*, *logger: Logger*)

> The SupvisorsWsEventInterface connects to Supvisors and receives the events published using the Websockets interface.
>
> **The SupvisorsWsEventInterface requires:**
>
> > - the node name where the Supvisors instance is running and publishing its events ;
> >
> > - the event port number used by the Supvisors instance to publish its events ;
> >
> > - a logger.
>
> **Considering the [supvisors] section of the Supervisor configuration file:**
>
> > - the event_link option MUST be set to WS ;
> >
> > - the event_port value MUST be used when creating an instance of this class.
>
> The default behaviour is to print the messages received. For any other behaviour, just specialize the methods on_xxx_status.
>
> WARN: Notifications are received in the context of the client thread.
>
> **Example:**
> > from supvisors.client.wssubscriber import SupvisorsWsEventInterface
> >
> > intf = SupvisorsWsEventInterface('localhost', 9003, logger) intf.subscribe_all() intf.start() # … receive notifications … intf.stop()
>
> **on_supvisors_status**(*data*)
>
> > Receive and log the contents of the Supvisors Status message.
> >
> > > **Parameters**
> > > > **data** – the latest Supvisors status
> > >
> > > **Returns**
> > > > None
>
> **on_instance_status**(*data*)
>
> > Receive and log the contents of the Supvisors Instance Status message.
> >
> > > **Parameters**
> > > > **data** – the latest status about a given Supvisors instance
> > >
> > > **Returns**
> > > > None
>
> **on_application_status**(*data*)
>
> > Receive and log the contents of the Application Status message.
> >
> > > **Parameters**
> > > > **data** – the latest status about a given Application
> > >
> > > **Returns**
> > > > None

on_process_status(*data*)

> Receive and log the contents of the Process Status message.

>> **Parameters**
>>> **data** – the latest status about a given Process

>> **Returns**
>>> None

on_process_event(*data*)

> Receive and log the contents of the Process Event message.

>> **Parameters**
>>> **data** – the latest event about a given Process

>> **Returns**
>>> None

on_host_statistics(*data*)

> Receive and log the contents of the Host Statistics message.

>> **Parameters**
>>> **data** – the latest statistics about a given host where Supvisors is running

>> **Returns**
>>> None

on_process_statistics(*data*)

> Receive and log the contents of the Process Statistics message.

>> **Parameters**
>>> **data** – the latest statistics about a given process running in Supvisors

>> **Returns**
>>> None

```python
from supvisors.client.clientutils import create_logger
from supvisors.client.wssubscriber import SupvisorsWsEventInterface

# create the subscriber thread
subscriber = SupvisorsWsEventInterface('localhost', 9003, create_logger())
# subscribe to all messages
subscriber.subscribe_all()
# start the thread
subscriber.start()
```

SPECIAL FEATURES

8.1 Synchronizing Supvisors instances

The overall design of **Supvisors** is to add a **Supvisors** plugin into every Supervisor instances, and to make them share the events generated by Supervisor with each other.

To that end, a communication protocol needs to be put in place place between all **Supvisors** instances. Given the objectives of **Supvisors**, a polling mechanism doesn't fit. All Supervisor events have to be processed, so an event-driven protocol is naturally considered.

8.1.1 Communication protocols

2 internal communication protocols are used in **Supvisors**.

XML-RPC publication

The main protocol implemented in **Supvisors** is based on the XML-RPC protocol provided by Supervisor. It is used to share the local events to the other **Supvisors** instances.

The XML-RPC protocol was originally discarded because it led easily to deadlocks when involving requests to multiple Supervisor instances. So a first implementation has been done based on a PyZmq PUB-SUB. It then has been replaced by a custom implementation to limit the mandatory dependencies and to have a better control over the underlying threads and sockets. In both case, the events were sent over a TCP socket and posted sequentially to the local Supervisor using a `supervisor.sendRemoteCommEvent` XML-RPC.

Finally, with a proper understanding of the limitations brought by the XML-RPC implementation and its non-thread-safe nature, the **Supvisors** design has been simplified so that the local events and requests are processed in threads dedicated to each Supervisor proxy.

The `TICK` events are sent to all **Supvisors** instances discovered of declared in the `supvisors_list` option of the `[supvisors]` section in the Supervisor configuration file, with the exception of `ISOLATED` instances. As soon as the **Supvisors** instance is `CHECKED`, all other events are shared.

UDP Multicast

The second protocol implemented in **Supvisors** is based on an **UDP Multicast**. It relies on the following options in the [supvisors] section in the Supervisor configuration file:

- multicast_group;

- multicast_interface;

- multicast_ttl.

With this protocol, the **Supvisors** instances could be unknown at start-up and will be discovered on-the-fly. The UDP Multicast group is used to exchange ticks. Upon reception of a tick coming from an unknown **Supvisors** instance, the local **Supvisors** instance adds the remote **Supvisors** instance into its internal model and opens the TCP connections with it.

Note: Although it has been considered at some point, the idea od having **Supvisors** working only in UDP Multicast, without the TCP Publish / Subscribe, has been discarded. **Supvisors** cannot afford to lose events or to receive them in an inappropriate sequence.

8.1.2 Principles of Synchronization

The INITIALIZATION state of **Supvisors** is used as a synchronization phase so that all **Supvisors** instances are mutually aware of each other.

The following options defined in the *rpcinterface extension point* of the Supervisor configuration file are particularly used for synchronizing multiple instances of Supervisor:

- supvisors_list;

- synchro_options;

- synchro_timeout;

- core_identifiers;

- auto_fence.

Common part

Once started, all **Supvisors** instances publish the events received from Supervisor, especially the TICK events that are triggered every 5 seconds.

At the beginning, all **Supvisors** instances are declared in an UNKNOWN state. When the first TICK event is received from a remote **Supvisors** instance, a hand-shake is performed between the 2 **Supvisors** instances. The local **Supvisors** instance:

- sets the remote **Supvisors** instance state to CHECKING;

- performs a supvisors.get_instance_info(local_identifier) XML-RPC to the remote **Supvisors** instance in order to know how the local **Supvisors** instance is perceived by the remote **Supvisors** instance.

At this stage, 2 possibilities:

- the local **Supvisors** instance is seen as ISOLATED by the remote instance:

  - the remote **Supvisors** instance status is then reciprocally set to ISOLATED;

- the local **Supvisors** instance is NOT seen as ISOLATED by the remote instance:

- a `supervisor.getAllProcessInfo()` XML-RPC is requested to the remote instance ;

- the processes information is loaded into the internal data structure ;

- the remote **Supvisors** instance status is set to `CHECKED`, then `RUNNING`.

What happens next will depend on the conditions selected in the `synchro_options` option.

Whatever the number of available **Supvisors** instances, **Supvisors** elects a *Master* among the active **Supvisors** instances and enters the `DISTRIBUTION` state to start automatically the applications.

By default, the **Supvisors** *Master* instance is the **Supvisors** instance having the smallest deduced name among all the active **Supvisors** instances, unless the attribute `core_identifiers` is used. In the latter case, candidates are taken from this list in priority.

Important: *About late Supvisors instances*

When a **Supvisors** instance is started while the others are already in `OPERATION`. During the hand-shake, the local **Supvisors** instance gets the *Master* identified by the remote **Supvisors**. That confirms that the local **Supvisors** instance is a late starter and thus the local **Supvisors** instance adopts this *Master* too and skips the synchronization phase.

STRICT option

When the `STRICT` option is selected, the synchronization is complete when all **Supvisors** instances declared in the `supvisors_list` option are marked as `RUNNING`. This excludes any **Supvisors** instance that has been added to **Supvisors** in discovery mode.

This option prevails over the `LIST` and `USER` options if combined with them.

LIST option

When the `LIST` option is selected, the synchronization is complete when all known **Supvisors** instances are marked as `RUNNING`. This includes the **Supvisors** instances declared in the `supvisors_list` option **AND** the **Supvisors** instances that has been added to **Supvisors** in discovery mode.

This option prevails over the `USER` options if combined with it.

TIMEOUT option

It may happen that some declared **Supvisors** instances do not publish (very late starting, no starting at all, system down, network down, etc).

When the `TIMEOUT` option is selected, each **Supvisors** instance waits for `synchro_timeout` seconds to give a chance to all other instances to publish. When this delay is exceeded, all the **Supvisors** instances that are **not** identified as `RUNNING` or `ISOLATED` are set to:

- `SILENT` if *Auto-Fencing* is **not** activated ;

- `ISOLATED` if *Auto-Fencing* is activated.

This option prevails over all other `synchro_options` options if combined with them.

`CORE` option

Another possibility is when it is predictable that some **Supvisors** instances may be started later. For example, the pool of nodes may include servers that will always be started from the very beginning and consoles that may be started only on demand.

In this case, it would be a pity to always wait for `synchro_timeout` seconds. That's why the `core_identifiers` attribute has been introduced so that the synchronization phase is considered completed when a subset of the **Supvisors** instances declared in `supvisors_list` are RUNNING.

This option prevails over `LIST` and `USER` options if combined with them.

`USER` option

This option is useful in a context where **Supvisors** is running in a system made up of many nodes that may be started on a random basis and where core **Supvisors** instances cannot be easily identified.

When the `USER` option is selected, it allows the user to put an end to the synchronization phase when the set of running **Supvisors** instances is suitable to the user.

This action can be performed through the **Supvisors** `end_sync` XML-RPC (via code, `supervisorctl` or the **Supvisors** Web UI). This XML-RPC has an optional parameter that allows the user to select the **Supvisors** *Master* instance. If not set, the default election mechanism applies.

8.2 Auto-Fencing

Auto-fencing is applied when the `auto_fence` option of the *rpcinterface extension point* is set. It takes place when one of the **Supvisors** instances is seen as inactive (crash, system power down, network failure) from the other **Supvisors** instances.

In this case, the running **Supvisors** instances disconnect the corresponding URL from their subscription socket. The **Supvisors** instance is marked as `ISOLATED` and, in accordance with the program rules defined, **Supvisors** may restart somewhere else the processes that were eventually running in that **Supvisors** instance.

If the incriminated **Supvisors** instance is restarted, the isolation doesn't prevent the new **Supvisors** instance to receive events from the other instances that have isolated it. Indeed, it has not been considered so far to filter the subscribers from the *Publish* side.

That's why the hand-shake is performed in *Synchronizing Supvisors instances*. Each newly arrived **Supvisors** instance asks to the others if it has been previously isolated before taking into account the incoming events.

In the case of a network failure, the same mechanism is of course applied on the other side. Here comes the premises of a *split-brain syndrome*, as it leads to have 2 separate and identical sets of applications.

If the network failure is fixed, both sets of **Supvisors** are still running but do not communicate between them.

> **Attention:** **Supvisors** does NOT isolate the nodes at the Operating System level, so that when the incriminated nodes become active again, it is still possible to perform network requests between all nodes, despite the **Supvisors** instances do not communicate anymore.
>
> Similarly, it is outside the scope of **Supvisors** to isolate the communication at application level. It is the user's responsibility to isolate his applications.

8.3 Extra Arguments

Supervisor users have requested the possibility to add extra arguments to the command line of a program without having to update and reload the program configuration in Supervisor.

#1023 - Pass arguments to program when starting a job?

Indeed, the applicative context is evolving at runtime and it may be quite useful to give some information to the new process (options, path, URL of a server, URL of a display, etc), especially when dealing with distributed applications.

Supvisors introduces new XML-RPCs that are capable of taking into account extra arguments that are passed to the command line before the process is started:

- `supvisors.start_args`: start a process in the local **Supvisors** instance ;
- `supvisors.start_process`: start a process using a starting strategy.

Note: The extra arguments of the program are shared by all **Supvisors** instances. Once used, they are published through a **Supvisors** internal event and are stored directly into the Supervisor internal configuration of the programs.

In other words, considering 2 **Supvisors** instances A and B, a process that is started in **Supvisors** instance A with extra arguments and configured to restart on node crash (refer to *Running Failure strategy*). if the **Supvisors** instance A crashes (or simply becomes unreachable), the process will be restarted in the **Supvisors** instance B with the same extra arguments.

Attention: A limitation however: the extra arguments are reset each time a new **Supvisors** instance connects to the other ones, either because it has started later or because it has been disconnected for a while due to a network issue.

8.4 Starting strategy

Supvisors provides a means to start a process without telling explicitly where it has to be started, and in accordance with the rules defined for this program.

8.4.1 Choosing a Supvisors instance

The following rules are applicable whatever the chosen strategy:

- the process must not be already in a *running* state in a broad sense, i.e. `RUNNING`, `STARTING` or `BACKOFF` ;
- the process must be known to the Supervisor of the targeted **Supvisors** instance ;
- the related program must be enabled in the targeted **Supvisors** instance ;
- the targeted **Supvisors** instance must be `RUNNING` ;
- the targeted **Supvisors** instance must be allowed in the `identifiers` rule of the process ;
- the *load* of the targeted node where multiple **Supvisors** instances may be running must not exceed 100% when adding the `expected_loading` of the program to be started.

The *load* of a **Supvisors** instance is defined as the sum of the `expected_loading` of each process running in this **Supvisors** instance.

The *load* of a node is defined as the sum of the loads of the **Supvisors** instances that are running on this node.

When applying the CONFIG strategy, **Supvisors** chooses the first **Supvisors** instance available in the supvisors_list.

TODO: discovery

When applying the LESS_LOADED strategy, **Supvisors** chooses the **Supvisors** instance in the supvisors_list having the lowest *load*. The aim is to distribute the process load among the available **Supvisors** instances.

When applying the MOST_LOADED strategy, **Supvisors** chooses the **Supvisors** instance in the supvisors_list having the greatest *load*. The aim is to maximize the loading of a **Supvisors** instance before starting to load another **Supvisors** instance. This strategy is more interesting when the resources are limited.

When applying the LESS_LOADED_NODE strategy, **Supvisors** chooses the **Supvisors** instance in the supvisors_list having the lowest *load* on the node having the lowest *load*.

When applying the MOST_LOADED_NODE strategy, **Supvisors** chooses the **Supvisors** instance in the supvisors_list having the greatest *load* on the node having the greatest *load*.

When applying the LOCAL strategy, **Supvisors** chooses the local **Supvisors** instance. A typical use case is to start an HCI application on a given console, while other applications / services may be distributed over other nodes.

> **Attention:** A consequence of choosing the LOCAL strategy as the default starting_strategy in the *rpcinterface extension point* is that all programs will be started on the **Supvisors** *Master* instance.

> **Note:** When a single **Supvisors** instance is running on each node, LESS_LOADED_NODE and MOST_LOADED_NODE are strictly equivalent to LESS_LOADED and MOST_LOADED.

8.4.2 Starting a process

The internal *Starter* of **Supvisors** applies the following logic to start a process:

if the process is stopped:
> choose a **Supvisors** instance for the process in accordance with the rules defined in the previous section
>
> perform a supvisors.start_args(namespec) XML-RPC to the chosen **Supvisors** instance

This single job is considered completed when:

- a RUNNING event is received and the wait_exit rule is **not** set for this process ;

- an EXITED event is received with an expected exit code and the wait_exit rule is set for this process ;

- an error is encountered (FATAL event, EXITED event with an unexpected exit code) ;

- no STARTING event has been received 2 ticks after the XML-RPC ;

- no RUNNING event has been received X+2 ticks after the XML-RPC, X corresponding to the number of ticks needed to cover the startsecs seconds of the program definition in the **Supvisors** instance where the process has been requested to start.

This principle is used for starting a single process using a supvisors.start_process XML-RPC.

> **Attention:** *About using the wait_exit rule*
>
> If the process is expected to exit and does not exit, it will block the *Starter* until **Supvisors** is restarted.

8.4.3 Starting an application

The application start sequence is re-evaluated every time a new **Supvisors** instance becomes active in **Supvisors**. Indeed, as explained above, the internal data structure is updated with the programs configured in the new Supervisor instance and this may have an impact on the application start sequence.

The start sequence corresponds to a dictionary where:

- the keys correspond to the list of `start_sequence` values defined in the program rules of the application ;
- the value associated to a key contains the list of programs having this key as `start_sequence`.

> **Hint:** The logic applied here is an answer to the following Supervisor unresolved issues:
>
> - #122 - supervisord Starts All Processes at the Same Time
> - #456 - Add the ability to set different "restart policies" on process workers

---

> **Important:** Only the *Managed* applications can have a start sequence, i.e. only those that are declared in the **Supvisors** *Supvisors' Rules File*.
>
> The programs having a `start_sequence` lower or equal to 0 are not considered in the start sequence, as they are not meant to be automatically started.

The internal *Starter* of **Supvisors** applies the following principle to start an application:

while application start sequence is not empty:
    pop the process list having the lower (strictly positive) `start_sequence`

    for each process in process list:
        apply *Starting a process*

    wait for the jobs to complete

This principle is used for starting a single application using a `supvisors.start_application` XML-RPC.

8.4.4 Starting all applications

When entering the `DISTRIBUTION` state, all **Supvisors** instances evaluate the global start sequence using the `start_sequence` rule configured for the applications and processes.

The global start sequence corresponds to a dictionary where:

- the keys correspond to the list of `start_sequence` values defined in the application rules ;

- the value associated to a key is the list of application start sequences whose applications have this key as `start_sequence`.

The **Supvisors** *Master* instance starts the applications using the global start sequence. The following pseudo-code explains the logic used:

while global start sequence is not empty:

> pop the application list having the lower (strictly positive) `start_sequence`

> for each application in application list:
> > apply *Starting an application*

> wait for the jobs to complete

Note: The applications having a `start_sequence` lower or equal to 0 are not considered, as they are not meant to be automatically started.

Important: When leaving the `DISTRIBUTION` state, it may happen that some applications are not started properly due to missing relevant **Supvisors** instances.

When a **Supvisors** instance is started later and is authorized in the **Supvisors** ensemble, **Supvisors** transitions back to the `DISTRIBUTION` state and tries to **repair** such applications. The applications are **not** restarted. Only the stopped processes are considered.

May the new **Supvisors** instance arrive during a `DISTRIBUTION` or `CONCILIATION` phase, the transition to the `DISTRIBUTION` state is deferred until the current distribution or conciliation jobs are completed. It has been chosen NOT to transition back to the `INITIALIZATION` state to avoid a new synchronization phase.

8.5 Starting Failure strategy

When an application is starting, it may happen that any of its programs cannot be started due to various reasons:

- the program command line is wrong ;

- third parties are missing ;

- none of the **Supvisors** instances defined in the `identifiers` of the program rules are started ;

- the applicable **Supvisors** instances are already too much loaded ;

- etc.

Supvisors uses the `starting_failure_strategy` option of the rules file to determine the behavior to apply when a `required` process cannot be started. Programs having the `required` set to False are not considered as their absence is minor by definition.

Possible values are:

- `ABORT`: Abort the application starting ;

- `STOP`: Stop the application ;

- `CONTINUE`: Skip the failure and continue the application starting.

8.6 Running Failure strategy

The `autorestart` option of Supervisor may be used to restart automatically a process that has crashed or has exited unexpectedly (or not). However, when the node itself crashes or becomes unreachable, the other Supervisor instances cannot do anything about that.

Supvisors uses the `running_failure_strategy` option of the rules file to warm restart a process that was running on a node that has crashed, in accordance with the default `starting_strategy` set in the *rpcinterface extension point* and with the `supvisors_list` program rules set in the *Supvisors' Rules File*.

This option can be also used to stop or restart the whole application after a process crash. Indeed, it may happen that some applications cannot survive if one of their processes is just restarted.

Possible values are:

- `CONTINUE`: Skip the failure and the application keeps running ;

- `RESTART_PROCESS`: Restart the lost process on another **Supvisors** instance ;

- `STOP_APPLICATION`: Stop the application ;

- `RESTART_APPLICATION`: Restart the application ;

- `SHUTDOWN`: Shutdown **Supvisors** (i.e. all **Supvisors** instances) ;

- `RESTART`: Restart **Supvisors** (i.e. all **Supvisors** instances).

Important: The `RESTART_PROCESS` is NOT intended to replace the Supervisor `autorestart` for the local **Supvisors** instance. Provided a program definition where `autorestart` is set to `false` in the Supervisor configuration and where the `running_failure_strategy` option is set to `RESTART_PROCESS` in the **Supvisors** rules file, if the process crashes, **Supvisors** will NOT restart the process.

Note: Given that this option is set on the program rules, program strategies within an application may be incompatible in the event of multiple failures. That's why priorities have been set on this strategy. `STOP_APPLICATION` supersedes `RESTART_APPLICATION`, which itself supersedes `RESTART_PROCESS` and finally `CONTINUE`. So if a program with the `RESTART_APPLICATION` option fails at the same time that a program of the same application with the `STOP_APPLICATION` option, only the `STOP_APPLICATION` will be applied.

When the `RESTART_PROCESS` strategy is evaluated, if the application is fully stopped - supposedly because of the failure -, **Supvisors** will promote the `RESTART_PROCESS` into `RESTART_APPLICATION`. The idea is to benefit from a full start sequence at application level rather than uncorrelated program restarts in the event of multiple failures within the same application.

Hint: The `STOP_APPLICATION` strategy provides an answer to the following Supervisor request:

- #874 - Bring down one process when other process gets killed in a group

---

Hint: The `SHUTDOWN` strategy provides an answer to the following Supervisor request:

- #712 - shutdown supervisord once one of the programs is killed

---

8.7 Stopping strategy

Supvisors provides a means to stop a process without telling explicitly where it is running.

8.7.1 Stopping a process

The internal *Stopper* of **Supvisors** applies the following logic to stop a process:

if the process is running:

> perform a `supervisor.stopProcess(namespec)` XML-RPC to the Supervisor instances where the process is running

This single job is considered completed when:

- a `STOPPED` event is received for this process ;

- an error is encountered (`FATAL` event, `EXITED` event whatever the exit code) ;

- no `STOPPING` event has been received 2 ticks after the XML-RPC ;

- no `STOPPED` event has been received X+2 ticks after the XML-RPC, X corresponding to the number of ticks needed to cover the `stopwaitsecs` seconds of the program definition in the **Supvisors** instance where the process has been requested to stop.

This principle is used for stopping a single process using a `supvisors.stop_process` XML-RPC.

8.7.2 Stopping an application

The application stop sequence is defined at the same moment than the application start sequence. It corresponds to a dictionary where:

- the keys correspond to the list of `stop_sequence` values defined in the program rules of the application ;

- the value associated to a key is the list of programs having this key as `stop_sequence`.

Note: The *Unmanaged* applications do have a stop sequence. All their programs have the default `stop_sequence` set to `0`.

Hint: The logic applied here is an answer to the following Supervisor unresolved issue:

- #520 - allow a program to wait for another to stop before being stopped?

Hint: All the programs sharing the same `stop_sequence` are stopped simultaneously, which solves some of the requests described in the following Supervisor unresolved issue:

- #723 - Restart waits for all processes to stop before starting any

The internal *Stopper* of **Supvisors** applies the following algorithm to stop an application:

while application stop sequence is not empty:

> pop the process list having the greater `stop_sequence`

> for each process in process list:
> > apply *Stopping a process*

> wait for the jobs to complete

This principle is used for stopping a single application using a `supvisors.stop_application` XML-RPC.

8.7.3 Stopping all applications

The applications are stopped when **Supvisors** is requested to restart or shut down.

When entering the `DISTRIBUTION` state, each **Supvisors** instance evaluates also the global stop sequence using the `stop_sequence` rule configured for the applications and processes.

The global stop sequence corresponds to a dictionary where:

- the keys correspond to the list of `stop_sequence` values defined in the application rules ;
- the value associated to a key is the list of application stop sequences whose applications have this key as `stop_sequence`.

Upon reception of the `supvisors.restart` or `supvisors.shutdown`, the **Supvisors** instance uses the global stop sequence to stop all the running applications in the defined order. The following pseudo-code explains the logic used:

while global stop sequence is not empty:

> pop the application list having the greater `stop_sequence`

> for each application in application list:
> > apply *Stopping an application*

> wait for the jobs to complete

8.8 Conciliation

Supvisors is designed so that there should be only one instance of the same process running on a set of nodes, although all of them may have the capability to start it.

Nevertheless, it is still likely to happen in a few cases:

- using a request to Supervisor itself (through Web UI, `supervisorctl`, XML-RPC) ;

- upon a network failure.

> **Attention:** In the event of a network failure - let's say a network cable is unplugged -, if the `auto_fence` option is not set, a **Supvisors** instance running on the isolated node will be set to `SILENT` instead of `ISOLATED` and its URL will not disconnected from the subscriber socket.
>
> Depending on the rules set, this situation may lead **Supvisors** to warm restart the processes that were running in the lost **Supvisors** instance onto other **Supvisors** instances.
>
> When the network failure is fixed, **Supvisors** will likely have to deal with a bunch of duplicated applications and processes.

When such a conflict is detected, **Supvisors** enters in the `CONCILIATION` state. Depending on the `conciliation_strategy` option set in the *rpcinterface extension point*, it applies a strategy to be rid of all duplicates:

`SENICIDE`

> When applying the `SENICIDE` strategy, **Supvisors** keeps the youngest process, i.e. the process that has been started the most recently, and stops all the others.

`INFANTICIDE`

> When applying the `INFANTICIDE` strategy, **Supvisors** keeps the oldest process and stops all the others.

`USER`

> That's the easy one. When applying the USER strategy, **Supvisors** just waits for a third party to solve the conflicts using Web UI, `supervisorctl`, XML-RPC, process signals, or any other solution.

`STOP`

> When applying the `STOP` strategy, **Supvisors** stops all conflicting processes, which may lead the corresponding applications to a degraded state.

`RESTART`

> When applying the `RESTART` strategy, **Supvisors** stops all conflicting processes and restarts a new one.

`RUNNING_FAILURE`

> When applying the `RUNNING_FAILURE` strategy, **Supvisors** stops all conflicting processes and deals with the conflict as it would deal with a running failure, depending on the strategy defined for the process. So, after the conflicting processes are all stopped, **Supvisors** may restart the process, stop the application, restart the application or do nothing at all.

Supvisors leaves the `CONCILIATION` state when all conflicts are conciliated.

FREQUENT ASKED QUESTIONS

This section deals with frequent problems that could happen when experiencing **Supvisors** for the first time.

It is assumed that Supervisor is operational without the **Supvisors** plugin.

9.1 Error: ... cannot be resolved

```
[bash] > supervisord -n
Error: supvisors.plugin:make_supvisors_rpcinterface cannot be resolved within␣
↪[rpcinterface:supvisors]
For help, use /usr/local/bin/supervisord -h
```

This error happens in a early stage of Supervisor startup, when the plugin factory is called.

Just in case, make sure that `supvisors.plugin:make_supvisors_rpcinterface` has been copied correctly. Otherwise, this is the symptom of an improper **Supvisors** installation.

Important: **Supvisors** requires a **Python** version greater than 3.6 and must be available from the **Python** interpreter used by Supervisor's **supervisord** command.

Upon any doubt, check the **Python** version and start the interpreter in a terminal to test the import of **Supvisors**:

```
[bash] > which supervisord
/usr/local/bin/supervisord

[bash] > head -1 /usr/local/bin/supervisord
#!/usr/bin/python

[bash] > /usr/bin/python --version
Python 3.9.6

[bash] > /usr/bin/python
Python 3.9.6 (default, Nov  9 2021, 13:31:27)
[GCC 8.5.0 20210514 (Red Hat 8.5.0-3)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import supvisors
>>>
```

If an `ImportError` is raised, here follow some possible causes:

9.1.1 Wrong pip program

Issue: **Supvisors** may have been installed with a **pip** command corresponding to another **Python** version.

Solution: Install **Supvisors** using the **pip** command whose version corresponds to the **Python** version used by Supervisor.

```
[bash] > /usr/bin/python --version
Python 3.9.6

[bash] > /usr/bin/pip --version
pip 20.2.4 from /usr/lib/python3.9/site-packages/pip (python 3.9)
```

9.1.2 Local Supvisors not in `PYTHONPATH`

Issue: In the case where **Supvisors** is not installed in the **Python** packages but used from a local directory, the PYTHONPATH environment variable may not include the **Supvisors** location.

Solution: Set the **Supvisors** location in the PYTHONPATH environment variable before starting Supervisor.

```
[bash] > ls -d ~/python/supvisors/supvisors/__init__.py
/home/user/python/my_packages/supvisors/__init__.py
[bash] > export PYTHONPATH=/home/user/python/my_packages:$PYTHONPATH
[bash] > supervisord
```

9.1.3 Incorrect UNIX permissions

Issue: The user cannot read the **Supvisors** files installed (via **pip** or pointed by PYTHONPATH).

Solution: Update the UNIX permissions of the **Supvisors** package so that its files can be read by the user.

```
[user bash] > ls -l /usr/local/lib/python3.9/site-packages/supvisors/__init__.py
-rw-------. 1 root root 56 Feb 28  2022 /usr/local/lib/python3.9/site-packages/supvisors/
↪__init__.py
[user bash] > su -
Password:
[root bash] > chmod -R a+r /usr/local/lib/python3.9/site-packages/supvisors
[root bash] > exit
exit
[user bash] > ls -l /usr/local/lib/python3.9/site-packages/supvisors/__init__.py
-rw-r--r--. 1 root root 56 Feb 28  2022 /usr/local/lib/python3.9/site-packages/supvisors/
↪__init__.py
[bash] > supervisord
```

9.2 Error: Could not make supvisors rpc interface

At this stage, there must be some log traces available. If the startup of Supervisor ends with the following lines, there must be an issue with the **Supvisors** configuration, and more particularly with the option supvisors_list.

```
[bash] > supervisord -n
[...]
2022-11-17 17:47:15,101 INFO RPC interface 'supervisor' initialized
[...]
Error: Could not make supvisors rpc interface
For help, use /usr/local/bin/supervisord -h
```

There are 4 main causes to that.

9.2.1 No inet_http_server

Issue: Supervisor is configured without any inet_http_server.

Solution: Configure Supervisor with a inet_http_server.

The aim of **Supvisors** is to deal with applications distributed over several hosts so it cannot work with a Supervisor configured with an unix_http_server.

Based on the the following **Supvisors** configuration including only an unix_http_server:

```
[unix_http_server]
file=/tmp/supervisor.sock

[rpcinterface:supvisors]
supervisor.rpcinterface_factory = supvisors.plugin:make_supvisors_rpcinterface
```

If Supervisor is started from the local host, the following log traces will be displayed:

```
[bash] > supervisord -n
[...]
2022-11-18 15:21:20,166 INFO RPC interface 'supervisor' initialized
2022-11-18 15:21:20,184;WARN;Traceback (most recent call last):
  File "/usr/local/lib/python3.9/site-packages/supervisor-4.2.4-py3.9.egg/supervisor/
→http.py", line 821, in make_http_servers
    inst = factory(supervisord, **d)
  File "/usr/local/lib/python3.9/site-packages/supvisors-0.15-py3.9.egg/supvisors/plugin.
→py", line 128, in make_supvisors_rpcinterface
    supervisord.supvisors = Supvisors(supervisord, **config)
  File "/usr/local/lib/python3.9/site-packages/supvisors-0.15-py3.9.egg/supvisors/
→initializer.py", line 94, in __init__
    self.supervisor_data = SupervisorData(self, supervisor)
  File "/usr/local/lib/python3.9/site-packages/supvisors-0.15-py3.9.egg/supvisors/
→supervisordata.py", line 94, in __init__
    raise ValueError(f'Supervisor MUST be configured using inet_http_server:
→{supervisord.options.configfile}')
ValueError: Supervisor MUST be configured using inet_http_server: etc/supervisord.conf

Error: Could not make supvisors rpc interface
For help, use /usr/local/bin/supervisord -h
```

9.2.2 Incorrect Host name or IP address

Issue: The option `supvisors_list` includes a host name or an IP address that is unknown to the network configuration of the local host.

Solution: Either fix the host name / IP address, or update your network configuration or remove the entry.

Based on the the following **Supvisors** configuration including an unknown host name:

```
[rpcinterface:supvisors]
supervisor.rpcinterface_factory = supvisors.plugin:make_supvisors_rpcinterface
supvisors_list = unknown_host,rocky51,rocky52
```

If Supervisor is started from the hosts `rocky51` or `rocky52`, the following log traces will be displayed:

```
[bash] > supervisord -n
[...]
2022-11-17 17:47:15,120;ERRO;get_node_names: unknown host unknown_host
2022-11-17 18:43:52,834;CRIT;Wrong Supvisors configuration (supvisors_list)
2022-11-17 18:42:24,352;WARN;Traceback (most recent call last):
  File "/usr/local/lib/python3.9/site-packages/supervisor-4.2.4-py3.9.egg/supervisor/
↪http.py", line 821, in make_http_servers
    inst = factory(supervisord, **d)
  File "/usr/local/lib/python3.9/site-packages/supvisors-0.15-py3.9.egg/supvisors/plugin.
↪py", line 128, in make_supvisors_rpcinterface
    supervisord.supvisors = Supvisors(supervisord, **config)
  File "/usr/local/lib/python3.9/site-packages/supvisors-0.15-py3.9.egg/supvisors/
↪initializer.py", line 98, in __init__
    self.supvisors_mapper.configure(self.options.supvisors_list, self.options.core_
↪identifiers)
  File "/usr/local/lib/python3.9/site-packages/supvisors-0.15-py3.9.egg/supvisors/
↪supvisorsmapper.py", line 236, in configure
    raise ValueError(message)
ValueError: could not define a Supvisors identification from "unknown_host"

Error: Could not make supvisors rpc interface
For help, use /usr/local/bin/supervisord -h
```

In the event where the host name or IP address seems legit to the user, here are a few explanations about how **Supvisors** identifies the local Supervisor instance among the `supvisors_list` elements:

- **Supvisors** extracts the `host_name` from the `<identifier>host_name:http_port:internal_port` element and stores the host name and aliases returned by the `socket.gethostbyaddr` function.

- **Supvisors** considers that the local Supervisor instance is the element whose fully-qualified domain name, as returned by the `socket.getfqdn` function, belongs to the list of host name and aliases.

From the example below, the values `rocky51.cliche.bzh`, `rocky51` and `192.168.1.65` are valid `host_name` elements to be used in `supvisors_list`.

```
>>> from socket import gethostbyaddr, getfqdn
>>> gethostbyaddr('rocky51.cliche.bzh')
('rocky51.cliche.bzh', ['rocky51'], ['192.168.1.65'])
>>> gethostbyaddr('rocky51')
('rocky51.cliche.bzh', ['rocky51'], ['192.168.1.65'])
>>> gethostbyaddr('192.168.1.65')
```

```
('rocky51.cliche.bzh', ['rocky51'], ['192.168.1.65'])
>>> getfqdn()
'rocky51.cliche.bzh'
```

9.2.3 could not find local the local Supvisors

Issue: The option `supvisors_list` does not include any host name or IP address corresponding to the local host.

Solution: Either add the local host to the list, or avoid to start Supervisor from the local host using this configuration.

Based on the the following **Supvisors** configuration including 2 host names:

```
[rpcinterface:supvisors]
supervisor.rpcinterface_factory = supvisors.plugin:make_supvisors_rpcinterface
supvisors_list = rocky52,rocky53
```

if Supervisor is started from a host that is not present in this list, the following traces will be displayed:

```
[bash] > supervisord -n
[...]
2022-11-17 18:30:33,863;INFO;SupvisorsMapper.configure: identifiers=['rocky52', 'rocky53
↪']
2022-11-17 18:30:33,863;ERRO;SupvisorsMapper.find_local_identifier: could not find local
↪the local Supvisors in supvisors_list
2022-11-17 18:44:45,571;CRIT;Wrong Supvisors configuration (supvisors_list)
2022-11-17 18:44:45,572;WARN;Traceback (most recent call last):
  File "/usr/local/lib/python3.9/site-packages/supervisor-4.2.4-py3.9.egg/supervisor/
↪http.py", line 821, in make_http_servers
    inst = factory(supervisord, **d)
  File "/usr/local/lib/python3.9/site-packages/supvisors-0.15-py3.9.egg/supvisors/plugin.
↪py", line 128, in make_supvisors_rpcinterface
    supervisord.supvisors = Supvisors(supervisord, **config)
  File "/usr/local/lib/python3.9/site-packages/supvisors-0.15-py3.9.egg/supvisors/
↪initializer.py", line 98, in __init__
    self.supvisors_mapper.configure(self.options.supvisors_list, self.options.core_
↪identifiers)
  File "/usr/local/lib/python3.9/site-packages/supvisors-0.15-py3.9.egg/supvisors/
↪supvisorsmapper.py", line 240, in configure
    self.find_local_identifier()
  File "/usr/local/lib/python3.9/site-packages/supvisors-0.15-py3.9.egg/supvisors/
↪supvisorsmapper.py", line 269, in find_local_identifier
    raise ValueError(message)
ValueError: could not find the local Supvisors in supvisors_list

Error: Could not make supvisors rpc interface
For help, use /usr/local/bin/supervisord -h
```

9.2.4 multiple candidates for the local Supvisors

Issue: This happens when multiple **Supvisors** instances have to be started on the same host. In that case, the option `supvisors_list` includes at least 2 host names or IP addresses referring to the same host and that have not been qualified using a Supervisor identification.

Solution: Use the Supervisor identification option and apply it to the `supvisors_list`.

Based on the the following **Supvisors** configuration including a host name `rocky51` and its IP address `192.168.1.70`:

```
[rpcinterface:supvisors]
supervisor.rpcinterface_factory = supvisors.plugin:make_supvisors_rpcinterface
supvisors_list = rocky51,rocky52,192.168.1.70:30000:
```

if Supervisor is started from the host `rocky51`, the following traces will be displayed:

```
[bash] > supervisord -n
[...]
2022-11-18 10:42:25,931;INFO;SupvisorsMapper.configure: identifiers=['rocky51', 'rocky52
→', '192.168.1.70:30000']
2022-11-18 10:42:25,931;ERRO;SupvisorsMapper.find_local_identifier: multiple candidates␣
→for the local Supvisors: ['rocky51', '192.168.1.70:30000']
2022-11-18 10:42:25,931;CRIT;Wrong Supvisors configuration (supvisors_list)
2022-11-18 10:42:25,940;WARN;Traceback (most recent call last):
  File "/usr/local/lib/python3.9/site-packages/supervisor-4.2.4-py3.9.egg/supervisor/
→http.py", line 821, in make_http_servers
    inst = factory(supervisord, **d)
  File "/usr/local/lib/python3.9/site-packages/supvisors-0.15-py3.9.egg/supvisors/plugin.
→py", line 128, in make_supvisors_rpcinterface
    supervisord.supvisors = Supvisors(supervisord, **config)
  File "/usr/local/lib/python3.9/site-packages/supvisors-0.15-py3.9.egg/supvisors/
→initializer.py", line 98, in __init__
    self.supvisors_mapper.configure(self.options.supvisors_list, self.options.core_
→identifiers)
  File "/usr/local/lib/python3.9/site-packages/supvisors-0.15-py3.9.egg/supvisors/
→supvisorsmapper.py", line 240, in configure
    self.find_local_identifier()
  File "/usr/local/lib/python3.9/site-packages/supvisors-0.15-py3.9.egg/supvisors/
→supvisorsmapper.py", line 269, in find_local_identifier
    raise ValueError(message)
ValueError: multiple candidates for the local Supvisors: ['rocky51', '192.168.1.70:30000
→']

Error: Could not make supvisors rpc interface
For help, use /usr/local/bin/supervisord -h
```

At the moment, a solution in **Supvisors** is to qualify the entry in `supvisors_list` by adding its Supervisor identifier. This is also the name that will be used for the Web UI.

```
[rpcinterface:supvisors]
supervisor.rpcinterface_factory = supvisors.plugin:make_supvisors_rpcinterface
supvisors_list = <supv-01>rocky51,rocky52,<supv-03>192.168.1.70:30000:
```

Then Supervisor shall be started by passing this identification to the **supervisord** program.

```
[bash] > supervisord -ni supv-01
```

9.3 Remote host `SILENT`

A remote **Supvisors** instance may be declared `SILENT`, although **supervisord** is running on the remote host.

There is likely an issue with the firewall of the hosts. By default, a firewall is configured to block almost everything. The Supervisor HTTP ports have to be explicitly allowed in the firewall configuration.

Issue: Without the **Supvisors** plugin, accessing the remote Supervisor web page using its URL is rejected.

Solution: Use HTTP ports that are allowed by the firewall or ask the UNIX administrator to enable the HTTP ports used by the Supervisor configuration.

9.3.1 Inconsistent Supvisors configuration

Issue: Accessing the remote **Supvisors** web page using its URL is accepted. Various error messages may be received.

Solution: Make sure that the `supvisors_list` is consistent for all **Supvisors** instances, in accordance with *rpcinterface extension point*.

When using a simple Supervisor / **Supvisors** configuration as follows:

```
[inet_http_server]
port=:60000

[rpcinterface:supvisors]
supervisor.rpcinterface_factory = supvisors.plugin:make_supvisors_rpcinterface
supvisors_list = rocky51,rocky52,rocky53
internal_port = 60001
```

It is assumed that **supervisord** will be started on the 3 hosts with the same configuration, i.e. with a Supervisor server available on port 60000 and with **Supvisors** internal publisher available on port 60001.

If the Supervisor configuration on rocky52 is different and declares an `inet_http_server` on port 60100, the XML-RPC from rocky51 and rocky53 towards rocky52 will fail.

A variety of different errors may be experienced depending on how wrong configuration is.

```
[bash] > supervisord -n
[...]
2022-11-18 18:16:20,428;ERRO;Context.on_tick_event: got tick from unknown␣
↪Supvisors=rocky52
[...]
[ERROR] failed to check Supvisors=rocky52
[...]
```

9.4 Empty Application menu

The Application Menu in the **Supvisors** Web UI unexpectedly contains only the application template.

In the **Supvisors** Web UI, it may happen that the Application Menu (see *Dashboard*) contains only the application template. In this case, all applications are considered *Unmanaged*.

9.4.1 Wrong `rules_files`

Issue: The `rules_files` option is not set or the targeted files are not reachable.

Solution: Make sure that the `rules_files` option is set with reachable files.

When the `rules_files` option is set and **Supvisors** does not find any corresponding file, the following log trace is displayed:

```
[bash] > supervisord -n
[...]
2022-11-18 19:22:53,201;WARN;SupvisorsOptions.to_filepaths: no rules file found
[...]
```

9.4.2 XML file not readable

Issue: The user cannot read the **Supvisors** XML rules file.

Solution: Update the UNIX permissions of the **Supvisors** XML rules file so that it can be read by the user.

When the `rules_files` option is set with a file that cannot be read, the following log trace is displayed:

```
[bash] > supervisord -n
[...]
2022-11-18 19:33:19,793;INFO;Parser: parsing rules from my_movies.xml
2022-11-18 19:33:19,797;WARN;Supvisors: cannot parse rules files: ['my_movies.xml'] -␣
→Error reading file 'my_movies.xml': failed to load external entity "my_movies.xml"
[...]
```

9.4.3 XML file invalid

Issue: The **Supvisors** XML rules file is syntactically incorrect.

Solution: Fix the **Supvisors** XML rules file syntax.

When the `rules_files` option is set with a file that is syntactically incorrect, the following log trace is displayed:

```
[bash] > supervisord -n
[...]
2022-11-18 19:26:34,713;INFO;Parser: parsing rules from my_movies.xml
2022-11-18 19:26:35,448;WARN;Supvisors: cannot parse rules files: ['my_movies.xml'] -␣
→expected '>', line 83, column 13 (my_movies.xml, line 83)
[...]
```

Hint: The XSD file `rules.xsd` provided in the **Supvisors** package can be used to validate the XML rules files.

```
[bash] > xmllint --noout --schema rules.xsd my_movies.xml
```

9.4.4 No application declared

Issue: The **Supvisors** XML rules file has been parsed correctly but still no application in the menu of the Web UI.

Solution: For the Supervisor group name considered, make sure that an application element exists in a **Supvisors** XML rules file.

So considering this group definition in Supervisor configuration:

```
[group:my_movies]
programs=program_1,program_2
```

An application element has to be included in a **Supvisors** XML rules file to make it *Managed* and displayed in the Application menu of the **Supvisors** Web UI.

```
<root>
    <application name="my_movies"/>
</root>
```

TEN

SCENARIO 1

10.1 Context

In this use case, the application is distributed over 3 nodes. The process distribution is fixed. The application logs and other data are written to a disk that is made available through a NFS mount point.

10.2 Requirements

Here are the use case requirements:

Requirement 1
    Due to the inter-processes communication scheme, the process distribution shall be fixed.

Requirement 2
    The application shall wait for the NFS mount point before it is started.

Requirement 3
    An operational status of the application shall be provided.

Requirement 4
    The user shall not be able to start an unexpected application process on any other node.

Requirement 5
    The application shall be restarted on the 3 nodes upon user request.

Requirement 6
    There shall be a non-distributed configuration for developers' use, assuming a different inter-processes communication scheme.

Requirement 7
    The non-distributed configuration shall not wait for the NFS mount point.

10.3 Supervisor configuration

There are undoubtedly many ways to skin the cat. Here follows one solution.

As an answer to REQUIREMENT 1 (Due to the inter-processes communication scheme, the process distribution shall be fixed.) and REQUIREMENT 4 (The user shall not be able to start an unexpected application process on any other node.), let's split the Supervisor configuration file into 4 parts:

- the `supervisord.conf` configuration file ;
- the program definitions and the group definition (`.ini` files) for the first node ;

- the program definitions and the group definition (`.ini` files) for the second node ;

- the program definitions and the group definition (`.ini` files) for the third node.

All programs are configured using `autostart=true`.

For packaging facility, the full configuration is available to all nodes but the `include` section of the configuration file uses the `host_node_name` so that the running configuration is actually different on all nodes.

```
[include]
files = %(host_node_name)s/*.ini
```

The resulting file tree would be as follows.

```
[bash] > tree
.
├── etc
            ├── rocky51
            │           ├── group_rocky51.ini
            │           └── programs_rocky51.ini
            ├── rocky52
            │           ├── group_rocky52.ini
            │           └── programs_rocky52.ini
            ├── rocky53
            │           ├── group_rocky53.ini
            │           └── programs_rocky53.ini
            └── supervisord.conf
```

For REQUIREMENT 6 (There shall be a non-distributed configuration for developers' use, assuming a different inter-processes communication scheme.), let's just define a group where all programs are declared. The proposal is to have 2 Supervisor configuration files, one for the distributed application and the other for the non-distributed application, the variation being just in the include section.

```
[bash] > tree
.
├── etc
            ├── rocky51
            │           ├── group_rocky51.ini
            │           └── programs_rocky51.ini
            ├── rocky52
            │           ├── group_rocky52.ini
            │           └── programs_rocky52.ini
            ├── rocky53
            │           ├── group_rocky53.ini
            │           └── programs_rocky53.ini
            ├── localhost
            │           ├── group_localhost.ini
            │           └── programs_localhost.ini
            ├── supervisord.conf -> supervisord_distributed.conf
            ├── supervisord_distributed.conf
            ├── supervisord_localhost.conf
            └── supvisors-rules.xml
```

Here is the resulting `include` sections:

```
# include section for distributed application in supervisord_distributed.conf
[include]
files = %(host_node_name)s/*.ini

# include section for non-distributed application in supervisord_localhost.conf
[include]
files = localhost/*.ini
```

About REQUIREMENT 2 (The application shall wait for the NFS mount point before it is started.), Supervisor does not provide any facility to stage the starting sequence (refer to Issue #122 - supervisord Starts All Processes at the Same Time). A workaround here would be to insert a wait loop in all the application programs (in the program command line or in the program source code). The idea of pushing this wait loop outside the Supervisor scope - just before starting **supervisord** - is excluded as it would impose this dependency on other applications eventually managed by Supervisor.

With regard to REQUIREMENT 7 (The non-distributed configuration shall not wait for the NFS mount point.), this workaround would require different program commands or parameters, so finally different program definitions from Supervisor configuration perspective.

Supervisor provides nothing for REQUIREMENT 3 (An operational status of the application shall be provided.). The user has to evaluate the operational status based on the process status provided by the Supervisor instances on the 3 nodes, either using multiple **supervisorctl** shell commands, XML-RPCs or event listeners.

To restart the whole application (REQUIREMENT 5 (The application shall be restarted on the 3 nodes upon user request.)), the user can perform **supervisorctl** shell commands or XML-RPCs on each Supervisor instance.

```
[bash] > for i in rocky51 rocky52 rocky53
... do
...    supervisorctl -s http://$i:<port> restart scenario_1:*
... done
```

Eventually, all the requirements could be met using Supervisor but it would require additional software development at application level to build an operational status, based on process information provided by Supervisor.

It would also require some additional complexity in the configuration files and in the program command lines to manage a staged starting sequence of the programs in the group and to manage the distribution of the application over different platforms.

10.4 Involving Supvisors

A solution based on **Supvisors** could use the following Supervisor configuration (same principles as the previous section):

- the supervisord_distributed.conf configuration file for the distributed application ;
- the supervisord_localhost.conf configuration file for the non-distributed application ;
- the program definitions and the group definition (.ini files) for the first node ;
- the program definitions and the group definition (.ini files) for the second node ;
- the program definitions and the group definition (.ini files) for the third node ;
- the group definition including all application programs for a local node.

All programs are now configured using autostart=false.

10.4.1 Introducing the staged start sequence

About REQUIREMENT 2, **Supvisors** manages staged starting sequences and it offers a possibility to wait for a planned exit of a process in the sequence. So let's define a program **scen1_wait_nfs_mount[_X]** per node and whose role is to exit (using an expected exit code, as defined in Supervisor program configuration) as soon as the NFS mount is available.

Satisfying REQUIREMENT 7 is just about avoiding the inclusion of the **scen1_wait_nfs_mount[_X]** programs in the Supervisor configuration file in the case of a non-distributed application. That's why the Supervisor configuration of these programs is isolated from the configuration of the other programs. That way, **Supvisors** makes it possible to avoid an impact to program definitions, scripts and source code when dealing with such a requirement.

Here follows what the include section may look like in both Supervisor configuration files.

```
# include section for distributed application in supervisord_distributed.conf (unchanged)
[include]
files = %(host_node_name)s/*.ini

# include section for non-distributed application in supervisord_localhost.conf
# the same program definitions as the distributed application are used
[include]
files = */programs_*.ini localhost/group_localhost.ini
```

10.4.2 Rules file

Now that programs are not started automatically by Supervisor, a **Supvisors** rules file is needed to define the staged starting sequence. A first naive - yet functional - approach would be to use a model for all programs to be started on the same node.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<root>
    <!-- models -->
    <model name="model_rocky51">
        <identifiers>rocky51</identifiers>
        <start_sequence>2</start_sequence>
        <required>true</required>
    </model>
    <model name="model_rocky52">
        <reference>model_rocky51</reference>
        <identifiers>rocky52</identifiers>
    </model>
    <model name="model_rocky53">
        <reference>model_rocky51</reference>
        <identifiers>rocky53</identifiers>
    </model>
    <!-- Scenario 1 Application -->
    <application name="scen1">
        <start_sequence>1</start_sequence>
        <starting_failure_strategy>CONTINUE</starting_failure_strategy>
        <programs>
            <!-- Programs on rocky51 -->
            <program name="scen1_hci">
                <reference>model_rocky51</reference>
```

```xml
                </program>
                <program name="scen1_config_manager">
                    <reference>model_rocky51</reference>
                </program>
                <program name="scen1_data_processing">
                    <reference>model_rocky51</reference>
                </program>
                <program name="scen1_external_interface">
                    <reference>model_rocky51</reference>
                </program>
                <program name="scen1_data_recorder">
                    <reference>model_rocky51</reference>
                </program>
                <program name="scen1_wait_nfs_mount_1">
                    <reference>model_rocky51</reference>
                    <start_sequence>1</start_sequence>
                    <wait_exit>true</wait_exit>
                </program>
                <!-- Programs on rocky52 -->
                <program name="scen1_sensor_acquisition_1">
                    <reference>model_rocky52</reference>
                </program>
                <program name="scen1_sensor_processing_1">
                    <reference>model_rocky52</reference>
                </program>
                <program name="scen1_wait_nfs_mount_2">
                    <reference>model_rocky52</reference>
                    <start_sequence>1</start_sequence>
                    <wait_exit>true</wait_exit>
                </program>
                <!-- Programs on rocky53 -->
                <program name="scen1_sensor_acquisition_2">
                    <reference>model_rocky53</reference>
                </program>
                <program name="scen1_sensor_processing_2">
                    <reference>model_rocky53</reference>
                </program>
                <program name="scen1_wait_nfs_mount_3">
                    <reference>model_rocky53</reference>
                    <start_sequence>1</start_sequence>
                    <wait_exit>true</wait_exit>
                </program>
            </programs>
        </application>
</root>
```

Note: *About the choice to prefix all program names with 'scen1_'*

These programs are all included in a Supervisor group named `scen1`. It may indeed seem useless to add the information into the program name. Actually the program names are quite generic and at some point the intention is to group all the applications of the different use cases into an unique **Supvisors** configuration. Adding `scen1` at this point is just

---

to avoid overwriting of program definitions.

Note: A few words about how the `scen1_wait_nfs_mount[_X]` programs have been introduced here. It has to be noted that:

- the `start_sequence` of these programs is lower than the `start_sequence` of the other application programs ;

- their attribute `wait_exit` is set to `true`.

The consequence is that the 3 programs `scen1_wait_nfs_mount[_X]` are started first on their respective node when starting the **scen1** application. Then **Supvisors** waits for *all* of them to exit before it triggers the starting of the other programs.

Well, assuming that the node name could be included as a prefix to the program names, that would simplify the rules file a bit.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<root>
    <!-- models -->
    <model name="model_rocky51">
        <identifiers>rocky51</identifiers>
        <start_sequence>2</start_sequence>
        <required>true</required>
    </model>
    <model name="model_rocky52">
        <reference>model_rocky51</reference>
        <identifiers>rocky52</identifiers>
    </model>
    <model name="model_rocky53">
        <reference>model_rocky51</reference>
        <identifiers>rocky53</identifiers>
    </model>
    <!-- Scenario 1 Application -->
    <application name="scen1">
        <start_sequence>1</start_sequence>
        <starting_failure_strategy>CONTINUE</starting_failure_strategy>
        <programs>
            <!-- Programs on rocky51 -->
            <program pattern="rocky51_">
                <reference>model_rocky51</reference>
            </program>
            <program name="scen1_wait_nfs_mount_1">
                <reference>model_rocky51</reference>
                <start_sequence>1</start_sequence>
                <wait_exit>true</wait_exit>
            </program>
            <!-- Programs on rocky52 -->
            <program pattern="rocky52_">
                <reference>model_rocky52</reference>
            </program>
            <program name="scen1_wait_nfs_mount_2">
                <reference>model_rocky52</reference>
                <start_sequence>1</start_sequence>
```

```xml
                    <wait_exit>true</wait_exit>
                </program>
                <!-- Programs on rocky53 -->
                <program pattern="rocky53_">
                    <reference>model_rocky53</reference>
                </program>
                <program name="scen1_wait_nfs_mount_3">
                    <reference>model_rocky53</reference>
                    <start_sequence>1</start_sequence>
                    <wait_exit>true</wait_exit>
                </program>
            </programs>
        </application>
</root>
```

A bit shorter, still functional but the program names are now quite ugly. And the non-distributed version has not been considered yet. With this approach, a different rules file is required to replace the node names with the developer's host name - assumed called `rocky51` here for the example.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<root>
    <!-- Scenario 1 Application -->
    <application name="scen1">
        <start_sequence>1</start_sequence>
        <starting_failure_strategy>CONTINUE</starting_failure_strategy>
        <programs>
            <!-- Programs on localhost -->
            <program pattern="">
                <identifiers>rocky51</identifiers>
                <start_sequence>1</start_sequence>
                <required>true</required>
            </program>
        </programs>
    </application>
</root>
```

This rules file is fairly simple here as all programs have the exactly same rules.

Hint: When the same rules apply to all programs in an application, an empty pattern can be used as it will match all program names of the application.

But actually, there is a much more simple solution in the present case. Let's consider this instead:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<root>
    <!-- models -->
    <model name="model_scenario_1">
        <start_sequence>2</start_sequence>
        <required>true</required>
    </model>
    <!-- Scenario 1 Application -->
```

```xml
    <application name="scen1">
        <start_sequence>1</start_sequence>
        <starting_failure_strategy>CONTINUE</starting_failure_strategy>
        <programs>
            <program pattern="">
                <reference>model_scenario_1</reference>
            </program>
            <program pattern="wait_nfs_mount">
                <reference>model_scenario_1</reference>
                <start_sequence>1</start_sequence>
                <wait_exit>true</wait_exit>
            </program>
        </programs>
    </application>
</root>
```

Much shorter. Yet it does the same. For both the distributed application and the non-distributed application !

The main point is that the `identifiers` attribute is not used at all. Clearly, this gives **Supvisors** the authorization to start all programs on every nodes. However **Supvisors** knows about the Supervisor configuration in the 3 nodes. When choosing a node to start a program, **Supvisors** considers the intersection between the authorized nodes - all of them here - and the possible nodes, i.e. the active nodes where the program is defined in Supervisor. One of the first decisions in this use case is that every programs are known to only one Supervisor instance so that gives **Supvisors** only one possibility.

For Requirement 3, **Supvisors** provides the operational status of the application based on the status of its processes, in accordance with their importance. In the present example, all programs are defined with the same importance (`required` set to `true`).

The key point here is that **Supvisors** is able to build a single application from the processes configured on the 3 nodes because the same group name (`scen1`) is used in all Supervisor configuration files. This also explains why `scen1_wait_nfs_mount[_X]` has been suffixed with a number. Otherwise, **Supvisors** would have detected 3 running instances of the same program in a *Managed* application, which is considered as a conflict and leads to a *Conciliation* phase. Please refer to *Conciliation* for more details.

Here follows the relevant sections of the `supervisord_distributed.conf` configuration file, including the declaration of the **Supvisors** plugin.

```ini
[include]
files = %(host_node_name)s/*.ini

[rpcinterface:supvisors]
supervisor.rpcinterface_factory = supvisors.plugin:make_supvisors_rpcinterface
supvisors_list = rocky51,rocky52,rocky53
rules_files = etc/supvisors_rules.xml

[ctlplugin:supvisors]
supervisor.ctl_factory = supvisors.supvisorsctl:make_supvisors_controller_plugin
```

And the equivalent in the `supervisord_localhost.conf` configuration file. No `supvisors_list` is provided here as the default value is the local host name, which is perfectly suitable here.

```ini
[include]
files = */programs_*.ini localhost/group_localhost.ini
```

```
[rpcinterface:supvisors]
supervisor.rpcinterface_factory = supvisors.plugin:make_supvisors_rpcinterface
rules_files = etc/supvisors_rules.xml

[ctlplugin:supvisors]
supervisor.ctl_factory = supvisors.supvisorsctl:make_supvisors_controller_plugin
```

The final file tree is as follows.

```
[bash] > tree
.
├── etc
│           ├── rocky51
│           │           ├── group_rocky51.ini
│           │           ├── programs_rocky51.ini
│           │           └── wait_nfs_mount.ini
│           ├── rocky52
│           │           ├── group_rocky52.ini
│           │           ├── programs_rocky52.ini
│           │           └── wait_nfs_mount.ini
│           ├── rocky53
│           │           ├── group_rocky53.ini
│           │           ├── programs_rocky53.ini
│           │           └── wait_nfs_mount.ini
│           ├── localhost
│           │           └── group_localhost.ini
│           ├── supervisord.conf -> supervisord_distributed.conf
│           ├── supervisord_distributed.conf
│           ├── supervisord_localhost.conf
│           └── supvisors_rules.xml
```

10.4.3 Control & Status

The operational status of **Scenario 1** required by the REQUIREMENT 3 is made available through:

- the *Application Page* of the **Supvisors** Web UI, as a LED near the application state,

- the *XML-RPC API* (example below),

- the *REST API* (if **supvisorsflask** is started),

- the *Status* of the extended **supervisorctl** or **supvisorsctl** (example below),

- the *Event interface*.

```
>>> from supervisor.childutils import getRPCInterface
>>> proxy = getRPCInterface({'SUPERVISOR_SERVER_URL': 'http://localhost:61000'})
>>> proxy.supvisors.get_application_info('scen1')
{'application_name': 'scen1', 'statecode': 2, 'statename': 'RUNNING', 'major_failure':␣
→False, 'minor_failure': False}
```

```
[bash] > supervisorctl -c etc/supervisord_localhost.conf application_info scen1
Node        State     Major  Minor
scen1       RUNNING   True   False


[bash] > supvisorsctl -s http://localhost:61000 application_info scen1
Node        State     Major  Minor
scen1       RUNNING   True   False
```

To restart the whole application (REQUIREMENT 5), the following methods are available:

- the *XML-RPC API* (example below),

- the *REST API* (if **supvisorsflask** is started),

- the *Status* of the extended **supervisorctl** or **supvisorsctl** (example below),

- the restart button at the top right of the *Application Page* of the **Supvisors** Web UI.

```
>>> from supervisor.childutils import getRPCInterface
>>> proxy = getRPCInterface({'SUPERVISOR_SERVER_URL': 'http://localhost:61000'})
>>> proxy.supvisors.restart_application('CONFIG', 'scen1')
True
```

```
[bash] > supervisorctl -c etc/supervisord_localhost.conf restart_application CONFIG scen1
scenario_1 restarted

[bash] > supvisorsctl -s http://localhost:61000 restart_application CONFIG scen1
scenario_1 restarted
```

Here is a snapshot of the Application page of the **Supvisors** Web UI for the `Scenario 1` application.

As a conclusion, all the requirements are met using **Supvisors** and without any impact on the application to be supervised. **Supvisors** improves application control and status.

10.5 Example

The full example is available in Supvisors Use Cases - Scenario 1.

SCENARIO 2

11.1 Context

In this use case, the application **Scenario 2** is used to control an item. It is delivered in 2 parts:

- **scen2_srv**: dedicated to application services and designed to run on a server only,

- **scen2_hci**: dedicated to Human Computer Interfaces (HCI) and designed to run on a console only.

scen2_hci is started on demand from a console, whereas **scen2_srv** is available on startup. **scen2_srv** cannot be distributed because of its inter-processes communication design. **scen2_hci** is not distributed so that the user gets all the windows on the same screen. An internal data bus will allow **scen2_hci** to communicate with **scen2_srv**.

Multiple instances of the **Scenario 2** application can be started because there are multiple items to control.

A common data bus - out of this application's scope - is available to exchange data between **Scenario 2** instances and other applications dealing with other types of items and/or a higher control/command application (as described in *Scenario 3*).

11.2 Requirements

Here follows the use case requirements.

11.2.1 Global requirements

Requirement 1
> Given X items to control, it shall be possible to start X instances of the application.

Requirement 2
> **scen2_hci** and **scen2_srv** shall not be distributed.

Requirement 3
> An operational status of each application shall be provided.

Requirement 4
> **scen2_hci** and **scen2_srv** shall start only when their internal data bus is operational.

Requirement 5
> The number of application instances running shall be limited in accordance with the resources available (consoles or servers up).

11.2.2 Services requirements

Requirement 10

      scen2_srv shall be started on servers only.

Requirement 11

      The X **scen2_srv** shall be started automatically.

Requirement 12

      Each **scen2_srv** shall be started once at most.

Requirement 13

      There shall be a load-balancing strategy to distribute the X **scen2_srv** over the servers.

Requirement 14

      Upon failure in its starting sequence, **scen2_srv** shall be stopped so that it doesn't consume resources uselessly.

Requirement 15

      As **scen2_srv** is highly dependent on its internal data bus, **scen2_srv** shall be fully restarted if its internal data bus crashes.

Requirement 16

      Upon server power down or failure, **scen2_srv** shall be restarted on another server, in accordance with the load-balancing strategy.

Requirement 17

      The **scen2_srv** interface with the common data bus shall be started only when the common data bus is operational.

11.2.3 HCI requirements

Requirement 20

      A **scen2_hci** shall be started upon user request.

Requirement 21

      The **scen2_hci** shall be started on the console from where the user request has been done.

Requirement 22

      When starting a **scen2_hci**, the user shall choose the item to control.

Requirement 23

      The user shall not be able to start two **scen2_hci** that control the same item.

Requirement 24

      Upon failure, the starting sequence of **scen2_hci** shall continue.

Requirement 25

      As **scen2_hci** is highly dependent on its internal data bus, **scen2_hci** shall be fully stopped if its internal data bus crashes.

Requirement 26

      Upon console failure, **scen2_hci** shall not be restarted on another console.

Requirement 27

      scen2_hci shall be stopped upon user request.

11.3 Supervisor configuration

The initial Supervisor configuration is as follows:

- The `bin` folder includes all the program scripts of the **Scenario 2** application. The scripts get the Supervisor `program_name` from the environment variable `${SUPERVISOR_PROCESS_NAME}`.

- The `template_etc` folder contains the generic configuration for the **Scenario 2** application:

    - the `console/group_hci.ini` file that contains the definition of the **scen2_hci** group and programs,

    - the `server/group_server.ini` file that contains the definition of the **scen2_srv** group and programs.

- The `etc` folder is the target destination for the configurations files of all applications to be supervised. In this example, it just contains a definition of the common data bus (refer to REQUIREMENT 17 (The scen2_srv interface with the common data bus shall be started only when the common data bus is operational.)) that will be auto-started on all **Supvisors** instances. The `etc` folder contains the Supervisor configuration files that will be used when starting **supervisord**.

    - the `supervisord_console.conf` includes the definition of groups and programs that are intended to run on the consoles,

    - the `supervisord_server.conf` includes the definition of groups and programs that are intended to run on the servers.

```
[bash] > tree
.
├── bin
│           ├── check_common_data_bus.sh
│           ├── check_internal_data_bus.sh
│           ├── common_bus_interface.sh
│           ├── common.sh
│           ├── config_manager.sh
│           ├── data_processing.sh
│           ├── internal_data_bus.sh
│           ├── sensor_acquisition.sh
│           ├── sensor_control.sh
│           └── sensor_view.sh
├── etc
│           ├── common
│           │       └── group_services.ini
│           ├── supervisord_console.conf
│           └── supervisord_server.conf
└── template_etc
    ├── console
    │       └── group_hci.ini
    └── server
        └── group_server.ini
```

11.3.1 Homogeneous applications

Let's tackle the first issue about REQUIREMENT 1. Supervisor does not provide support to handle *homogeneous* groups. It only provides support for *homogeneous* process groups. Defining *homogeneous* process groups in the present case won't help as the program instances cannot be shared across multiple groups. However, it is possible to assign multiple times the same program to different groups.

Assuming that the `Scenario 2` application is delivered with the Supervisor configuration files for one generic item to control and assuming that there are X items to control, the first job is duplicate X times all group definitions.

This may be a bit painful when X is great or when the number of applications concerned is great, so a script is provided in the **Supvisors** package to make life easier.

```
[bash] > supvisors_breed -h
usage: supvisors_breed [-h] -t TEMPLATE [-p PATTERN] -d DESTINATION
                       [-b app=nb [app=nb ...]] [-x] [-v]

Duplicate the application definitions

optional arguments:
  -h, --help            show this help message and exit
  -t TEMPLATE, --template TEMPLATE
                        the template folder
  -p PATTERN, --pattern PATTERN
                        the search pattern from the template folder
  -d DESTINATION, --destination DESTINATION
                        the destination folder
  -b app=nb [app=nb ...], --breed app=nb [app=nb ...]
                        the applications to breed
  -x, --extra           create new files
  -v, --verbose         activate logs
```

For this example, let's define X=3. Using greater values don't change the complexity of what follows. It would just need more resources to test. **supvisors_breed** duplicates 3 times the **scen2_srv** and **scen2_hci** groups found in the `template_etc` folder and writes new configuration files into the `etc` folder.

```
[bash] > supvisors_breed -d etc -t template_etc -b scen2_srv=3 -x -v
ArgumentParser: Namespace(breed={'scen2_srv': 3}, destination='etc', extra=True, pattern=
→'server/*.ini', template='template_etc', verbose=True)
Configuration files found:
    server/programs_server.ini
    server/group_server.ini
Template group elements found:
    group:scen2_srv
New File: server/group_scen2_srv_01.ini
New [group:scen2_srv_01]
New File: server/group_scen2_srv_02.ini
New [group:scen2_srv_02]
New File: server/group_scen2_srv_03.ini
New [group:scen2_srv_03]
Writing file: etc/server/programs_server.ini
Empty sections for file: server/group_server.ini
Writing file: etc/server/group_scen2_srv_01.ini
Writing file: etc/server/group_scen2_srv_02.ini
Writing file: etc/server/group_scen2_srv_03.ini
```

```
[bash] > supvisors_breed -d etc -t template_etc -b scen2_hci=3 -x -v
ArgumentParser: Namespace(breed={'scen2_hci': 3}, destination='etc', extra=True, pattern=
↪'console/*ini', template='template_etc', verbose=True)
Configuration files found:
    console/group_console.ini
    console/programs_console.ini
Template group elements found:
    group:scen2_hci
New File: console/group_scen2_hci_01.ini
New [group:scen2_hci_01]
New File: console/group_scen2_hci_02.ini
New [group:scen2_hci_02]
New File: console/group_scen2_hci_03.ini
New [group:scen2_hci_03]
Empty sections for file: console/group_console.ini
Writing file: etc/console/programs_console.ini
Writing file: etc/console/group_scen2_hci_01.ini
Writing file: etc/console/group_scen2_hci_02.ini
Writing file: etc/console/group_scen2_hci_03.ini
```

> **Attention:** The program definitions **scen2_internal_data_bus** and **scen2_internal_check_data_bus** are common to **scen2_srv** and **scen2_hci**. In the use case design, it doesn't matter as Supervisor is not configured to include these definitions together. Otherwise, loading twice the same program definition may have led to incorrect behavior (unless they're strictly identical).

> **Note:** *About the choice to prefix all program names with 'scen2_'*

These programs are all included in a Supervisor group named scen2. It may indeed seem useless to add the information into the program name. Actually the program names are quite generic and at some point the intention is to group all the applications of the different use cases into an unique **Supvisors** configuration. Adding scen2 at this point is just to avoid overwriting of program definitions.

The resulting file tree is as follows.

```
[bash] > tree
.
├── bin
│           ── check_common_data_bus.sh
│           ── check_internal_data_bus.sh
│           ── common_bus_interface.sh
│           ── common.sh
│           ── config_manager.sh
│           ── data_processing.sh
│           ── internal_data_bus.sh
│           ── sensor_acquisition.sh
│           ── sensor_control.sh
│           └── sensor_view.sh
├── etc
```

```
                    ├── common
                    │           └── programs_services.ini
                    ├── console
                    │           ├── group_scen2_hci_01.ini
                    │           ├── group_scen2_hci_02.ini
                    │           ├── group_scen2_hci_03.ini
                    │           └── programs_console.ini
                    ├── server
                    │           ├── group_scen2_srv_01.ini
                    │           ├── group_scen2_srv_02.ini
                    │           ├── group_scen2_srv_03.ini
                    │           └── programs_server.ini
                    ├── supervisord_console.conf
                    └── supervisord_server.conf
        └── template_etc
            ├── console
            │           └── group_hci.ini
            └── server
                └── group_server.ini
```

Note: As a definition, let's say that the combination of **scen2_srv_01** and **scen2_hci_01** is the **Scenario 2** application that controls the item 01.

Here follows what the include section may look like in both Supervisor configuration files.

```
# include section in supervisord_server.conf
[include]
files = common/*.ini server/*.ini


# include section in supervisord_console.conf
[include]
files = common/*.ini console/*.ini
```

From this point, the `etc` folder contains the Supervisor configuration that satisfies REQUIREMENT 1.

11.3.2 Requirements met with Supervisor only

Server side

REQUIREMENT 10 (scen2_srv shall be started on servers only.) is satisfied by the `supervisord_[server|console].conf` files. Only the `supervisord_server.conf` file holds the information to start **scen2_srv**.

REQUIREMENT 16 (Upon server power down or failure, scen2_srv shall be restarted on another server, in accordance with the load-balancing strategy.) implies that all **scen2_srv** definitions must be available on all servers. So a single `supervisord_server.conf` including all **scen2_srv** definitions and made available on all servers still makes sense.

The automatic start required in REQUIREMENT 11 (The X scen2_srv shall be started automatically.) could be achieved by using the Supervisor `autostart=True` on the programs but considering REQUIREMENT 2 and REQUIREMENT 12 (Each scen2_srv shall be started once at most.), that becomes a bit complex.

It looks like 2 sets of program definitions are needed, one definition with `autostart=True` and one with `autostart=False`. All **scen2_srv** groups must include program definitions with a homogeneous use of `autostart`.

In order to maintain the load balancing required in REQUIREMENT 13 (There shall be a load-balancing strategy to distribute the X scen2_srv over the servers.), the user must define in advance which **scen2_srv** shall run on which server and use the relevant program definition (`autostart`-dependent).

This all ends up with a dedicated `supervisord_server[_S].conf` configuration file for each server.

Console side

Now let's have a look at the console side. Like for the server configuration, all **scen2_hci** must be available on all consoles to satisfy REQUIREMENT 22 (When starting a scen2_hci, the user shall choose the item to control.). Per definition, choosing one of the **scen2_hci** is a way to choose the item to control.

REQUIREMENT 20 (A scen2_hci shall be started upon user request.), REQUIREMENT 21 (The scen2_hci shall be started on the console from where the user request has been done.) and REQUIREMENT 27 (scen2_hci shall be stopped upon user request.) are simply met using **supervisorctl** commands.

```
[bash] > supervisorctl start scen2_hci_01:*
[bash] > supervisorctl stop scen2_hci_01:*
```

It is possible to do that from the Supervisor Web UI too, one program at a time, although it would be a bit clumsy and a source of mistakes that would break REQUIREMENT 2.

However, there's nothing to prevent another user to start the same **scen2_hci** on his console, as required in REQUIREMENT 23 (The user shall not be able to start two scen2_hci that control the same item.).

All the other requirements are about operational status, staged start sequence and automatic behaviour and there's no Supervisor function for that. It would require dedicated software development to satisfy them. Or **Supvisors** may be used, which is the point of the next section.

11.4 Involving Supvisors

When involving **Supvisors**, all `Scenario 2` programs are configured using `autostart=false`. Only the common data bus - that is outside of the application scope - will be auto-started.

The **Supvisors** configuration is built over the Supervisor configuration defined above.

11.4.1 Rules file

All the requirements about automatic behaviour are dealt within the **Supvisors** rules file. This section will detail step by step how it is built against the requirements.

First, as all **scen2_srv** instances have the same rules, a single application entry with a matching pattern is used for all of them. The same idea applies to **scen2_hci**. Declaring these applications in the **Supvisors** rules file makes them all *Managed* in **Supvisors**, which gives control over the X instances of the `Scenario 2` application, as required in REQUIREMENT 1.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<root>
    <application pattern="scen2_srv_"/>
    <application pattern="scen2_hci_"/>
</root>
```

REQUIREMENT 2 is just about declaring the `distribution` element to `SINGLE_INSTANCE`. It tells **Supvisors** that all the programs of the application have to be started in the same **Supvisors** instance.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<root>
    <application pattern="scen2_srv_">
        <distribution>SINGLE_INSTANCE</distribution>
    </application>

    <application pattern="scen2_hci_">
        <distribution>SINGLE_INSTANCE</distribution>
    </application>
</root>
```

So far, all applications can be started on any **Supvisors** instance. Let's compel **scen2_hci** to consoles and **scen2_srv** to servers, which satisfies REQUIREMENT 10 and contributes to some console-related requirements. For better readability, Instance aliases are introduced.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<root>
    <alias name="servers">server_1,server_2,server_3</alias>
    <alias name="consoles">console_1,console_2,console_3</alias>

    <application pattern="scen2_srv_">
        <distribution>SINGLE_INSTANCE</distribution>
        <identifiers>servers</identifiers>
    </application>

    <application pattern="scen2_hci_">
        <distribution>SINGLE_INSTANCE</distribution>
        <identifiers>consoles</identifiers>
    </application>
</root>
```

It's time to introduce the staged start sequences. REQUIREMENT 11 asks for an automatic start of **scen2_srv**, so a strictly positive `start_sequence` is added to the application configuration.

Because of REQUIREMENT 4, **scen2_srv** and **scen2_hci** applications will be started in three phases:

- first the **scen2_internal_data_bus** program,

- then the **scen2_check_internal_data_bus** whose job is to periodically check the **scen2_internal_data_bus** status and exit when it is operational,

- other programs.

scen2_internal_data_bus and **scen2_check_internal_data_bus** are common to **scen2_srv** and **scen2_hci** and follow the same rules so it makes sense to define a common model for them.

Due to REQUIREMENT 17, two additional phases are needed for **scen2_srv**:

- the **scen2_check_common_data_bus** whose job is to periodically check the **common_data_bus** status and exit when it is operational,

- finally the **scen2_common_bus_interface**.

They are set at the end of the starting sequence so that the core of the **scen2_srv** application can be operational as a standalone application, even if it's not connected to other possible applications in the system.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<root>
    <alias name="servers">server_1,server_2,server_3</alias>
    <alias name="consoles">console_1,console_2,console_3</alias>

    <model name="model_services">
        <start_sequence>3</start_sequence>
        <required>true</required>
    </model>
    <model name="check_data_bus">
        <start_sequence>2</start_sequence>
        <required>true</required>
        <wait_exit>true</wait_exit>
    </model>
    <model name="data_bus">
        <start_sequence>1</start_sequence>
        <required>true</required>
    </model>

    <application pattern="scen2_srv_">
        <distribution>SINGLE_INSTANCE</distribution>
        <identifiers>servers</identifiers>
        <start_sequence>1</start_sequence>
        <programs>
            <program name="scen2_common_bus_interface">
                <reference>model_services</reference>
                <start_sequence>4</start_sequence>
            </program>
            <program name="scen2_check_common_data_bus">
                <reference>check_data_bus</reference>
                <start_sequence>3</start_sequence>
            </program>
            <pattern name="">
                <reference>model_services</reference>
            </pattern>
            <program name="scen2_check_internal_data_bus">
                <reference>check_data_bus</reference>
            </program>
            <program name="scen2_internal_data_bus">
                <reference>data_bus</reference>
            </program>
        </programs>
    </application>

    <application pattern="scen2_hci_">
        <distribution>SINGLE_INSTANCE</distribution>
        <identifiers>consoles</identifiers>
        <programs>
            <program pattern="">
                <start_sequence>3</start_sequence>
            </program>
            <program name="scen2_check_internal_data_bus">
                <reference>check_data_bus</reference>
```

```xml
                </program>
                <program name="scen2_internal_data_bus">
                    <reference>data_bus</reference>
                </program>
            </programs>
        </application>
</root>
```

Let's now introduce the automatic behaviors.

REQUIREMENT 5 implies to check the resources available before allowing an application or a program to be started. **Supvisors** has been designed to consider the resources needed by the program over the resources actually taken. To achieve that, the `expected_loading` elements of the programs have been set (quite arbitrarily for the example).

The `starting_strategy` element of the **scen2_srv** application is set to LESS_LOADED to satisfy REQUIREMENT 13. Before **Supvisors** starts an application or a program, it relies on the `expected_loading` set just before to:

- evaluate the current load on all **Supvisors** instances (due to processes already running),

- choose the **Supvisors** instance having the lowest load and that can accept the additional load required by the program or application to start.

If none found, the application or the program is not started, which satisfies REQUIREMENT 5.

The `starting_strategy` element of the **scen2_hci** application is set to LOCAL to satisfy REQUIREMENT 21. Actually this value is only used as a default parameter in the *Application Page* of the **Supvisors** Web UI and can be overridden.

In the same vein, the `starting_failure_strategy` element of the **scen2_srv** application is set to STOP_APPLICATION (REQUIREMENT 14 (Upon failure in its starting sequence, scen2_srv shall be stopped so that it doesn't consume resources uselessly.)) and the `starting_failure_strategy` element of the **scen2_hci** application is set to CONTINUE (REQUIREMENT 24 (Upon failure, the starting sequence of scen2_hci shall continue.)).

Finally, there is automatic behavior to be set on the **scen2_internal_data_bus** programs. The `running_failure_strategy` element of the `internal_data_bus` pattern is set to:

- RESTART_APPLICATION for **scen2_srv** applications (REQUIREMENT 15 (As scen2_srv is highly dependent on its internal data bus, scen2_srv shall be fully restarted if its internal data bus crashes.)),

- STOP_APPLICATION for **scen2_hci** applications (REQUIREMENT 25 (As scen2_hci is highly dependent on its internal data bus, scen2_hci shall be fully stopped if its internal data bus crashes.)),

A last impact on the rules file is about the application operational status (REQUIREMENT 3). Setting the `required` element on the program definitions will discriminate between major and minor failures for the applications. **Supvisors** will provide a separate operational status for **scen2_srv** and **scen2_hci**. It is still the user's responsibility to merge the status of **scen2_srv_N** and **scen2_hci_N** to get a global status for the **Scenario 2** application controlling the Nth item.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<root>
    <!-- aliases -->
    <alias name="servers">server_1,server_2,server_3</alias>
    <alias name="consoles">console_1,console_2,console_3</alias>

    <!-- models -->
    <model name="model_services">
        <start_sequence>3</start_sequence>
        <required>true</required>
        <expected_loading>10</expected_loading>
```

```xml
    </model>
    <model name="check_data_bus">
        <start_sequence>2</start_sequence>
        <required>true</required>
        <wait_exit>true</wait_exit>
    </model>
    <model name="data_bus">
        <start_sequence>1</start_sequence>
        <required>true</required>
        <expected_loading>2</expected_loading>
    </model>

    <!-- Scenario 2 Applications -->
    <!-- Services -->
    <application pattern="scen2_srv_">
        <distribution>SINGLE_INSTANCE</distribution>
        <identifiers>servers</identifiers>
        <start_sequence>1</start_sequence>
        <starting_strategy>LESS_LOADED</starting_strategy>
        <starting_failure_strategy>STOP</starting_failure_strategy>
        <programs>
            <program name="scen2_common_bus_interface">
                <reference>model_services</reference>
                <start_sequence>4</start_sequence>
            </program>
            <program name="scen2_check_common_data_bus">
                <reference>check_data_bus</reference>
                <start_sequence>3</start_sequence>
            </program>
            <pattern name="">
                <reference>model_services</reference>
            </pattern>
            <program name="scen2_check_internal_data_bus">
                <reference>check_data_bus</reference>
            </program>
            <program name="scen2_internal_data_bus">
                <reference>data_bus</reference>
                <running_failure_strategy>RESTART_APPLICATION</running_failure_strategy>
            </program>
        </programs>
    </application>

    <!-- HCI -->
    <application pattern="scen2_hci_">
        <distribution>SINGLE_INSTANCE</distribution>
        <identifiers>consoles</identifiers>
        <starting_strategy>LOCAL</starting_strategy>
        <starting_failure_strategy>CONTINUE</starting_failure_strategy>
        <programs>
            <program pattern="">
                <start_sequence>3</start_sequence>
                <expected_loading>8</expected_loading>
```

```
            </program>
            <program name="scen2_check_internal_data_bus">
                <reference>check_data_bus</reference>
            </program>
            <program name="scen2_internal_data_bus">
                <reference>data_bus</reference>
                <running_failure_strategy>STOP_APPLICATION</running_failure_strategy>
            </program>
        </programs>
    </application>

</root>
```

Two last requirements to discuss. Actually they're already met by the combination of others.

REQUIREMENT 16 aks for a restart of **scen2_srv** on another server if the server it is running on is shut down or crashes. Due to the non-distributed status of this application, all its processes will be declared FATAL in **Supvisors** for such an event and the application will be declared stopped. The RESTART_APPLICATION set to the **scen2_internal_data_bus** program (REQUIREMENT 15) will then take over and restart the application on another server, using the starting strategy LESS_LOADED set to the **scen2_srv** application (REQUIREMENT 13) and in accordance with the resources available (REQUIREMENT 5).

On the same principle, the running failure strategy applied to the **scen2_internal_data_bus** program of the **scen2_hci** application is STOP_APPLICATION. In the event of a console shutdown or crash, the **scen2_hci** will already be declared stopped, so nothing more to do and REQUIREMENT 26 (Upon console failure, scen2_hci shall not be restarted on another console.) is therefore satisfied.

11.4.2 Control & Status

The operational status of **Scenario 2** required by the REQUIREMENT 3 is made available through:

- the *Application Page* of the **Supvisors** Web UI, as a LED near the application state,

- the *XML-RPC API* (example below),

- the *REST API* (if **supvisorsflask** is started),

- the *Status* of the extended **supervisorctl** or **supvisorsctl** (example below),

- the *Event interface*.

For the example, the following context applies:

- due to limited resources - 3 nodes are available (rocky51, rocky52 and rocky53) -, each node hosts 2 **Supvisors** instances, one server and one console ;

- **common_data_bus** is *Unmanaged* so **Supvisors** always considers this 'application' as STOPPED (the process status is yet RUNNING) ;

- **scen2_srv_01**, **scen2_srv_02** and **scen2_srv_03** are running on server_1, server_2, server_3, respectively hosted by the nodes rocky51, rocky52, rocky53 ;

- **scen2_hci_02** has been started on console_3 ;

- an attempt to start **scen2_hci_03** on the server rocky51 has been rejected (only allowed on a console).

```
>>> from supervisor.childutils import getRPCInterface
>>> proxy = getRPCInterface({'SUPERVISOR_SERVER_URL': 'http://localhost:61000'})
>>> proxy.supvisors.get_all_applications_info()
[{'application_name': 'common_data_bus', 'statecode': 0, 'statename': 'STOPPED', 'major_
→failure': False, 'minor_failure': False},
{'application_name': 'scen2_srv_01', 'statecode': 2, 'statename': 'RUNNING', 'major_
→failure': False, 'minor_failure': False},
{'application_name': 'scen2_srv_02', 'statecode': 2, 'statename': 'RUNNING', 'major_
→failure': False, 'minor_failure': False},
{'application_name': 'scen2_srv_03', 'statecode': 2, 'statename': 'RUNNING', 'major_
→failure': False, 'minor_failure': False},
{'application_name': 'scen2_hci_01', 'statecode': 0, 'statename': 'STOPPED', 'major_
→failure': False, 'minor_failure': False},
{'application_name': 'scen2_hci_02', 'statecode': 2, 'statename': 'RUNNING', 'major_
→failure': False, 'minor_failure': False},
{'application_name': 'scen2_hci_03', 'statecode': 0, 'statename': 'STOPPED', 'major_
→failure': True, 'minor_failure': True}]
```

```
[bash] > supvisorsctl -s http://localhost:61000 application_info
Application       State     Major  Minor
common_data_bus   STOPPED   False  False
scen2_srv_01      RUNNING   False  False
scen2_srv_02      RUNNING   False  False
scen2_srv_03      RUNNING   False  False
scen2_hci_01      STOPPED   False  False
scen2_hci_02      RUNNING   False  False
scen2_hci_03      STOPPED   True   True
```

To start a **scen2_hci** in accordance with Requirement 20, Requirement 21 and Requirement 22, the following methods are available:

- the *XML-RPC API* (example below - **beware of the target**),

- the *REST API* (if **supvisorsflask** is started),

- the *Application Control* of the extended **supervisorctl** or **supvisorsctl** (examples below):

  - using the configuration file **if executed from the targeted console**,

  - using the URL otherwise,

- the start button at the top right of the *Application Page* of the **Supvisors** Web UI, **assuming that the user has navigated to this page using the relevant |Supvisors| instance** (check the url if necessary).

```
>>> from supervisor.childutils import getRPCInterface
>>> proxy = getRPCInterface({'SUPERVISOR_SERVER_URL': 'http://rocky53:61000'})
>>> proxy.supvisors.start_application('LOCAL', 'scen2_hci_02')
True
```

```
[bash] > hostname
rocky53
[bash] > supervisorctl -c etc/supervisord_console.conf start_application LOCAL scen2_hci_
→02
```

(continues on next page)

```
scen2_hci_02 started

[bash] > hostname
rocky51
[bash] > supvisorsctl -s http://rocky53:61000 start_application LOCAL scen2_hci_02
scen2_hci_02 started
```

Hint: Supervisor's `supervisorctl` does not provide support for extended API using the `-s` URL option. But **Supvisors**' `supvisorsctl` does.

REQUIREMENT 12 and REQUIREMENT 23 require that one instance of one instance of **scen2_srv_N** and **scen2_hci_N** are running at most. This is all managed by **Supvisors** as they are *Managed* applications. If the **scen2_srv_N** is already running on a console, the **Supvisors** Web UI will prevent the user to start **scen2_srv_N** from another console. **Supvisors** XML-RPC and extended `supervisorctl` commands will be rejected.

Attention: The Supervisor commands (XML-RPC or `supervisorctl`) are NOT curbed in any way by **Supvisors** so it is still possible to break the rule using Supervisor itself. In the event of multiple `Scenario 2` programs running, **Supvisors** will detect the conflicts and enter a `CONCILIATION` state. Please refer to *Conciliation* for more details.

To stop a **scen2_hci** (REQUIREMENT 27), the following methods are available:

- the *XML-RPC API* (example below - **whatever the target**),

- the *REST API* (if `supvisorsflask` is started),

- the *Application Control* of the extended `supervisorctl` or `supvisorsctl from any |Supvisors| instance` (example below),

- the stop button at the top right of the *Application Page* of the **Supvisors** Web UI, **whatever the |Supvisors| instance displaying this page**.

Indeed, as **Supvisors** knows where the application is running, it is able to drive the application stop from anywhere.

```
>>> from supervisor.childutils import getRPCInterface
>>> proxy = getRPCInterface({'SUPERVISOR_SERVER_URL': 'http://localhost:61000'})
>>> proxy.supvisors.stop_application('scen2_hci_02')
True
```

```
[bash] > hostname
rocky51
[bash] > supervisorctl -c etc/supervisord_server.conf stop_application scen2_hci_02
scen2_hci_02 stopped
```

As a conclusion, all the requirements are met using **Supvisors** and without any impact on the application to be supervised. **Supvisors** improves application control and status.

11.5 Example

The full example is available in Supvisors Use Cases - Scenario 2.

Two versions are provided:

- one corresponding to the description above ;

- one based on the **Supvisors** discovery mode, using stereotypes rather than aliases.

SCENARIO 3

12.1 Context

The application **Scenario 3** is the control/command application referenced to in **Scenario 2**. It is delivered in 2 parts:

- **scen3_srv**: dedicated to application services and designed to run on a server only,
- **scen3_hci**: dedicated to Human Computer Interfaces (HCI) and designed to run on a console only.

Both **scen3_hci** and **scen3_srv** are started automatically.

scen3_srv is unique and distributed over the servers. One instance of the **scen3_hci** application is started per console.

An internal data bus will allow all instances of **scen3_hci** to communicate with **scen3_srv**.

A common data bus - out of this application's scope - is available to exchange data between **Scenario 3** and other applications dealing with other types of items (typically as described in the use case *Scenario 2*).

12.2 Requirements

Here follows the use case requirements.

12.2.1 Global requirements

Requirement 1
>   **scen3_hci** and **scen3_srv** shall be started automatically.

Requirement 2
>   An operational status of each application shall be provided.

Requirement 3
>   **scen3_hci** and **scen3_srv** shall start only when their internal data bus is operational.

Requirement 4
>   The starting of **scen3_hci** and **scen3_srv** shall not be aborted in case of failure of one of its programs.

Requirement 5
>   Although the **Scenario 3** is not directly concerned by resource limitations, it shall partake in the overall load information.

12.2.2 Services requirements

Requirement 10
> **scen3_srv** shall be distributed on servers only.

Requirement 11
> There shall be a load-balancing strategy to distribute the **scen3_srv** programs over the servers.

Requirement 12
> As **scen3_srv** is distributed, its internal data bus shall be made available on all servers.

Requirement 13
> Upon server power down or failure, the programs of **scen3_srv** shall be re-distributed on the other servers, in accordance with the load-balancing strategy.

Requirement 14
> The **scen3_srv** interface with the common data bus shall be started only when the common data bus is operational.

12.2.3 HCI requirements

Requirement 20
> A **scen3_hci** shall be started on each console.

Requirement 21
> Upon console failure, **scen3_hci** shall not be restarted on another console.

12.3 Supervisor configuration

The initial Supervisor configuration is as follows:

- The `bin` folder includes all the program scripts of the **Scenario 3** application. The scripts get the Supervisor `program_name` from the environment variable `${SUPERVISOR_PROCESS_NAME}`.

- The `template_etc` folder contains the generic configuration for the **scen3_hci** group and programs.

- The `etc` folder is the target destination for the configurations files of all applications to be supervised. It initially contains:

  - a definition of the common data bus (refer to REQUIREMENT 14) that will be auto-started on all **Supvisors** instances.

  - the configuration of the **scen3_srv** group and programs.

  - the Supervisor configuration files that will be used when starting **supervisord**:

    * the `supervisord_console.conf` includes the configuration files of the programs that are intended to run on the consoles,

    * the `supervisord_server.conf` includes the configuration files of the programs that are intended to run on the servers.

```
[bash] > tree
.
├── bin
│       ├── chart_view.sh
│       ├── check_common_data_bus.sh
```

<div align="right">(continues on next page)</div>

```
│               ├── check_internal_data_bus.sh
│               ├── common_bus_interface.sh
│               ├── common.sh
│               ├── internal_data_bus.sh
│               ├── item_control.sh
│               ├── item_manager.sh
│               └── track_manager.sh
├── etc
│       ├── common
│       │       └── group_services.ini
│       ├── server
│       │       └── group_server.ini
│       ├── supervisord_console.conf
│       └── supervisord_server.conf
└── template_etc
    └── console
        └── group_hci.ini
```

The first update to the configuration is driven by the fact that the program **internal_data_bus** is common to **scen3_srv** and **scen3_hci**. As the **Scenario 3** application may be distributed on all consoles and servers (Requirement 12 and Requirement 20), this program follows the same logic as the common data bus, so let's remove it from **scen3_srv** and **scen3_hci** and to insert it into the common services.

Like the **scen2_hci** of the **Scenario 2**, **scen3_hci** needs to be duplicated so this an instance could be started on each console. In this example, there are 3 consoles. **supvisors_breed** will thus be used again to duplicate 3 times the **scen3_hci** groups and programs found in the template_etc folder.

However, unlike **Scenario 2** where any **scen2_hci** could be started from any console, only one **scen3_hci** has to be started here per console and including more than one instance of it in the local Supervisor is useless. So the option -x of **supvisors_breed** will be used so that the duplicated configurations are written into separated files in the etc folder. That will allow more flexibility when including files from the Supervisor configuration file.

```
[bash] > supvisors_breed -d etc -t template_etc -b scen3_hci=3 -x -v
ArgumentParser: Namespace(breed={'scen3_hci': 3}, destination='etc', extra=True, pattern=
↪'**/*.ini', template='template_etc', verbose=True)
Configuration files found:
    console/group_console.ini
    console/programs_console.ini
Template group elements found:
    group:scen3_hci
New File: console/group_scen3_hci_01.ini
New [group:scen3_hci_01]
New File: console/group_scen3_hci_02.ini
New [group:scen3_hci_02]
New File: console/group_scen3_hci_03.ini
New [group:scen3_hci_03]
Empty sections for file: console/group_console.ini
Writing file: etc/console/programs_console.ini
Writing file: etc/console/group_scen3_hci_01.ini
Writing file: etc/console/group_scen3_hci_02.ini
Writing file: etc/console/group_scen3_hci_03.ini
```

Note: *About the choice to prefix all program names with 'scen3_'*

These programs are all included in a Supervisor group named `scen3`. It may indeed seem useless to add the information into the program name. Actually the program names are quite generic and at some point the intention is to group all the applications of the different use cases into an unique **Supvisors** configuration. Adding `scen3` at this point is just to avoid overwriting of program definitions.

Based on the expected names of the consoles, an additional script is used to sort the files generated. The resulting file tree is as follows.

```
[bash] > tree
.
├── bin
│           ├── chart_view.sh
│           ├── check_common_data_bus.sh
│           ├── check_internal_data_bus.sh
│           ├── common_bus_interface.sh
│           ├── common.sh
│           ├── internal_data_bus.sh
│           ├── item_control.sh
│           ├── item_manager.sh
│           ├── system_health.sh
│           └── track_manager.sh
├── etc
│        ├── common
│        │          └── group_services.ini
│        ├── console
│        │          ├── console_1
│        │          │          └── group_scen3_hci_01.ini
│        │          ├── console_2
│        │          │          └── group_scen3_hci_02.ini
│        │          ├── console_3
│        │          │          └── group_scen3_hci_03.ini
│        │          └── programs_console.ini
│        ├── server
│        │          └── group_server.ini
│        ├── supervisord_console.conf
│        └── supervisord_server.conf
└── template_etc
     └── console
          ├── group_console.ini
          └── programs_console.ini
```

Here follows what the include section may look like in both Supervisor configuration files.

```
# include section in supervisord_server.conf
[include]
files = common/*.ini server/*.ini

# include section in supervisord_console.conf
[include]
files = common/*.ini console/*.ini console/%(host_node_name)s/*.ini
```

12.3.1 Requirements met with Supervisor only

Based on the configuration defined above, Supervisor can definitely satisfy the following requirements: REQUIREMENT 1, REQUIREMENT 4, REQUIREMENT 10, REQUIREMENT 12, REQUIREMENT 20 and REQUIREMENT 21.

As already described in the previous use cases, requirements about operational status, staged start sequence and automatic behaviour are out of Supervisor's scope and would require dedicated software development to satisfy them.

Next section details how **Supvisors** can be used to deal with them.

12.4 Involving Supvisors

As usual, when involving **Supvisors**, all `Scenario 3` programs are configured using `autostart=false`. Exception is made to the programs in the `etc/common` folder (common and internal data buses).

The **Supvisors** configuration is built over the Supervisor configuration defined above.

12.4.1 Rules file

As the logic of the starting sequence of **Scenario 3** very similar to the *Scenario 2* use case, there won't be much detail about that in the present section. Please refer to the other use case if needed.

The main difference is that **scen3_internal_data_bus** has been removed. As a reminder, the consequence of REQUIREMENT 12 and REQUIREMENT 20 is that this program must run in all **Supvisors** instances, so it has been moved to the services file and configured as auto-started.

Both applications **scen3_srv** and **scen3_hci** have their `start_sequence` set and strictly positive so they will be automatically started, as required by REQUIREMENT 1. Please just note that **scen3_hci** has a greater `start_sequence` than **scen3_srv** so it will be started only when **scen3_srv** is fully running.

REQUIREMENT 3 and REQUIREMENT 14 are satisfied by the following programs that are configured with a `wait_exit` option:

- **scen3_check_internal_data_bus** and **scen3_check_common_data_bus** for **scen3_srv**.
- **scen3_check_internal_data_bus** for **scen3_hci**

The `distribution` options is not set for the **scen3_srv** application. As it is defaulted to `true` and as all **scen3_srv** programs are configured with the `identifiers` option set with the servers alias, **scen3_srv** will be distributed over the servers when starting, as required by REQUIREMENT 10.

The `distribution` options is set to `false` for the **scen3_hci**. In this case, only the `identifiers` option set to the application element is taken into account and NOT the `identifiers` options set to the program elements. The value `#,consoles` used here needs some further explanation.

When using hashtags in `identifiers`, applications and programs cannot be started anywhere until **Supvisors** solves the 'equation'. As defined in *Using patterns and signs*, an association will be made between the Nth application **scen3_hci_N** and the Nth element of the `consoles` list. In the example, **scen3_hci_01** will be mapped with **Supvisors** instance `console_1` once resolved.

This will result in having exactly one **scen3_hci** application per console, which satisfies REQUIREMENT 20.

Note: In **scen3_hci**, the program `scen3_check_internal_data_bus` references a model that uses server **Supvisors** instances in its `identifiers` option. It doesn't matter in the present case because, as told before, the `identifiers` option of the non-distributed application supersedes the `identifiers` eventually set in its programs.

Let's now focus on the strategies and options used at application level.

In accordance with REQUIREMENT 4, the `starting_failure_strategy` option of both **scen3_srv** and **scen3_hci** are set to CONTINUE (default is `ABORT`).

To satisfy REQUIREMENT 13, the `running_failure_strategy` option of **scen3_srv** has been set to `RESTART_PROCESS` (via the model `model_services`). For **scen3_hci**, this option is not set and the default CONTINUE is then used, as required in REQUIREMENT 21. Anyway, as the Nth application si only known by the Supervisor of the Nth console, it is just impossible to start this application elsewhere.

Finally, in order to satisfy REQUIREMENT 5 and to have a load-balancing over the server **Supvisors** instances (refer to REQUIREMENT 11), an arbitrary `expected_loading` has been set on programs. It is expected that relevant figures are used for a real application. The `starting_strategy` option of **scen3_srv** has been set to LESS_LOADED.

Here follows the resulting rules file.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<root>
    <!-- aliases -->
    <alias name="servers">server_1,server_2,server_3</alias>
    <alias name="consoles">console_1,console_2,console_3</alias>

    <!-- models -->
    <model name="model_services">
        <identifiers>servers</identifiers>
        <start_sequence>2</start_sequence>
        <required>true</required>
        <expected_loading>2</expected_loading>
        <running_failure_strategy>RESTART_PROCESS</running_failure_strategy>
    </model>
    <model name="check_data_bus">
        <identifiers>servers</identifiers>
        <start_sequence>1</start_sequence>
        <required>true</required>
        <wait_exit>true</wait_exit>
    </model>

    <!-- Scenario 3 Applications -->
    <!-- Services -->
    <application name="scen3_srv">
        <start_sequence>1</start_sequence>
        <starting_strategy>LESS_LOADED</starting_strategy>
        <starting_failure_strategy>CONTINUE</starting_failure_strategy>
        <programs>
            <program name="scen3_common_bus_interface">
                <reference>model_services</reference>
                <start_sequence>3</start_sequence>
            </program>
            <program name="scen3_check_common_data_bus">
                <reference>check_data_bus</reference>
                <start_sequence>2</start_sequence>
            </program>
            <program pattern="">
                <reference>model_services</reference>
            </program>
```

```xml
            <program name="scen3_check_internal_data_bus">
                <reference>check_data_bus</reference>
            </program>
        </programs>
    </application>

    <!-- HCI -->
    <application pattern="scen3_hci_">
        <distribution>SINGLE_INSTANCE</distribution>
        <identifiers>#,consoles</identifiers>
        <start_sequence>3</start_sequence>
        <starting_failure_strategy>CONTINUE</starting_failure_strategy>
        <programs>
            <program pattern="">
                <start_sequence>2</start_sequence>
                <expected_loading>8</expected_loading>
            </program>
            <program name="scen3_check_internal_data_bus">
                <reference>check_data_bus</reference>
            </program>
        </programs>
    </application>

</root>
```

12.4.2 Control & Status

The operational status of **Scenario 3** required by the Requirement 2 is made available through:

- the *Application Page* of the **Supvisors** Web UI, as a LED near the application state,
- the *XML-RPC API* (example below),
- the *REST API* (if **supvisorsflask** is started),
- the *Status* of the extended **supervisorctl** or **supvisorsctl** (example below),
- the *Event interface*.

For the examples, the following context applies:

- due to limited resources - 3 nodes are available (`rocky51`, `rocky52` and `rocky53`) -, each node hosts 2 **Supvisors** instances, one server and one console ;
- **common_data_bus** and **scen3_internal_data_bus** are *Unmanaged* so **Supvisors** always considers these 'applications' as STOPPED ;
- **scen3_srv** is distributed over the 3 servers ;
- **scen3_hci_01**,program:*scen3_hci_02*, program:*scen3_hci_03* have been respectively started on `console_1`, `console_2`, `console_3` .

The Supervisor configuration of the consoles has been changed to include the files related to the Supervisor identifier `console_X` rather than those related to `host_node_name`. As there is no automatic expansion related to the Supervisor identifier so far, an environmental variable is used.

```
# include section in supervisord_console.conf
[include]
files = common/*.ini console/*.ini console/%(ENV_IDENTIFIER)s/*.ini
```

```
>>> from supervisor.childutils import getRPCInterface
>>> proxy = getRPCInterface({'SUPERVISOR_SERVER_URL': 'http://localhost:61000'})
>>> proxy.supvisors.get_all_applications_info()
[{'application_name': 'common_data_bus', 'statecode': 0, 'statename': 'STOPPED', 'major_
↪failure': False, 'minor_failure': False},
{'application_name': 'scen3_internal_data_bus', 'statecode': 0, 'statename': 'STOPPED',
↪'major_failure': False, 'minor_failure': False},
{'application_name': 'scen3_srv', 'statecode': 2, 'statename': 'RUNNING', 'major_failure
↪': False, 'minor_failure': False},
{'application_name': 'scen3_hci_01', 'statecode': 2, 'statename': 'RUNNING', 'major_
↪failure': False, 'minor_failure': False},
{'application_name': 'scen3_hci_02', 'statecode': 2, 'statename': 'RUNNING', 'major_
↪failure': False, 'minor_failure': False},
{'application_name': 'scen3_hci_03', 'statecode': 2, 'statename': 'RUNNING', 'major_
↪failure': False, 'minor_failure': False}]
```

```
[bash] > supvisorsctl -s http://localhost:61000 application_info
Application              State     Major  Minor
common_data_bus          STOPPED   False  False
scen3_internal_data_bus  STOPPED   False  False
scen3_srv                RUNNING   False  False
scen3_hci_01             RUNNING   False  False
scen3_hci_02             RUNNING   False  False
scen3_hci_03             RUNNING   False  False
```


As a conclusion, all the requirements are met using **Supvisors** and without any impact on the application to be supervised. **Supvisors** improves application control and status.

12.5 Example

The full example is available in Supvisors Use Cases - Scenario 3.

Two versions are provided:

- one corresponding to the description above ;
- one based on the **Supvisors** discovery mode, using stereotypes rather than aliases.

CHANGE LOG

13.1 0.18 (2024-05-02)

- Refactoring of the **Supvisors** internal communications. The `internal_port` of the **Supvisors** section in the Supervisor configuration file is no longer needed. As a consequence, the `supvisors_list` option is simplified as follows: `<identifier>host_name:http_port`. The transitional `SupvisorsInstanceStates.ISOLATING` state has been removed. The remote **Supvisors** instance becomes `SILENT` as soon as the published events fails due to a transport issue.

- Implement Issue #50. A new tag `operational_status` in the Application rules allows to declare the formula applicable to evaluate the application operational status. `status_formula` is added to the result of the XML-RPC `get_application_rules`.

- Implement Issue #15. A `StarterModel` has been added to **Supvisors** to give a prediction of the application distribution when started. The command is available through the new XML-RPCs `test_start_application` and `test_start_process` and have been added to `supervisorctl`.

- The **Supvisors** `core_identifiers` option and the **Supvisors** rules can now accept indifferently Supervisor identifiers or keys with the `host:http_port` format.

- Update the `get_instance_info` XML-RPC so that the function accepts a stereotype as parameter. As a consequence, it now returns a list of dictionaries.

- Add a `lazy` attribute to the `update_numprocs` XML-RPC, so that when set combined to a numprocs decrease, **Supvisors** defers the obsolete processes deletion from the Supervisor configuration only when the processes stop (exit, crash or later user request) instead of stopping them immediately.

- Add monotonic time in internal model and exchanges to cope with time updates while **Supvisors** is running. Impact on the XML-RPC `get_instance_info`, `get_process_info` and on the event interface for instance status and process event.

- Add new `get_statistics_status`, `enable_host_statistics`, `enable_process_statistics`, `update_collecting_period` XML-RPCs to support the possibility to get and update the collection of host and process statistics. The corresponding commands `stats_status`, `enable_stats` and `stats_period` have been added to `supervisorctl`. The JAVA client includes the new XML-RPCs.

- Add new `get_all_inner_process_info` and `get_inner_process_info` XML-RPCs to support debug investigation. They return internal information on the processes declared on a **Supvisors** instance.

- Move the host statistics collector to the statistics collector process. The option `stats_collecting_period` is now applicable to host statistics collector.

- Re-apply the eventual process `extra_args` when restarting the application.

- In the Supervisors navigation menu of the Web UI, add a red light to Supervisor instances having raised a failure.

- Allow the display of a software name and icon at the top of the **Supvisors** Web UI. The options `software_name` and `software_icon` have been added to the **Supvisors** section of the Supervisor configuration file.

- All internal identifiers are now based on the `host:http_port` format.

- Rename the `DEPLOYMENT` state as `DISTRIBUTION` state to lift ambiguity ("deployment" is rather connoted when dealing with the orchestration domain).

- Rework **Supvisors** `RPCInterface` exceptions.

- Rework the Web UI.

13.2 0.17.4 (2024-03-24)

- Fix bug that was randomly blocking **Supvisors** on restart or shutdown, due to a stdout flush hanging in multi-processing bowels. The statistics Process is now started before any other thread.

- `guest` time removed from because CPU calculation because it is already accounted in `user` time on Linux.

- Fix process CPU so that it corresponds to the Linux `top` result.

- Use the latest versions of Sphinx-related modules for documentation, as `sphinx-5.0` is now the minimal dependency.

13.3 0.17.3 (2023-12-14)

- Fix regression when displaying the Network statistics in the **Supvisors** Web UI.

- Add restrictions to `flask-restx` dependencies.

13.4 0.17.2 (2023-12-04)

- Fix rare I/O exception by joining the `SupervisorsProxy` thread before exiting the `SupvisorsMainLoop`.

- Fix rare exception when host network statistics are prepared for display in the **Supvisors** Web UI in the event where network interfaces have different history sizes.

- Fix the **Supvisors** identifier possibilities when using the distribution rule `SINGLE_INSTANCE`.

- Update the process statistics collector thread so that it exits by itself when `supervisord` is killed.

- Improve the node selection when using the distribution rule `SINGLE_NODE`.

- Use an asynchronous server in the **Supvisors** internal communications. The refactoring fixes an issue with the TCP server that sometimes wouldn't bind despite the `SO_REUSEADDR` set.

- Restore the `action` class in the HTML of the **Supvisors** Web UI.

- CI targets added for Python 3.11 and 3.12.

13.5 0.17 (2023-08-17)

- Fix Issue #112. Write the disabilities file even if no call to `disable` and `enable` XML-RPCs have been done. Try to create the folder at startup if it does not exist.

- Fix a case where the `Starter` would block if the process reaches the expected state without reception of the corresponding event.

- Fix typo for `zmq` requirement when installing **Supvisors** from `pypi`.

- Fix `flask-restx` dependency in setup according to Python version.

- Fix uncaught exception the request to start a process is rejected due to a lack of resources. The exception was dependent from the Python version (absent in 3.6 but raised in 3.9).

- Monkeypatch fix of Supervisor Issue #1596. Shutdown of the asyncore socket before it is closed.

- Improve robustness against network failures. All Supervisor events are applied to the local **Supvisors** instance before they are published, so that it remains functional despite a network failure. The internal TCP sockets are rebound when a network interface becomes up (requires `psutil`).

- Provide a discovery mode where the **Supvisors** instances are added on-the-fly without declaring them in the `supvisors_list` option. The function relies on a Multicast Group definition (options `multicast_group`, `multicast_interface` and `multicast_ttl` added to that purpose). The attribute `discovery_mode` is added to the `get_state` and `get_instance_info` XML-RPCs.

- Add a new option `stereotypes` to support the discovery mode. The `identifiers` of the Application and Program rules can now reference a **Supvisors** stereotype in addition to identifiers and aliases. By extension, it is made available to the non-discovery mode.

- Add a new option `syncho_options` to enable the user to choose the conditions putting an end to the **Supvisors** synchronization phase. More particularly when using the new USER condition, the **Supvisors** Web UI provides a means to end the INITIALIZATION state, with optional *Master* selection. The command is also available as an XML-RPC `end_synchro` and has been added to `supervisorctl`.

- The new item `@` in the `identifiers` of the Program rules takes the behavior of the item `#` as it was before **Supvisors** version 0.13, i.e. the assignment is strictly limited by the length of the `identifiers` list, without roll-over. NOTE: This is not available for Application rules.

- Use host aliases when looking for the local **Supvisors** instance.

- Use IP address rather than host identification when dealing with SINGLE_NODE starting strategy.

- To prevent the situation that led the `Starter` to block, a new state CHECKED is added to `SupvisorsInstanceStates`, which is actually a pre-RUNNING state. Such a **Supvisors** instance is considered active and is updated with received events but cannot be part of any starting sequence until all starting jobs in progress are completed.

- Limit the consideration of the process forced state to display in the Application page of the **Supvisors** Web UI, so that it does not interfere with the real process state.

- Add `master_identifier` to the output of the XML-RPCs `get_supvisors_state` and `get_instances_info`. The `supervisorctl` commands `sstate` and `instance_status` have also been updated.

- Monkeypatch **Supervisor** on-the-fly so that its logger is thread-safe and add log traces in **Supvisors** threads.

- Simplify the **Supvisors** state machine and replace the states RESTART and SHUTDOWN by a single state FINAL.

- Highlight the process line hovered by the cursor in the **Supvisors** Web UI.

- Remove the figures from the **Supvisors** Web UI when `matplotlib` is not installed.

- Add RPC `changeLogLevel` to the JAVA client.

- Do not catch XmlRpc exceptions in the JAVA client.

- Refactoring of the **Supvisors** internal communications.

13.6 0.16 (2023-03-12)

- Add `websockets` as an option to the **Supvisors** event listener (Python 3.7+ only).

- Re-design the `PyZMQ` event listener using the `zmq.asyncio` support for better commonalities with the `wesockets` solution.

- Re-design the statistics collection and compilation.

- The option `stats_enabled` takes additional values to control host and process statistics independently.

- The option `stats_collecting_period` has been added to set the minimum time between process statistics collection.

- The option `stats_periods` accepts float values, not necessarily multiples of 5.

- Fix Issue #54. Add host and process statistics to the **Supvisors** event interface.

- Fix children process CPU times in statistics.

- Fix Solaris mode not taken into account for the process mean CPU value in the **Supvisors** Web UI.

- Fix Flask `start_args` to pass the extra arguments in the URL attributes rather than in the route.

- Only one **Supvisors** instance is running when both `unix_http_server` and `inet_http_server` sections are defined in the supervisor configuration file.

- The local **Supvisors** instance is identified as the item having the same fully qualified domain name (as returned by `socket.gethostaddr` and `socket.getfqdn`) among the items of the `supvisors_list` option.

- Use the HTTP server port to help the identification of the local **Supvisors** instance when multiple items of the `supvisors_list` option fit and identifier is not set.

- The attribute `process_failure` is added to the `get_instance_info` XML-RPC to inform if there is a process failure in the **Supvisors** instance. The attribute is also provided in the event interface and in the `instance_status` option of the `supervisorctl` command.

- Raise an exception when the matching **Supvisors** instance in the `supvisors_list` option is inconsistent with the local configuration.

- Add a **Supvisors** logo.

13.7 0.15 (2022-11-20)

- Publish / Subscribe pattern implemented for **Supvisors** internal communication. `PyZmq` is now only used for the optional external publication interface.

- Make **Supvisors** robust to `addProcessGroup` / `removeProcessGroup` / `reloadConfig` Supervisor XML-RPCs.

- Fix process CPU times in statistics so that children processes are all taken into account.

- Fix regression in `supervisorctl application_rules` where the former `distributed` entry was still used instead of `distribution`.

- Fix uncaught exception when an unknown host name or IP address is used in the `supvisors_list` option.

- Fix `ProcessEvent` publication when no resource is available to start a process.

- Fix `SupvisorsStatus` event in JAVA ZMQ client.

- Manage the `RuntimeError` exception that could be raised by matplotlib when saving a graph.

- Add `all_start` and `all_start_args` to the list of `supervisorctl` commands. These commands respectively invoke `supervisor.startProcess` and `supvisors.start_args` on all running **Supvisors** instances.

- Add `tail_limit` and `tailf_limit` options to override the default 1024 bytes used by Supervisor to display the Tail pages of the Web UI.

- Inactive Log Clear / Stdout / Stderr buttons in the Web UI if no stdout / stderr is configured.

- Add resolution to `ProcessStatus` time information and store event time, so that forced state is correctly considered.

- A process is not considered disabled anymore when process rules don't allow any candidate **Supvisors** instance.

- When `psutil` is not installed on a host, the statistics-related options of the Process and Host pages of the Web UI are not displayed, just as if the option `stats_enabled` was set to `False`.

- Clarify the exceptions that could be raised in **Supvisors** startup.

- Add a FAQ to the documentation.

13.8 0.14 (2022-05-01)

- Implement Supervisor Issue #1054. Start / Stop / Restart buttons have been added to groups in the Supervisor page of the Web UI so that it is possible to start / stop / restart all the processes of the group at once. The application state and description have been removed from this table as the information was confusing.

- Fix issue where starting strategies would not work as expected when multiple **Supvisors** instances run on the same node but their `host_name` is identified differently in the option `supvisors_list`.

- Replace on-the-fly the Supervisor `gettags` function so that the XML-RPC `system.methodSignature` works with both Supervisor and **Supvisors**.

- Use `socket.gethostaddr` to validate the host names provided in the option `supvisors_list`.

- In the Application page of the Web UI, apply a *disabled* status to programs that are disabled on all their possible **Supvisors** instances (according to rules and configuration).

- Maintain the auto-refresh set on the **Supvisors** `restart` / `shutdown` actions of the Web UI.

- Change the style of the *matplotlib* graphs.

13.9 0.13 (2022-02-27)

- Implement Supervisor Issue #591. It is possible to disable/enable programs using the new `disable` and `enable` XML-RPCs. A new option `disabilities_file` has been added to support the persistence. The `disabled` status of the processes is made available through the `supvisors.get_local_process_info` XML-RPC and in the process table of the Web UI.

- Fix issue where **Supvisors** may be blocked in the `DEPLOYMENT` phase due to late process events.

- Add a new `start_any_process` XML-RPC that starts one process whose namespec matches the regular expression.

- Add a `wait` parameter to the `update_numprocs` XML-RPC.

- Add the principle of **Supvisors** modes to the output of the XML-RPCs `get_supvisors_state` and `get_instance_info`. The modes are linked to the existence of jobs in progress in `Starter` and `Stopper`.

- The **Supvisors** modes are displayed to the Main page of the Web UI and the **Supvisors** instance modes are displayed to the Process and Host pages of the Web UI. In the navigation menu, the local **Supvisors** instance points out the **Supvisors** instances where the modes are activated, and the applications involved in its own `Starter` or `Stopper`.

- When using the item `#` in the `identifiers` of the Application or Program rules and with a number of candidate applications or processes greater than the candidate `identifiers`, the assignment is performed by rolling over the `identifiers` list.

- Add pid and uptime information to the `supervisord` entry of the process table in the Web UI.

- The application rules of a **Supvisors** rules file can be inserted in any order.

- Protect the Supervisor thread against any exception that could be raised by **Supvisors** when processing a Supervisor event.

- Provide a Flask server that can be added as a Supervisor program to interact with **Supvisors** using a REST API.

- Update the CSS style of the inactive buttons in the Web UI.

- Fix CSS resources table cell height with recent versions of Firefox.

- Update the Web UI to allow multiple processes per line in the **Supvisors** instance boxes.

- Remove support to deprecated option `distributed` and to the possibility to have the `program` element directly under the `application element` in a **Supvisors** rules file.

13.10 0.12 (2022-01-26)

- Fix crash following a `supervisorctl update` as the group added doesn't include `extra_args` and `command_ref` attributes in the Supervisor internal structure.

- Fix crash when the state of the **Supvisors** master is received before any **Supvisors** instance has been confirmed.

- Fix crash when receiving process state events from a **Supvisors** instance that has been checked while it was in a `RESTARTING` state.

- Fix regression in **Supvisors** restarting / shutting down as the *Master* would actually restart / shut down before notifying the other **Supvisors** instances of its state. The new **Supvisors** state `RESTART` has been introduced.

- Add `supervisord` entry to the process table of the **Supvisors** instance in the Web UI. This entry provides process statistics and the possibility to view the Supervisor logs.

- Fix issue in Web UI with the Solaris mode not applied to the process CPU plot.

- Fix CSS for Supvisors instance boxes (table headers un-stickied) in the Main page of the Web UI.

- Fix process children CPU times counted twice in statistics.

- Add regex support to the `pattern` attribute of the `application` and `program` elements of the **Supvisors** rules file.

- The `distribution` option has been added to replace the `distributed` option in the **Supvisors** rules file. The `distributed` option is deprecated and will be removed in the next version.

- Update the starting strategies so that the node load is considered in the event where multiple **Supvisors** instances are running on the same node. The `LESS_LOADED_NODE` and `MOST_LOADED_NODE` starting strategies have been added.

- Update the `RunningFailureHandler` so that `Starter` and `Stopper` actions are all stored before they are actually triggered.

- Add the `RESTART` and `SHUTDOWN` strategies to the `running_failure_strategy` option.

- Update `Starter` and `Stopper` so that event timeouts are based on ticks rather than time.

- Update `InfanticideStrategy` and `SenicideStrategy` so that the conciliation uses the `Stopper`. This avoids duplicated conciliation requests when entering the `CONCILIATION` state.

- When receiving a forced state due to a `Starter` or `Stopper` timeout, check if the expected process state has been reached before actually forcing the state. Events may have crossed.

- The `programs` section has been added in the `application` section of the **Supvisors** rules file. All `program` definitions should be placed in this section rather than directly in the `application` section. The intention is for the next **Supvisors** version to be able to declare application options in any order. Note that having `program` sections directly in the `application` section is still supported but deprecated and will be removed in the next version.

- Add the `starting_failure_strategy` option in the `program` section of the **Supvisors** rules file. It supersedes the values eventually set in the `application` section.

- Add the `inactivity_ticks` option to the **Supvisors** section of the Supervisor configuration file to enable more flexibility in a congested system.

- Add `node_name` and `port` information to the result of the `get_instance_info` XML-RPC and to the instance status of the **Supvisors** event listener.

- In the Process page of the Web UI, add buttons to shrink / expand all applications.

- Use a different gradient in the Web UI for running processes that have ever crashed.

- Fix CSS process table cell height with recent versions of Firefox.

- Use hexadecimal strings for the `shex` attribute in the Web UI URL.

- Add `action` class to the start/stop/restart/shutdown buttons in the headers of the **Supvisors** web pages.

- Move PyZmq sockets creation to the main thread so that a bind error is made explicit in log traces.

- Remove support to deprecated options, attributes and XML-RPCs (`address_list`, `force_synchro_if`, `rules_file`, `address_name`, `addresses`, `get_master_address`, `get_address_info` and `get_all_addresses_info`).

13.11 0.11 (2022-01-02)

- Fix Issue #99. Update the **Supvisors** design so that it can be used to supervise multiple Supervisor instances on multiple nodes. This update had a major impact on the source code. More particularly:

    - The XML-RPCs `get_master_identifier`, `get_instance_info` and `get_all_instances_info` have been added to replace `get_master_address`, `get_address_info` and `get_all_addresses_info`.

    - The `supervisorctl` command `instance_status` has been added to replace `address_status`.

    - The XML-RPCs that would return attributes `address_name` and `addresses` are now returning `identifier` and `identifiers` respectively. This impacts the following XML-RPCs (and related `supervisorctl` commands):

        * `get_application_info`

        * `get_all_application_info`

* get_application_rules

* get_address_info

* get_all_addresses_info

* get_all_process_info

* get_process_info

* get_process_rules

* get_conflicts.

- The `supvisors_list` option has been added to replace `address_list` in the **Supvisors** section of the Supervisor configuration file. This option accepts a more complex definition: `<identifier>host_name:http_port:internal_port`. Note that the simple `host_name` is still supported in the event where **Supvisors** doesn't have to deal with multiple Supervisor instances on the same node.

- The `core_identifiers` option has been added to replace `force_synchro_if` in the **Supvisors** section of the Supervisor configuration file. It targets the names deduced from the `supvisors_list` option.

- The `identifiers` option has been added to replace the `addresses` option in the **Supvisors** rules file. This option targets the names deduced from the `supvisors_list` option.

- The `address`-like attributes, XML-RPCs and options are deprecated and will be removed in the next version.

* Fix Issue #98. Move the heartbeat emission to the Supvisors thread to avoid being impacted by a Supervisor momentary freeze. On the heartbeat reception part, consider that the node is `SILENT` based on a number of ticks instead of time.

* Fix issue with `supvisors.stop_process` XML-RPC that wouldn't stop all processes when any of the targeted processes is already stopped.

* Fix exception when authorization is received from a node that is not in `CHECKING` state.

* Fix regression (missing disconnect) on node isolation when fencing is activated.

* Fix issue in statistics compiler when network interfaces are dynamically created / removed.

* Refactoring of `Starter` and `Stopper`.

* The module `rpcrequests` has been removed because useless. The function `getRPCInterface` of th module `supervisor.childutils` does the job.

* The `startsecs` and `stopwaitsecs` program options have been added to the results of `get_all_local_process_info` and `get_local_process_info`.

* The option `rules_file` is updated to `rules_files` and supports multiple files for **Supvisors** rules. The option `rules_file` is thus deprecated and will be removed in the next version.

* Add a new `restart_sequence` XML-RPC to trigger a full application start sequence.

* Update the `restart_application` and `restart_process` XML-RPC so that processes can restart themselves using them.

* Add `expected_exit` to the output of `supervisorctl sstatus` when the process is `EXITED`.

* Add the new option `stats_enabled` to enable/disable the statistics function.

* Update `start_process`, `stop_process`, `restart_process`, `process_rules` in `supervisorctl` so that calls are made on each individually process rather than process group when `all` is used as parameter.

* Add exit codes to erroneous **Supvisors** calls in `supervisorctl`.

- When aborting jobs when re-entering the `INITIALIZATION` state, clear the structure holding the jobs in progress. It has been found to stick **Supvisors** in the `DEPLOYMENT` state in the event where the *Master* node is temporarily `SILENT`.

- Restrict the use of the XML-RPCs `start_application`, `stop_application`, `restart_application` to *Managed* applications only.

- Review the logic of the refresh button in the Web UI.

- Add node time to the node boxes in the Main page of the Web UI.

- Sort alphabetically the entries of the application menu of the Web UI.

- Update the mouse pointer look on nodes in the Main and Host pages of the Web UI.

- Remove the useless timecode in the header of the Process and Host pages of the Web UI as it is now provided at the bottom right of all pages.

- Add class "action" to Web UI buttons that trigger an XML-RPC.

- Switch from Travis-CI to GitHub Actions for continuous integration.

13.12 0.10 (2021-09-05)

- Implement Supervisor Issue #177. It is possible to update dynamically the program numprocs using the new `update_numprocs` XML-RPC.

- Add targets **Python 3.7** and **Python 3.8** to Travis-CI.

13.13 0.9 (2021-08-31)

- Enable the hash '#' for the `addresses` of a non-distributed application.

- Add `supvisorsctl` to pally the lack of support of `supervisorctl` when used with `--serverurl` URL option. See related Supervisor Issue #1455.

- Provide `breed.py` as a binary of **Supvisors**: `supvisors_breed`. The script only considers group duplication as it is fully valid to include multiple times a program definition in several groups.

- Move the contents of the `[supvisors]` section into the `[rpcinterface:supvisors]` section and benefit from the configuration structure provided by Supervisor. The `[supvisors]` section itself is thus obsolete.

- Remove deprecated support of `pattern` elements.

- Fix issue when using the Web UI Application page from a previous launch.

- Invert the stop sequence logic, starting from the greatest `stop_sequence` number to the lowest one.

- When `stop_sequence` is not set in the rules files, it is defaulted to the `start_sequence` value. With the new stop sequence logic, the stop sequence is by default exactly the opposite of the start sequence.

- Fix Nodes' column width for `supervisorctl application_rules`.

- `CHANGES.rst` replaced with `CHANGES.md`.

- 'Scenario 3' has been added to the **Supvisors** use cases.

- A 'Gathering' configuration has been added to the **Supvisors** use cases. It combines all uses cases.

13.14 0.8 (2021-08-22)

- Fix exception in `INITIALIZATION` state when the *Master* declared by other nodes is not RUNNING yet and the *core nodes* are RUNNING.

- Fix exception when program rules and extra arguments are tested against a program unknown to the local Supervisor.

- Fix issue about program patterns that were applicable to all elements. The scope of program patterns is now limited to their owner application.

- Fix issue with infinite tries of application restart when the process cannot be started due to a lack of resources and `RESTART_APPLICATION` is set on the program in the **Supvisors** rules file.

- Fix issue about application state not updated after a node has become silent.

- Fix issue when choosing a node in `Starter`. Apply the requests that have not been satisfied yet for non-distributed applications.

- Review logic for application major / minor failures.

- Simplify the insertion of applications to start or stop in Commander jobs.

- Add support for application patterns in the **Supvisors** rules file.

- In the **Supvisors** rules file, `pattern` elements are **deprecated** and are replaced by `program` elements with a `pattern` attribute instead of a `name` attribute. Support for `pattern` elements will be removed in the next version of **Supvisors**.

- Node aliases have been added to the **Supvisors** rules file.

- Add the current node to all pages of Web UI to be aware of the node that displays the page.

- The Web UI is updated to handle a large list of applications, nodes, processor cores and network interfaces.

- In the Process page of the Web UI, expand / shrink actions are not applicable to programs that are not owned by a Supervisor group.

- In the Applications navigation menu of the Web UI, add a red light near the Applications title if any application has raised a failure.

- In the Application page of the Web UI, default starting strategy is the starting strategy defined in the **Supvisors** rules file for the application considered.

- In the Application ang Process page, the detailed process statistics can be deselected.

- Titles added to the output of :program:supervisorctl `address_status` and `application_info`.

- The XML schema has been moved to a separate file `rules.xsd`.

- Remove dependency to *netifaces* as *psutil* provides the function.

- 'Scenario 2' has been added to the **Supvisors** use cases.

- A script `breed.py` has been added to the installation package. It can be used to duplicate the applications based on a template configuration and more particularly used to prepare the Scenario 2 of the **Supvisors** use cases.

13.15 0.7 (2021-08-15)

- Fix Issue #92. The *Master* drives the state of all **Supvisors** instances and a simplified state machine has been assigned to non-master **Supvisors** instances. The loss of the *Master* instance is managed in all relevant states.

- Fix issue about applications that would be started automatically whereas their `start_sequence` is 0. The regression has been introduced during the implementation of applications repair in **Supvisors 0.6**.

- Enable stop sequence on *Unmanaged* applications.

- In the Applications navigation menu of the Web UI, add a red light to applications having raised a failure.

- New application rules `distributed` and `addresses` added to the **Supvisors** rules file. Non-distributed applications have all their processes started on the same node chosen in accordance with the `addresses` and the `starting_strategy`.

- Add the `starting_strategy` option to the `application` section of the **Supvisors** rules file.

- Fix issue when choosing a node in `Starter`. The starting strategies considers the current load of the nodes and includes the requests that have not been satisfied yet.

- Fix issue with infinite process restart when the process crashes and `RESTART_PROCESS` is set on the program in the **Supvisors** rules file. When the process crashes, only the *Supervisor* `autorestart` applies. The **Supvisors** `RESTART_PROCESS` applies only when the node becomes inactive.

- Fix exception when forcing the state on a process that is unknown to the local Supervisor.

- Promote the `RESTART_PROCESS` into `RESTART_APPLICATION` if the application is stopped.

- For the *Master* election, give a priority to nodes declared in the `forced_synchro_if` option if used.

- When using the `forced_synchro_if` option and when `auto_fence` is activated, do not isolate nodes as long as `synchro_timeout` has not passed.

- In the `INITALIZATION` state, skip the synchronization phase upon notification of a known *Master* and adopt it.

- Add reciprocity to isolation even if `auto_fence` is not activated.

- In the process description of the Web UI Application page, add information about the node name. In particular, it is useful to know where the process was running when it is stopped.

- Start adding use cases to documentation, inspired by real examples. 'Scenario 1' has been added.

13.16 0.6 (2021-08-01)

- Applications that are not declared in the **Supvisors** rules file are not *managed*. *Unmanaged* applications have no start/stop sequence, no state and status (always STOPPED) and **Supvisors** does not raise a conflict if multiple instances are running over multiple nodes.

- Improve **Supvisors** stability when dealing with remote programs undefined locally.

- Add expand / shrink actions to applications to the `ProcInstanceView` of the Web UI.

- Upon authorization of a new node in **Supvisors**, back to `DEPLOYMENT` state to repair applications.

- Add RPC `change_log_level` to dynamically change the **Supvisors** logger level.

- Application state is evaluated only against the starting sequence of its processes.

- Fix blocking issue when *Master* is stopped while in `DEPLOYMENT` state.

- Fix issue with applications that would not fully stop when using the `STOP_APPLICATION` starting failure strategy.

- Fix issue related to Issue #85. An exception was raised when the program `procnum` was greater than the list of applicable nodes.

- Fix Issue #91. Fix CSS style on the process tables in HTML.

- Fix Issue #90. The **Supvisors** *Master* node drives the transition to `OPERATION`.

- In the Web UI, set the process state color to `FATAL` when the process has exited unexpectedly.

- Change the default expected loading to `0` in the `program` section of the **Supvisors** rules file.

- Python `Enum` used for enumerations (not available in Python 2.7).

- Remove `supvisors_shortcuts` from source code to get rid of IDE warnings.

- All unit tests updated from `unittest` to `pytest`.

- Include this Change Log to documentation.

13.17 0.5 (2021-03-01)

- New option `force_synchro_if` to force the end of the synchronization phase when a subset of nodes is active.

- New starting strategy `LOCAL` added to command the starting of an application on the local node only.

- Fix Issue #87. Under particular circumstances, **Supvisors** could have multiple *Master* nodes.

- Fix Issue #86. The starting and stopping sequences may fail and block when a sub-sequence includes only failed programs.

- Fix Issue #85. When using `#` in the `address_list` program rule of the **Supvisors** rules file, a subset of nodes can optionally be defined.

- Fix Issue #84. In the **Supvisors** rules file, program rules can be defined using both model reference and attributes.

- The Web UI uses the default starting strategy of the configuration file.

- The layout of Web UI statistics sections has been rearranged.

- Fix CSS style missing for `CHECKING` node state in tables.

- Star added to the node box of the *Master* instance on the main page.

- Type annotations are added progressively in source code.

- Start switching from `unittest` to `pytest`.

- Logs (especially `debug` and `trace`) updated to remove printed objects.

13.18 0.4 (2021-02-14)

- Auto-refresh button added to all pages.

- Web UI Main page reworked by adding a subdivision of application in node boxes.

- Fix exception when exiting using `Ctrl+c` from shell.

- Fix exception when rules files is not provided.

13.19 0.3 (2020-12-29)

- Fix Issue #81. When **Supvisors** logfile is set to `AUTO`, **Supvisors** uses the same logger as **Supervisor**.
- Fix Issue #79. When `FATAL` or `UNKNOWN` Process state is forced by **Supvisors**, `spawnerr` was missing in the listener payload.
- Useless folder `rsc_ref` deleted.
- `design` folder moved to a dedicated *GitHub* repository.
- 100% coverage reached in unit tests.

13.20 0.2 (2020-12-14)

- Migration to **Python 3.6**. Versions of dependencies are refreshed, more particularly **Supervisor 4.2.1**.
- CSS of Web UI updated / simplified.
- New action added to Host Process page of WebUI: `tail -f stderr` button.
- New information actions added to Application page of WebUI:
    - `description` field.
    - `clear logs`, `tail -f stdout`, `tail -f stderr` buttons.
- Fix Issue #75. **Supvisors** takes into account the `username` and the `password` of `inet_http_server` in the `supervisord` section.
- Fix Issue #17. The user selections on the web UI are passed to the URL.
- Fix Issue #72. The extra arguments are shared between all **Supvisors** instances.
- `README.rst` replaced with `README.md`.
- Coverage improved in tests.
- Docs target added to Travis-CI.

13.21 0.1 (2017-08-11)

Initial release.

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S