developer.skatelescope.org Documentation

Release 0.2.6

Marco Bartolini

Oct 26, 2021

CONTENTS

1 Introduction 3

2 API 5

3 Receive Process and Port Configuration 11

4 Indices and tables 13

Index 15

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

CHAPTER

ONE

INTRODUCTION

1.1 Functionality

The required functionality of the workflow library is as follows.

1.1.1 Starting, monitoring and ending a workflow

- 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 workflow.
- Resource requests
  - Make requests for input and output buffer space. The workflow will calculate the resources it needs based on the parameters, then request them from the processing controller. This is currently a placeholder.
- Declare workflow phases
  - Workflows 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.
- 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 workflows, it also checks if the scheduling block instance has been cancelled.
  - Deploy execution engines to execute a script/function.
  - Monitor the execution engines and processing block state. Waits until the execution is finished, or the processing block is cancelled.
- At the end
  - Remove the execution engines to release the resources.
  - Update processing block state with information about the success or failure of the workflow.

1.1.2 Receive workflows

- 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.

1.2 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-workflow
```

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-workflow
```

1.3 Usage

Once the SDP workflow library have been installed, use:

```
import ska_sdp_workflow
```

1.4 Develop a new workflow

The steps to develop and test an SDP workflow can be found at Science Pipelines - Workflow Development.

CHAPTER

TWO

API

2.1 Processing block

class ska_sdp_workflow.workflow.**ProcessingBlock**(*pb_id=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.

Parameters

- **scan_types** – scan types from SBI
- **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 configured host and port

Return type dict

create_phase(*name*, *requests*)

Create a workflow 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()

Close connection to the configuration.

get_parameters(*schema=None*)

Get workflow 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()
Get scan types from the scheduling block instance.

This is only supported for real-time workflows

Returns scan types

Return type list

nested_parameters(*parameters*)
Convert flattened dictionary to nested dictionary.

Parameters **parameters** – parameters to be converted

Returns nested parameters

receive_addresses(*chart_name=None*, *service_name=None*, *namespace=None*, *configured_host_port=None*)
Generate receive addresses and update the processing block state.

Parameters

- **scan_types** – Scan types
- **chart_name** – Name of the statefulset
- **service_name** – Name of the headless service
- **namespace** – namespace where its going to be deployed
- **configured_host_port** – constructed host and port

request_buffer(*size*, *tags*)
Request a buffer reservation.

This returns a buffer reservation request that is used to create a workflow 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*, *parameters*)
Nested overwrite of default_parameter values with ones in parameters.

Parameters

- **default_parameters** –

  dict default parameter values

- **parameters** –

  dict workflow specific parameters

Returns processing block additional parameters

Return type dict

2.2 Buffer request

class ska_sdp_workflow.buffer_request.**BufferRequest**(*size*, *tags*)
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

2.3 Workflow phase

class ska_sdp_workflow.phase.**Phase**(*name*, *list_requests*, *config*, *pb_id*, *sbi_id*, *workflow_type*)
Workflow 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
- **sbi_id** (*str*) – scheduling block instance ID
- **workflow_type** (*str*) – workflow type

check_state(*txn*)
Check the state of the processing block.

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

Parameters **txn** (*ska_sdp_config.Transaction*) – SDP configuration transaction

ee_deploy_dask(*name*, *n_workers*, *func*, *f_args*)
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*, *values=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*, *func=None*, *f_args=None*)
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()
Remove execution engines deployments.

is_sbi_finished(*txn*)
Check if the SBI is finished or cancelled.

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

Return type bool

update_pb_state(*status=None*)
Update processing block state.

If the status is not provided, it is marked as finished.

Parameters **status** (*str, optional*) – status

wait_loop()
Wait loop to check the status of the processing block.

2.4 Execution engine deployment

class ska_sdp_workflow.ee_base_deploy.**EEDeploy**(*pb_id*, *config*)
Base class for execution engine deployment.

Parameters

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

get_id()
Get the deployment ID.

Returns deployment ID

Return type str

is_finished(*txn*)
Check if the deployment is finished.

Parameters **txn** (*ska_sdp_config.Transaction*) – configuration transaction

Return type bool

remove(*deploy_id*)
Remove the execution engine.

Parameters **deploy_id** (*str*) – deployment ID

update_deploy_status(*status*)
Update deployment status.

Parameters **status** (*str*) – status

2.5 Helm EE Deployment

class ska_sdp_workflow.helm_deploy.**HelmDeploy**(*pb_id*, *config*, *deploy_name*, *values=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

2.6 Dask EE deployment

class ska_sdp_workflow.dask_deploy.**DaskDeploy**(*pb_id*, *config*, *deploy_name*, *n_workers*, *func*, *f_args*)
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

2.7 Fake EE deployment

class ska_sdp_workflow.fake_deploy.**FakeDeploy**(*pb_id*, *config*, *deploy_name*, *func=None*, *f_args=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

CHAPTER

THREE

RECEIVE PROCESS AND PORT CONFIGURATION

3.1 Multiple Port Configuration

The workflow 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]]
```

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

B

BufferRequest (*class in ska_sdp_workflow.buffer_request*), 7

C

check_state() (*ska_sdp_workflow.phase.Phase method*), 7
configure_recv_processes_ports() (*ska_sdp_workflow.workflow.ProcessingBlock method*), 5
create_phase() (*ska_sdp_workflow.workflow.ProcessingBlock method*), 5

D

DaskDeploy (*class in ska_sdp_workflow.dask_deploy*), 9

E

ee_deploy_dask() (*ska_sdp_workflow.phase.Phase method*), 7
ee_deploy_helm() (*ska_sdp_workflow.phase.Phase method*), 7
ee_deploy_test() (*ska_sdp_workflow.phase.Phase method*), 8
ee_remove() (*ska_sdp_workflow.phase.Phase method*), 8
EEDeploy (*class in ska_sdp_workflow.ee_base_deploy*), 8
exit() (*ska_sdp_workflow.workflow.ProcessingBlock method*), 5

F

FakeDeploy (*class in ska_sdp_workflow.fake_deploy*), 10

G

get_id() (*ska_sdp_workflow.ee_base_deploy.EEDeploy method*), 8
get_parameters() (*ska_sdp_workflow.workflow.ProcessingBlock method*), 5
get_scan_types() (*ska_sdp_workflow.workflow.ProcessingBlock method*), 6

H

HelmDeploy (*class in ska_sdp_workflow.helm_deploy*), 9

I

is_finished() (*ska_sdp_workflow.ee_base_deploy.EEDeploy method*), 9
is_sbi_finished() (*ska_sdp_workflow.phase.Phase method*), 8

N

nested_parameters() (*ska_sdp_workflow.workflow.ProcessingBlock method*), 6

P

Phase (*class in ska_sdp_workflow.phase*), 7
ProcessingBlock (*class in ska_sdp_workflow.workflow*), 5

R

receive_addresses() (*ska_sdp_workflow.workflow.ProcessingBlock method*), 6
remove() (*ska_sdp_workflow.ee_base_deploy.EEDeploy method*), 9
request_buffer() (*ska_sdp_workflow.workflow.ProcessingBlock method*), 6

U

update_deploy_status() (*ska_sdp_workflow.ee_base_deploy.EEDeploy method*), 9
update_parameters() (*ska_sdp_workflow.workflow.ProcessingBlock method*), 6
update_pb_state() (*ska_sdp_workflow.phase.Phase method*), 8

W

wait_loop() (*ska_sdp_workflow.phase.Phase method*), 8