
developer.skatelescope.org

Documentation

Release 0.6.3

Marco Bartolini

Mar 13, 2024

CONTENTS

1	Development	3
2	Functionality	5
3	Receive Process and Port Configuration	7
4	API	9
5	Indices and tables	17
	Python Module Index	19
	Index	21

The SDP scripting library is a high-level interface for writing processing scripts. Its goal is to provide abstractions to enable the developer to express the high-level organisation of a processing script without needing to interact directly with the low-level interfaces such as the SDP configuration library.

DEVELOPMENT

1.1 Installation

The library can be installed using `pip` but you need to make sure to use the SKA artefact repository as the index:

```
pip install \  
  --index-url https://artefact.skao.int/repository/pypi-all/simple \  
  ska-sdp-scripting
```

To install it using a `requirements.txt` file, the `pip` options can be added to the top of the file like this:

```
--index-url https://artefact.skao.int/repository/pypi-all/simple  
ska-sdp-scripting
```

1.2 Usage

Once the SDP scripting library has been installed, use:

```
import ska_sdp_scripting
```

1.3 Develop a new script

The steps to develop and test an SDP processing script can be found at [Script Development](#).

FUNCTIONALITY

The required functionality of the scripting library is as follows.

2.1 Starting, monitoring and ending a script

2.1.1 At the start

- Claim the processing block.
- Get the parameters defined in the processing block. They should be checked against the parameter schema defined for the script.

2.1.2 Resource requests

- Make requests for input and output buffer space. The script will calculate the resources it needs based on the parameters, then request them from the processing controller. This is currently a placeholder.

2.1.3 Declare script phases

- Scripts will be divided into phases such as preparation, processing, and clean-up. In the current implementation, only one phase can be declared, which we refer to as the ‘work’ phase.

2.1.4 Execute the work phase

- On entry to the work phase, it waits until the resources are available. Meanwhile it monitors the processing block to see it has been cancelled. For real-time scripts, it also checks if the execution block has been cancelled.
- Deploys execution engines to execute a script/function.
- Monitors the execution engines and processing block state. Waits until the execution is finished, or the processing block is cancelled.
- Continuously updates the processing block state with the status of execution engine deployments. It provides aggregate information about these statuses to inform other components about the readiness of deployments.

2.1.5 At the end

- Remove the execution engines to release the resources.
- Update processing block state with information about the success or failure of the script.

2.2 Receive scripts

- Get IP and MAC addresses for the receive processes.
- Monitor receive processes. If any get restarted, then the addresses may need to be updated.
- Write the addresses in the appropriate format into the processing block state.

2.3 Compatibility with the telescope model library

We keep the scripting library compatible with the latest version of the [telescope model library](#). If you use a configuration string that is based on an older version of the telescope model, you may experience errors or unexpected behaviour.

RECEIVE PROCESS AND PORT CONFIGURATION

3.1 Multiple Port Configuration

The scripting library has the capability to configure multiple ports. This allows to deploy a single receiver with multiple ports. In another word, this will allow a single receiver to receive data for a single SPEAD stream coming from multiple processes.

Now, assuming each sender sends data for 1 channel and all baselines, then we'll want to have as many ports as channels on the receiver side. For cbf-receive, a single receiver process can receive on multiple ports already, and this is configurable via `reception.receiver_port_start` and `reception.num_ports`.

To make sense of multiple ports, the port map was required to be updated from a three-value list (ADR-10) to a four-value list. The four value defines the increment of the port number.

For example, if we set `reception.receiver_port_start = 9000` and `reception.num_ports = 3`, `count= 3`, and `max_channels=1` then the resulting `port_map` would look like:

```
"port": [[0, 9000, 1, 0], [1, 9001, 1, 1], [2, 9002, 1, 2]]
```


4.1 Processing block

class `ska_sdp_scripting.processing_block.ProcessingBlock`(*pb_id*: *str* = *None*)

Claim the processing block.

Parameters

pb_id(*str*, *optional*) – processing block ID

configure_recv_processes_ports(*scan_types*, *max_channels_per_process*, *port_start*,
channels_per_port)

Calculate how many receive process(es) and ports are required, And configure a dictionary to be fed back into the `receive_addresses` attribute.

Parameters

- **scan_types** – scan types from EB
- **max_channels_per_process** – maximum number of channels per process
- **port_start** – starting port the receiver will be listening in
- **channels_per_port** – number of channels to be sent to each port

Returns

tuple(configured receive dict, number of processes)

Return type

tuple

create_phase(*name*: *str*, *requests*: *List*[*BufferRequest*]) → *Phase*

Create a script phase for deploying execution engines.

The phase is created with a list of resource requests which must be satisfied before the phase can start executing. For the time being the only resource requests are (placeholder) buffer reservations, but eventually this will include compute requests too.

Parameters

- **name** (*str*) – name of the phase
- **requests** (list of *BufferRequest*) – resource requests

Returns

the phase

Return type

Phase

exit()

Perform clean-up.

get_dependencies()

Get the list of processing block dependencies.

Returns

processing block dependencies

Return type

`list`

get_parameters(*schema=None*)

Get script parameters from processing block.

The schema checking is not currently implemented.

Parameters

schema – schema to validate the parameters

Returns

processing block parameters

Return type

`dict`

get_scan_types() → `List[str]`

Get scan types from the execution block.

Updates the scan types with the default parameters and channels.

This is only supported for real-time scripts

Returns

scan types

Return type

`list`

nested_parameters(*parameters: dict*)

Convert flattened dictionary to nested dictionary.

Parameters

parameters – parameters to be converted

Returns

nested parameters

receive_addresses(*configured_host_port, chart_name=None, service_name=None, namespace=None, script='vis-receive'*)

Generate receive addresses and update the processing block state.

Parameters

- **configured_host_port** – constructed host and port
- **chart_name** – Name of the statefulset
- **service_name** – Name of the headless service
- **namespace** – namespace where it's going to be deployed
- **script** – processing script that is configuring receive addresses options: vis-receive, pointing-offset; default: vis-receive

static request_buffer(*size: float, tags: List[str]*) → *BufferRequest*

Request a buffer reservation.

This returns a buffer reservation request that is used to create a script phase. These are currently only placeholders.

Parameters

- **size** (*float*) – size of the buffer
- **tags** (*list of str*) – tags describing the type of buffer required

Returns

buffer reservation request

Return type

BufferRequest

update_parameters(*default_parameters: dict, parameters: dict | Mapping*)

Nested overwrite of default_parameter values with ones in parameters.

Parameters

- **default_parameters** –
 dict
 default parameter values
- **parameters** –
 dict
 script specific parameters

Returns

processing block additional parameters

Return type

dict

4.2 Buffer request

class *ska_sdp_scripting.buffer_request.BufferRequest*(*size: float, tags: List[str]*)

Request a buffer reservation.

This is currently just a placeholder.

Parameters

- **size** (*float*) – size of the buffer
- **tags** (*list of str*) – tags describing the type of buffer required

size: *float*

tags: *List[str]*

4.3 Script phase

class `ska_sdp_scripting.phase.Phase`(*name*: *str*, *list_requests*: *List*, *config*: *ska_sdp_config.Config*, *pb_id*: *str*, *eb_id*: *str*, *script_kind*: *str*, *qc_config*: *str*)

Script phase.

This should not be created directly, use the `ProcessingBlock.create_phase()` method instead.

Parameters

- **name** (*str*) – name of the phase
- **list_requests** (*list*) – list of requests
- **config** (*ska_sdp_config.Config*) – SDP configuration client
- **pb_id** (*str*) – processing block ID
- **eb_id** (*str*) – execution block ID
- **script_kind** (*str*) – script kind
- **qc_config** (*str*) – Queue Connector config string

check_state(*txn*: *ska_sdp_config.config.Transaction*, *check_realtime*: *bool* = *True*) → *None*

Check the state of the processing block.

Check if the PB is finished or cancelled, and for real-time scripts check if the EB is finished or cancelled.

Parameters

- **txn** (*ska_sdp_config.Transaction*) – SDP configuration transaction
- **check_realtime** (*bool*) – Whether to check execution block state if the processing block is realtime (i.e. cancel processing script for FINISHED/CANCELLED)

ee_deploy_dask(*name*: *str*, *n_workers*: *int*, *func*: *Callable*, *f_args*: *Tuple*[*Any*])

Deploy a Dask execution engine.

Parameters

- **name** (*str*) – deployment name
- **n_workers** (*int*) – number of Dask workers
- **func** (*function*) – function to execute
- **f_args** (*tuple*) – function arguments

Returns

Dask execution engine deployment

Return type

DaskDeploy

ee_deploy_helm(*deploy_name*: *str*, *values*: *dict* | *None* = *None*)

Deploy a Helm execution engine.

This can be used to deploy any Helm chart.

Parameters

- **deploy_name** (*str*) – name of Helm chart
- **values** (*dict*, *optional*) – values to pass to Helm chart

Returns

Helm execution engine deployment

Return type

HelmDeploy

ee_deploy_test(*deploy_name*: *str*, *func*: *Callable* = *None*, *f_args*: *List[Any]* = *None*) → *EEDeploy*

Deploy a fake execution engine.

This is used for testing and example purposes.

Parameters

- **deploy_name** (*str*) – deployment name
- **func** (*function*) – function to execute
- **f_args** (*tuple*) – function arguments

Returns

fake execution engine deployment

Return type

FakeDeploy

ee_remove() → *None*

Remove execution engines deployments.

is_eb_finished(*txn*: *ska_sdp_config.config.Transaction*) → *bool*

Check if the EB is finished or cancelled.

Parameters

txn (*ska_sdp_config.Transaction*) – config db transaction

Return type

bool

monitor_deployments(*txn*: *ska_sdp_config.config.Transaction*, *iteration*: *int* = 0)

Monitor deployments, update the deployment status in the processing block state based on the deployments' pods' state.

Also update the deployments_ready pb state key.

At the moment deployments_ready = True only if all deployments are RUNNING. Else, it is False.

Parameters

- **txn** – Transaction object (config.txn)
- **iteration** – number of txn iteration

update_pb_state(*status*: *ProcessingBlockStatus* = *ProcessingBlockStatus.UNSET*)

Update processing block state.

If the status is UNSET, it is marked as finished.

Parameters

status (*str*, *optional*) – status

wait_loop(*func*: *Callable[[ska_sdp_config.config.Transaction], bool]*, *time_to_ready*: *int* = 0)

Wait loop to check the status of the processing block. It also updates the processing block state with deployment statuses for realtime scripts.

Parameters

- **func** – function to check condition for exiting the watcher loop
- **time_to_ready** – set deployments_ready to true after this amount of time has passed (seconds). Only for deployments deployed with phase.ee_deploy_test

4.4 Execution engine deployment

class `ska_sdp_scripting.ee_base_deploy.EEDeploy(pb_id: str, config: ska_sdp_config.Config)`

Base class for execution engine deployment.

Parameters

- **pb_id** (*str*) – processing block ID
- **config** (*ska_sdp_config.Client*) – SDP configuration client

get_id() → *str*

Get the deployment ID.

Returns

deployment ID

Return type

str

is_finished(*txn: ska_sdp_config.config.Transaction*) → *bool*

Check if the deployment is finished.

Parameters

txn (*ska_sdp_config.Transaction*) – configuration transaction

Return type

bool

remove(*deploy_id: str*) → *None*

Remove the execution engine.

Parameters

deploy_id (*str*) – deployment ID

update_deploy_status(*status: str*) → *None*

Update deployment status.

Parameters

status (*str*) – status

4.5 Helm EE Deployment

class `ska_sdp_scripting.helm_deploy.HelmDeploy(pb_id: str, config: ska_sdp_config.Config,
deploy_name: str, values: dict = None)`

Deploy Helm execution engine.

This should not be created directly, use the `Phase.ee_deploy_helm()` method instead.

Parameters

- **pb_id** (*str*) – processing block ID

- **config** (*ska_sdp_config.Config*) – SDP configuration client
- **deploy_name** (*str*) – name of Helm chart to deploy
- **values** (*dict*, *optional*) – values to pass to Helm chart

4.6 Dask EE deployment

```
class ska_sdp_scripting.dask_deploy.DaskDeploy(pb_id: str, config: ska_sdp_config.Config,
                                              deploy_name: str, n_workers: int, func: Callable,
                                              f_args: Tuple[Any])
```

Deploy a Dask execution engine.

The function when called with the arguments should return a Dask graph. The graph is then executed by calling the compute method:

```
result = func(*f_args)
result.compute()
```

This happens in a separate thread so the constructor can return immediately.

This should not be created directly, use the `Phase.ee_deploy_dask()` method instead.

Parameters

- **pb_id** (*str*) – processing block ID
- **config** (*ska_sdp_config.Client*) – configuration DB client
- **deploy_name** (*str*) – deployment name
- **n_workers** (*int*) – number of Dask workers
- **func** (*function*) – function to execute
- **f_args** (*tuple*) – function arguments

4.7 Fake EE deployment

```
class ska_sdp_scripting.fake_deploy.FakeDeploy(pb_id: str, config: ska_sdp_config.Config,
                                              deploy_name: str, func: Callable = None, f_args:
                                              Tuple[Any] = None)
```

Deploy a fake execution engine.

The function is called with the arguments in a separate thread so the constructor can return immediately.

This should not be created directly, use the `Phase.ee_deploy_test()` method instead.

Parameters

- **pb_id** (*str*) – processing block ID
- **config** (*ska_sdp_config.Client*) – SDP configuration client
- **deploy_name** (*str*) – deployment name
- **func** (*function*) – function to execute
- **f_args** (*tuple*) – function arguments

set_deployments_ready(*time_to_ready*: *int* = 0)

Set deployments_ready to True

Parameters

time_to_ready – set deployments_ready to true after this amount of time has passed (seconds)

4.8 Queue connector functions

Common functions used to configure the QueueConnector

`ska_sdp_scripting.queue_connector.configure_queue_connector(txn, qc_config, pb_id, eb_id)`

Function to configure queue connector.

Parameters

- **qc_config** – QueueConnector configuration string
- **txn** – Transaction
- **pb_id** – Processing block ID
- **eb_id** – Execution block ID

Returns

`ska_sdp_scripting.queue_connector.tear_down_queue_connector(config, qc_config_path)`

Function for tearing down the QueueConnector

Parameters

- **config** – SDP configuration client
- **qc_config_path** – QueueConnector config path

`ska_sdp_scripting.queue_connector.wait_for_queue_connector_state(config, qc_config_path, expected_state, timeout)`

Function to wait for queue connector state.

Parameters

- **config** – SDP configuration client
- **qc_config_path** – Path of the QueueConnector configuration (str)
- **expected_state** – Expected state the QueueConnector needs to reach (str)
- **timeout** – How much time to elapse before TimeoutError

INDICES AND TABLES

- `genindex`

PYTHON MODULE INDEX

S

`ska_sdp_scripting.queue_connector`, [16](#)

INDEX

B

BufferRequest (class in *ska_sdp_scripting.buffer_request*), 11

C

check_state() (*ska_sdp_scripting.phase.Phase* method), 12

configure_queue_connector() (in module *ska_sdp_scripting.queue_connector*), 16

configure_recv_processes_ports() (*ska_sdp_scripting.processing_block.ProcessingBlock* method), 9

create_phase() (*ska_sdp_scripting.processing_block.ProcessingBlock* method), 9

D

DaskDeploy (class in *ska_sdp_scripting.dask_deploy*), 15

E

ee_deploy_dask() (*ska_sdp_scripting.phase.Phase* method), 12

ee_deploy_helm() (*ska_sdp_scripting.phase.Phase* method), 12

ee_deploy_test() (*ska_sdp_scripting.phase.Phase* method), 13

ee_remove() (*ska_sdp_scripting.phase.Phase* method), 13

EEDeploy (class in *ska_sdp_scripting.ee_base_deploy*), 14

exit() (*ska_sdp_scripting.processing_block.ProcessingBlock* method), 9

F

FakeDeploy (class in *ska_sdp_scripting.fake_deploy*), 15

G

get_dependencies() (*ska_sdp_scripting.processing_block.ProcessingBlock* method), 10

get_id() (*ska_sdp_scripting.ee_base_deploy.EEDeploy* method), 14

get_parameters() (*ska_sdp_scripting.processing_block.ProcessingBlock* method), 10

get_scan_types() (*ska_sdp_scripting.processing_block.ProcessingBlock* method), 10

H

HelmDeploy (class in *ska_sdp_scripting.helm_deploy*), 14

I

is_job_finished() (*ska_sdp_scripting.phase.Phase* method), 13

is_finished() (*ska_sdp_scripting.ee_base_deploy.EEDeploy* method), 14

M

module *ska_sdp_scripting.queue_connector*, 16

monitor_deployments() (*ska_sdp_scripting.phase.Phase* method), 13

N

nested_parameters() (*ska_sdp_scripting.processing_block.ProcessingBlock* method), 10

P

Phase (class in *ska_sdp_scripting.phase*), 12

ProcessingBlock (class in *ska_sdp_scripting.processing_block*), 9

R

receive_addresses() (*ska_sdp_scripting.processing_block.ProcessingBlock* method), 10

remove() (*ska_sdp_scripting.ee_base_deploy.EEDeploy* method), 14

request_buffer() (*ska_sdp_scripting.processing_block.ProcessingBlock* static method), 10

S

`set_deployments_ready()`
 (*ska_sdp_scripting.fake_deploy.FakeDeploy*
 method), 15

`size` (*ska_sdp_scripting.buffer_request.BufferRequest*
 attribute), 11

`ska_sdp_scripting.queue_connector`
 module, 16

T

`tags` (*ska_sdp_scripting.buffer_request.BufferRequest*
 attribute), 11

`tear_down_queue_connector()` (in module
 ska_sdp_scripting.queue_connector), 16

U

`update_deploy_status()`
 (*ska_sdp_scripting.ee_base_deploy.EEDeploy*
 method), 14

`update_parameters()`
 (*ska_sdp_scripting.processing_block.ProcessingBlock*
 method), 11

`update_pb_state()` (*ska_sdp_scripting.phase.Phase*
 method), 13

W

`wait_for_queue_connector_state()` (in module
 ska_sdp_scripting.queue_connector), 16

`wait_loop()` (*ska_sdp_scripting.phase.Phase* *method*),
 13