developer.skatelescope.org Documentation

Release 0.1.0

Marco Bartolini

Aug 23, 2021

CONTENTS

CHAPTER

ONE

WORKFLOW DEVELOPMENT

The steps to develop and test an SDP workflow are as follows:

1.1 1. Create the workflow

- Clone the ska-sdp-science-pipelines repository from GitLab and create a new branch for your work.
- Create a directory for your workflow in `src`:

```
$ mkdir src/<my_workflow>
$ cd src/<my_workflow>
```

- Write the workflow script (`<my_workflow>.py`). See the existing workflows for examples of how to do this. The examples *Real-time workflow* (*Test Real-Time Workflow*) and *Batch workflow* (*Test Batch Workflow*) are the best place to start. These are meant to give you a general idea of what structure batch and realtime workflows should have, and help develop your own.

  List of available Helm charts, which can be used for workflows, and their documentation can be found at: TBC

- Create a `requirements.txt` file with your workflow's Python requirements, e.g.

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

- Create a `Dockerfile` for building the workflow image, e.g.

```
FROM python:3.9-slim

COPY requirements.txt ./
RUN pip install -r requirements.txt

WORKDIR /app
COPY <my_workflow>.py ./
ENTRYPOINT ["python", "<my_workflow>.py"]
```

  Use the base-image of your choice, preferably the latest numbered slim version of it, e.g. python:3.9-slim.

- Create a file called `version.txt` containing the semantic version number of the workflow.
- Create a `Makefile` containing

```
NAME := ska-sdp-wflow-<my-workflow>
VERSION := $(shell cat version.txt)

include ../../make/Makefile
```

1.2 2. Test the workflow locally

- Build the workflow image. If you are using `minikube` to deploy the SDP, run:

  ```
  $ eval $(minikube -p minikube docker-env)
  $ make build
  ```

  else, just run the `make build` command. This will add the image to your `minikube` or local Docker daemon where it can be used for testing with a local deployment of the SDP.

- Deploy SDP locally and start a shell in the console pod.

- Add the new workflow to the configuration DB. This will tell the SDP where to find the Docker image to run the workflow:

  ```
  ska-sdp create workflow <kind>:<name>:<version> '{"image": "<docker-image:version>"}
  ↪'
  ```

  where the values are:

  - `<kind>`: `batch` or `realtime`, depending on the kind of workflow
  - `<name>`: name of your workflow
  - `<version>`: version of your workflow
  - `<docker-image:version>`: Docker image you just built from your workflow, including its version tag.

  If you have multiple workflows to add, you can import the definitions with:

  ```
  ska-sdp import some-workflows.json
  ```

  An example JSON file for importing workflows can be found at: Example Workflow JSON

- To run the workflow, create a processing block, either via the Tango interface, or by creating it directly in the configuration DB with ska-sdp create pb.

1.3 3. Finish development and deploy the workflow

- Once you are happy with the workflow, add it to the GitLab CI file (`.gitlab-ci.yml`) in the root of the repository. You need to add a build and publish job for it:

```
build-<my_workflow>:
  extends: .docker_build_workflow
  before_script:
    - cd src/<my_workflow>>
  only:
    changes:
```

(continues on next page)

(continued from previous page)

```
      - src/<my_workflow>/*

publish-<my_workflow>:
  extends: .publish
  before_script:
    - cd src/<my_workflow>
  only:
    refs:
      - master
    changes:
      - src/<my_workflow>/*
```

This will enable the Docker image to be built and pushed to the SKA artefact repository when it is merged into the master branch.

- Add the workflow to the workflow definition file `workflows.json` in the root of the repository. By default the SDP uses this file to populate the workflow definitions in the configuration DB when it starts up.
- Create a `README.md` and add the description and instructions to run your workflow. Include it in the documentation:
  - create a new file in `docs/src/<my_workflow>.rst`
  - add the following to it:

    ```
    .. mdinclude:: ../../src/<my_workflow>/README.md
    ```

  - update `docs/src/index.rst`
- Commit the changes to your branch and push to GitLab.

1.4 Workflow Script Generator

To speed things up for the workflow developer, a python script has been developed to automatically generate the source files for a workflow.

The script will generate all the required source files (Python script, requirements, version number, Dockerfile, Makefile, README) for both real-time and batch workflow.

To run the script:

```
cd workflow_template
python create_workflow.py <kind> <name>
```

For example:

```
python create_workflow.py realtime test_realtime_workflow
```

1.5 Usage

```
Create generic source files for batch/real-time workflows.

Usage:
    create_workflow.py <kind> <name>
    create_workflow.py (-h|--help)

Arguments:
    <kind>    Type of workflow (realtime or batch)
    <name>    Name of the workflow to be created
```

CHAPTER

TWO

EXAMPLES

Simple examples of realtime and batch workflows to help developing new ones.

2.1 Real-time workflow

This is a simple example of a real-time workflow (`test_realtime`). It requires the *logging*, *ska_ser_logging* and *ska_sdp_workflow* Python modules.

```
import logging
import ska_ser_logging
import ska_sdp_workflow
```

We initialise the logging

```
ska_ser_logging.configure_logging()
LOG = logging.getLogger('test_realtime')
LOG.setLevel(logging.DEBUG)
```

The first step is to claim the processing block and get its parameters.

```
pb = workflow.ProcessingBlock()
parameters = pb.get_parameters()
```

We then need to request the input and output buffer reservations. This is just a placeholder currently to give an example of how resources will be requested.

```
in_buffer_res = pb.request_buffer(100e6, tags=['sdm'])
out_buffer_res = pb.request_buffer(parameters['length'] * 6e15 / 3600, tags=[
↪'visibilities'])
```

We declare the phases of the workflow and define which resource requests are needed for each phase. In the current implementation, we can only declare one phase, which in this example we call the ‘work’ phase:

```
work_phase = pb.create_phase('Work', [in_buffer_res, out_buffer_res])
```

We start the work phase using a `with` block. On entry, it waits until the resources are available and in the meantime it monitors the processing block state. Once the resources are available, it proceeds into the body of the `with` block and deploys a Helm chart using the `ee_deploy_helm` method. This happens in a separate thread, so the method returns immediately. It then enters the loop at the end, which monitors the scheduling block instance to check if it has been cancelled.

```
with work_phase:
    work_phase.ee_deploy_helm('cbf-sdp-emulator')

    for txn in work_phase.wait_loop():
        if work_phase.is_sbi_finished(txn):
            break
        txn.loop(wait=True)
```

On exiting the `with` block it waits until the scheduling block instance is finished. When the scheduling block instance state is updated, it then removes the execution engine and updates the processing block state.

2.2 Batch workflow

This is a simple example of a batch workflow (`test_batch`). It requires the *time*, *logging*, *ska_ser_logging* and *ska_sdp_workflow* Python modules:

```
import time
import logging
import ska_ser_logging
import ska_sdp_workflow
```

We initialise the logging:

```
ska_ser_logging.configure_logging()
LOG = logging.getLogger('test_batch')
LOG.setLevel(logging.DEBUG)
```

The first step is to claim the processing block and get its parameters.

```
pb = ska_sdp_workflow.ProcessingBlock()
parameters = pb.get_parameters()
```

We then need to request the input and output buffer reservations. This is just a placeholder currently to give an example of how resources will be requested.

```
in_buffer_res = pb.request_buffer(100e6, tags=['sdm'])
out_buffer_res = pb.request_buffer(parameters['length'] * 6e15 / 3600, tags=[
↪'visibilities'])
```

We declare the phases of the workflow and define which resource requests are needed for each phase. In the current implementation, we can only declare one phase, which in this example we call the ‘work’ phase:

```
work_phase = pb.create_phase('work', [in_buffer_res, out_buffer_res])
```

Next, we define the function to be executed by the execution engine. In a real workflow this would most likely be imported from a library of standard workflow functions.

```
def some_processing(duration: float):
    """
    Do some 'processing' for the required duration.
    """
    LOG.info('Starting processing for %f s', duration)
```

(continues on next page)

(continued from previous page)

```
    time.sleep(duration)
    LOG.info('Finished processing')
```

We start the work phase using a `with` block. On entry, it waits until the resources are available and in the meantime it monitors the processing block state to check if it has been cancelled. Once the resources are available, it proceeds into the body of the `with` block and deploys a fake execution engine using the `ee_deploy_test` method to execute our function. The example we defined sleeps for the specified duration. This happens in a separate thread, so the method returns immediately. It then enters the loop at the end which waits until the execution is finished, and also monitors the processing block state and whether the deployment has been cancelled or not.

```
with work_phase:

    deploy = work_phase.ee_deploy_test(
        'test', some_processing, (parameters['duration'],)
    )

    for txn in work_phase.wait_loop():
        if deploy.is_finished(txn):
            break
```

On exiting the `with` block, it removes the execution engine deployment and updates the processing block state with the status of the workflow.

CHAPTER

THREE

WORKFLOWS

Science Pipelines for SDP.

3.1 Visibility Receive Workflow

This is the integration of the CBF-SDP interface emulation and receive workflow. This workflow deploys and updates the receive addresses attribute with DNS-based IP address.

3.1.1 Deploying the Receive Workflow in the SDP Prototype

To set up SDP on your local machine using Minikube, follow the instructions at Running SDP stand-alone.

Configuring the Workflow

Although this repository contains example workflows - the helm-charts specific to deployment of the SDP prototype are stored in another repository (https://gitlab.com/ska-telescope/sdp/ska-sdp-helmdeploy-charts). For the purposes of continuity we will document the use of the chart here. The documentation may be replicated in other repositories.

The important consideration for the current version of the emulator and receive workflow is that the interface Telescope Model is via the measurement set. As the charts need to be agnostic about where and how they are deployed it was necessary to provide a schema whereby the data-model could be accessed by the deployment. What we do here is we provide a mechanism by which the model can be pulled by providing a URL to a compressed tarfile of the model measurement set, and the name of that measurement set once unzipped. This should be the same as the measurement set that will be transmitted by the emulator to allow the UVW and timestamps to match.

There are few setups we need to do before running the workflow. First need to download the `sim-vis.ms.tar.gz` from the CBF-SDP Emulator Repository.

Extract the file:

```
> tar -xf sim-vis.ms.tar
```

Need to create a persistent volume, to do that create a file called `pvc.yaml` and add the following:

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: pv-local
  labels:
    type: local
```

(continues on next page)

(continued from previous page)

```
spec:
  storageClassName: local
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "<path to sim-vis.ms"
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: local-pvc
spec:
  storageClassName: local
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
  selector:
    matchLabels:
      type: local
```

Make sure to update the hostPath.

Create persistent volume by executing the following command:

```
> kubectl create -f pvc.yaml -n sdp
```

The configuration of the receive workflow is managed via adding to the configuration of the processing block. The processing block (PB) can only be created using the iTango interface. This is a realtime workflow, therefore it is linked to a Scheduling Block Instance (SBI). Currently, there is no option to create a PB and link it to SBI using the `ska-sdp` utility.

To run the workflow using iTango interface, follow the instructions at Running SDP stand-alone.

Use the following configuration string. This contains one real-time processing block, which uses the `vis_receive` workflow, and one batch processing block the `test_batch` as a placeholder workflow:

```
config_sbi = '''
{
  "interface": "https://schema.skao.int/ska-sdp-assignres/0.2",
  "id": "sbi-mvp01-20200619-00000",
  "max_length": 21600.0,
  "scan_types": [
    {
      "id": "science_A",
      "coordinate_system": "ICRS", "ra": "02:42:40.771", "dec": "-00:00:47.84",
      "channels": [
        { "count": 5, "start": 0, "stride": 2, "freq_min": 0.35e9, "freq_max": 0.368e9,
↪"link_map": [[0,0], [200,1], [744,2], [944,3]] }
      ]
    },
```

(continues on next page)

(continued from previous page)

```
    {
      "id": "calibration_B",
      "coordinate_system": "ICRS", "ra": "12:29:06.699", "dec": "02:03:08.598",
      "channels": [
        { "count": 5, "start": 0, "stride": 2, "freq_min": 0.35e9, "freq_max": 0.368e9,
↪"link_map": [[0,0], [200,1], [744,2], [944,3]] }
      ]
    }
  ],
  "processing_blocks": [
    {
      "id": "pb-mvp01-20200619-00000",
      "workflow": {"type": "realtime", "id": "vis_receive", "version": "0.3.3"},
      "parameters": {}
    },
    {
      "id": "pb-mvp01-20200619-00001",
      "workflow": {"type": "batch", "id": "test_batch", "version": "0.2.4"},
      "parameters": {},
      "dependencies": [
        {"pb_id": "pb-mvp01-20200619-00000", "type": ["visibilities"]}
      ]
    }
  ]
}'''
```

Note that each workflow may come with multiple versions. Always use the latest number, unless you know a specific version that suits your needs, or you follow the example above. (The Changelog at the end of this page may help to decide.)

This will start up a default deployment. It will launch a receiver pod. All the options supported by the receiver are supported by the chart deployment. The defaults set by the workflow currently are:

```
"image": "nexus.engageska-portugal.pt/ska-docker/cbf_sdp_emulator",
"version": "latest",
"recv_emu": "emu-recv",
"model.name": "sim-vis.ms",
"transmission.channels_per_stream": 4,
"transmission.rate": "147500",
"payload.method": "icd",
"reader.num_timestamps": 0,
"reader.start_chan": 0,
"reader.num_chan": 0,
"reader.num_repeats": 1,
"results.push": False,
"pvc.name": "local-pvc",
"pvc.path": "/mnt/data",
"reception.outputfilename": "output.ms",
"reception.receiver_port_start": "41000",
"reception.num_ports": 1,
```

To add additional parameters or to override default parameters, can be done by adding to the `parameters` key of the PB via the configuration string.

Once the pod is deployed with the desired configuration, the receive will be running as a server inside a pod and waiting for packets from the emulator (or even the actual CBF).

Following functionality are not available with the generic `receive` chart. Would need to update workflow to use the `cbf-sdp-emulator` chart. This is can be done by updating the `deploy_name`

Retrieving Data from Kubernetes Deployments

If the receive workflow is configured to generate a measurement set, then it needs to be exported from the Kubernetes environment. The mechanism we have provided for this is mediated by the `rclone` package `https://rclone.org`. In order for this to work in a secure manner we have provided a mechanism by which a container can pull an rclone configuration file - containing the credentials and configured end points. This configuration is then used by a container to push the results out. There are only two configuration options required:

```
- ``rclone.configurl``. This is a URL of an ``rclone.conf``. Please see the rclone␣
↪documentation for instructions regarding the generation of this.
- ``rclone.command``. This is the destination you want for the measurement set in the␣
↪format expected by rclone - namely theremote type, as defined in your configuration␣
↪file, followed by the path for that remote.
```

For example this is a workflow configuration utilising this capability:

```
"parameters": {"transmit.model": False,
               "results.push": True,
               "rclone.configurl": "https://www.dropbox.com/s/yqmzfs8ovtnonbe/rclone.
↪conf?dl=1",
               "rclone.command": "gcs:/yan-486-bucket/demo.ms"}
```

After the receive workflow completes, the data will be synchronised with the end-point.

Deploying the Receive Workflow Behind a Proxy (PSI deployments)

One of the more complex issues to deal with when deploying to a Kubernetes environment is networking. This is made more difficult if the Kubernetes environment itself is behind a firewall. The SDP prototype deployment can be thought of as charts that instantiate containers that themselves instantiate containers. Proxies are usually exposed through environment variables which requires the environment to be propagated from chart to chart.

The PSI (Prototype System Integration) in an integration environment which is managed by CSIRO and behind a web-proxy. When SDP is deployed, all the elements of the prototype need to be informed of the proxy.

Configuring Workflow to Use The Proxy

Firstly, SDP needs to be deployed with a proxy setting exposed. Upgrading the SDP helm deployment with the following command will expose the CSIRO proxy to the helm charts of the SDP:

```
helm upgrade test ska/sdp --set proxy.server=delphinus.atnf.csiro.au:8888 --set proxy.
↪noproxy='{}'
```

(Above we assumed you deployed SDP using the `ska/sdp` chart with the name `test`.) This will ensure the prototype itself is launched with the correct proxy settings.

But as you would expect this does not necessarily pass the proxy settings on to the workflows.

In the case of the receive workflow, with the proxy information:

```
"parameters": {"transmit.model": False,
               "results.push": True,
               "rclone.configurl": "https://www.dropbox.com/s/yqmzfs8ovtnonbe/rclone.
↪conf?dl=1",
               "rclone.command": "gcs:/yan-483-bucket/psi-demo002.ms",
               "reception.ring_heaps": 133}
```

Passing these parameters would launch the receive workflow on the PSI, behind a proxy, configured to push the results to a Google Cloud Services bucket.

Changelog

0.3.4

- Use dependencies from the central artefact repository and publish the workflow image there.

0.3.3

- Ported to use the latest version of workflow library (0.2.4). Capable to deploy multiple receive processes. Ports published in the receive addresses match with the actual ports of the receive process(es)

0.3.2

- use python:3.9-slim as the base docker image

3.2 PSS Receive Workflow

This is a cheetah-based User Datagram Protocol (UDP) receiver for ingesting PSS (Pulsar Search Sub-system) single pulse candidate data in the SDP (Science Data Processor).

3.2.1 Description

Cheetah is a set of real-time pulsar and fast-transient searching pipelines designed to process data from the CBF (Correlator and Beamformer). Its end products are candidate pulsar and transient signals which are exported to the SDP for further analysis. The pipelines are started using a command line executable which can be configured using command line flags or by passing a file containing pipeline configuration information. A full description of the pipeline can be found in the PSS Detailed Design Document.

For the purposes of this demonstration we will use the cheetah_pipeline executable (PSS) to receive UDP time-frequency data from a CBF emulator pipeline, from which it will produce single-pulse candidate data which is exported over a UDP stream using SPEAD2. We will use a separate cheetah based pipeline to "receive" this data on the SDP side.

The executables are served from the pss-centos-docker repository.

3.2.2 Running send and receive standalone

To demonstrate their functionality, it is possible to deploy the CBF emulator, the PSS pipeline and the SDP receive pipeline in same docker container by downloading and running the pss docker.

Pull the PSS docker image by running.

```
$ docker pull nexus.engageska-portugal.pt/ska-telescope/pss-docker-centos-dev:1.0.0
```

It's important to ensure that the maximum receive buffer size for all connection types is appropriately tuned on the host, so it may be necessary to run the following command before we start the docker container. If this is not set correctly, SPEAD2 will print a warning to the console when we start the receiver and we risk packets being lost. (Note, the below command was tested on Linux.)

```
sudo sysctl net.core.rmem_max=268435456 && sudo sysctl net.core.wmem_max=268435456 &&␣
↪sudo sysctl net.core.netdev_max_backlog=65536 && sudo sysctl net.core.wmem_
↪default=268435456 && sudo sysctl net.core.rmem_default=268435456
```

Once the docker image is pulled, we can start it. The entrypoint starts the receiver listening so we don't need to override it.

```
$ docker run -it nexus.engageska-portugal.pt/ska-telescope/pss-docker-centos-dev:1.0.0
```

In a separate terminal we can connect to this docker in order to trigger the cheetah pipeline to start listening for a UDP stream from the CBF emulator.

```
$ docker exec -it <container id> bash
```

By default, the cheetah_pipeline is configured (in our config files) to send candidate data to pss-receive and our CBF emulator will send data to cbf-receive, but this time we want both to send to localhost, so we'll need to adjust this parameter in our config files. This file is located at /opt/pss-pipeline/configurations/mvp_emulator_config.xml. Find the lines where we configure the IP address spead2 will send to and replace "pss-receive" with "localhost" and save the file". Do the same with cbf-emulator.xml (in the same directory) and replace "cbf-receive" with "localhost".

```
<spccl_spead>
 <candidate_window>
  <ms_before>10</ms_before>
  <ms_after>10</ms_after>
 </candidate_window>
 <rate_limit>1e7</rate_limit>
 <ip>
  <port>9021</port>
  <ip_address>pss-receive</ip_address>
 </ip>
 <id>spead_stream</id>
</spccl_spead>
```

Navigate to /opt/build/thirdparty/cheetah/src/cheetah-build/cheetah/pipeline. In this directory we'll find the cheetah_pipeline executable. Run this with -h to see the options. We'll trigger the cheetah pipeline with the following command.

```
$ ./cheetah_pipeline -s udp_low -p SinglePulse --log-level debug --config /opt/pss-
↪pipeline/configurations/mvp_emulator_config.xml
```

Where:

- -s denotes the source of the input data stream (in this case a udp stream from the CBF emulator
- -p is the specific pipeline that cheetah should run (in this case the single pulse search pipeline)
- –config is the path to the configuration file

We see some output from the pss pipeline to show that it is in a listening state.

```
[log][tid=140131450292096][/opt/pss-pipeline/thirdparty/cheetah/cheetah/../cheetah/
↪pipeline/detail/BeamLauncher.cpp:148][1613571585]Creating Beams....
[log][tid=140131450292096][/opt/pss-pipeline/thirdparty/cheetah/cheetah/rcpt_low/src/
↪UdpStream.cpp:37][1613571585]listening for UDP Low stream from 0.0.0.0:9029
[debug][tid=140131450292096][/opt/pss-pipeline/thirdparty/cheetah/cheetah/../cheetah/
↪exporters/detail/DataExport.cpp:61][1613571585]Creating sink of type spccl_files␣
↪(id=spccl_files)
[debug][tid=140131450292096][/opt/pss-pipeline/thirdparty/cheetah/cheetah/../cheetah/
↪exporters/detail/DataExport.cpp:61][1613571585]Creating sink of type spccl_sigproc_
↪files (id=candidate_files)
[debug][tid=140131450292096][/opt/pss-pipeline/thirdparty/cheetah/cheetah/../cheetah/
↪exporters/detail/DataExport.cpp:61][1613571585]Creating sink of type spccl_spead␣
↪(id=spead_stream)
[log][tid=140131450292096][/opt/pss-pipeline/thirdparty/cheetah/cheetah/exporters/src/
↪SpCclSpeadStreamer.cpp:46][1613571585]Spead UDP output stream on 127.0.0.1:9021␣
↪(limited to 10000000.000000 bytes/sec)
[log][tid=140131450292096][/opt/build/thirdparty/panda/install/include/panda/detail/
↪packet_stream/PacketStreamImpl.cpp:125][1613571585]start packet stream listening on:0.
↪0.0.0:9029
[log][tid=140131450292096][/opt/pss-pipeline/thirdparty/cheetah/cheetah/../cheetah/
↪pipeline/detail/BeamLauncher.cpp:171][1613571585]Finished creating pipelines
[log][tid=140131450292096][/opt/pss-pipeline/thirdparty/cheetah/cheetah/../cheetah/
↪pipeline/detail/BeamLauncher.cpp:223][1613571585]Starting Beam: 1
```

We can then trigger the CBF emulator to send to data into pss where the single pulse search pipeline will process it. Connect a third terminal to this docker container and navigate to /opt/build/thirdparty/cheetah/src/cheetah-build/cheetah/emulator. In this directory we'll find the cheetah_emulator executable. Run this with -h to see the options. We'll trigger the emulator with the following command.

```
$ ./cheetah_emulator --config /opt/pss-pipeline/configurations/cbf_emulator_config.xml --
↪log-level debug
```

The emulator will produce some log messages to show it's working and streaming Gaussian noise over UDP.

```
[log][tid=140592331782016][/opt/build/thirdparty/panda/install/include/panda/detail/
↪SocketConnectionImpl.cpp:203][1613572250][this=0x1792948] setting remote endpoint:127.
↪0.0.1:9029
[log][tid=140592331782016][/opt/pss-pipeline/thirdparty/cheetah/cheetah/emulator/src/
↪emulator_main.cpp:63][1613572250]emulator using generator: 'gaussian_noise'
```

and after a short time, the cheetah_pipeline will show a bunch of log message to show that it is processing the data that it is receiving from CBF.

```
[debug][tid=140130935695104][/opt/build/thirdparty/panda/install/include/panda/detail/
↪packet_stream/ChunkerContext.cpp:453][1613572937]clearing chunk 0x7f72c0099610
[debug][tid=140130935695104][/opt/build/thirdparty/panda/install/include/panda/detail/
↪DataManager.cpp:221][1613572937]pushing data 0x7f72c0099610
```

Eventually we'll see that pss-receive is starting to receive candidate data when log message like the following appear

```
[debug][tid=139774761645952][/opt/pss-pipeline/thirdparty/cheetah/cheetah/exporters/src/
↪SpeadLoggingAdapter.cpp:50][1613572975]spead: packet with 1432 bytes of payload at␣
↪offset 52279064 added to heap 9
```

Whenever we like, we can simple CTRL+C on the CBF emulator and wait for the final packets to arrive at pss-receive.

What just happened? We triggered the pss pipeline to listen for test data from a udp stream. This data was passed through a single pulse search emulator module and the resulting single pulse candidate data was exported to the pss-receive application.

3.2.3 Deploying pss_receive as an SDP workflow in Minikube

Generalised instruction for deploying the SDP can be found at Running SDP stand-alone.

Start minikube,

```
$ minikube config set memory 16384
$ minikube start --cpus=16 --driver virtualbox
```

and set the minikube buffer size (command tested on Linux)

```
$ minikube ssh
> sudo sysctl net.core.rmem_max=268435456 && sudo sysctl net.core.wmem_max=268435456 &&␣
↪sudo sysctl net.core.netdev_max_backlog=65536 && sudo sysctl net.core.wmem_
↪default=268435456 && sudo sysctl net.core.rmem_default=268435456
> CTRL+D
```

Create the SDP namespace, add the SDP chart repository to helm and launch the SDP

```
$ kubectl create namespace sdp
$ helm repo add ska https://artefact.skao.int/repository/helm-internal
$ helm repo update
$ helm install test ska/ska-sdp --set helmdeploy.createClusterRole=true
```

You can watch the progress of installing the SDP by running

```
$ watch -n 0.5 kubectl get all
```

and once it's up and running it should look something like..

```
Every 0.5s: kubectl get all

NAME                                  READY   STATUS      RESTARTS   AGE
pod/databaseds-tango-base-test-0      1/1     Running     3          23h
pod/sdp-console-0                     1/1     Running     0          23h
pod/sdp-etcd-0                        1/1     Running     0          23h
pod/sdp-helmdeploy-0                  1/1     Running     0          23h
pod/sdp-lmc-configurator-dtmrp        0/1     Completed   0          23h
pod/sdp-lmc-master-0                  1/1     Running     0          23h
pod/sdp-lmc-subarray-1-0              1/1     Running     0          23h
pod/sdp-lmc-subarray-2-0              1/1     Running     0          23h
pod/sdp-lmc-subarray-3-0              1/1     Running     0          23h
```

(continues on next page)

(continued from previous page)

```
pod/sdp-proccontrol-0                 1/1     Running     0          23h
pod/tango-base-tangodb-0              1/1     Running     0          23h
pod/tangotest-tango-base-test-test-0  1/1     Running     0          23h

NAME                                     TYPE        CLUSTER-IP       EXTERNAL-IP  ␣
↪PORT(S)              AGE
service/databaseds-tango-base-test   NodePort    10.106.233.129   <none>       ␣
↪10000:30412/TCP     23h
service/kubernetes                       ClusterIP   10.96.0.1        <none>        443/
↪TCP             43h
service/sdp-console                      ClusterIP   None             <none>
↪<none>              23h
.
.
.
. and so on
```

Now we can connect to the sdp-console pod to give us access to the `ska-sdp` tool which with we can start a workflow.

```
$ kubectl exec -it sdp-console-0 -- bash
```

Now let's start the pss_receive workflow which will deploy the cheetah receiver container

```
$ ska-sdp create pb realtime:pss_receive:0.2.1
```

Note that each workflow may come with multiple versions. Always use the latest number, unless you know a specific version that suits your needs. (The Changelog at the end of this page may help to decide.)

We can watch the deployment of the workflow in the sdp namespace

```
$ watch -n 0.5 kubectl get all - n sdp
  NAME                                         READY   STATUS    RESTARTS   AGE
  pod/proc-pb-sdpcli-20210217-00003-workflow-njh8s   1/1     Running   0          21s
  pod/pss-receive-9rsbf                        1/1     Running   0          14s

  NAME                  TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
  service/pss-receive   ClusterIP   10.111.94.131   <none>        9021/UDP   14s

  NAME                                         COMPLETIONS   DURATION   AGE
  job.batch/proc-pb-sdpcli-20210217-00003-workflow   0/1           21s        22s
  job.batch/pss-receive                        0/1           14s        14s
```

Looking at the logs from the pss-receive pod we can see the receiver waiting for data…

```
$ kubectl logs pss-receive-9rsbf -n sdp
[debug][tid=139702737897344][/opt/pss-pipeline/thirdparty/cheetah/cheetah/../cheetah/
↪exporters/detail/DataExport.cpp:61][1613583356]Creating sink of type sp_candidate_data␣
↪(id=candidate_files)
```

and we have a service waiting to route data sent to the hostname pss-receive to this pod. Now to send this some data we can use the pss-pipeline kubernetes manifest, deploy-sender.yaml. This will deploy cheetah as a new pod. It will wait for data to arrive from the CBF emulator (which we'll deploy shortly) and when it arrives will run the single pulse emulator pipeline and export the candidate to the pss-receive application.

```
$ kubectl apply deploy-sender.yaml -n sdp
```

Our sender pod "pss-pipeline" will appear in the sdp namespace

```
NAME                                                READY   STATUS              RESTARTS ␣
↪ AGE
pod/proc-pb-sdpcli-20210217-00003-workflow-njh8s   1/1     Running             0        ␣
↪ 3m9s
pod/pss-pipeline-wjdm6                              0/1     ContainerCreating   0        ␣
↪ 2s
pod/pss-receive-9rsbf                               1/1     Running             0        ␣
↪ 3m2s

NAME                    TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
service/cbf-receive     ClusterIP   10.102.80.249   <none>        9029/UDP   2s
service/pss-receive     ClusterIP   10.111.94.131   <none>        9021/UDP   3m2s

NAME                                                COMPLETIONS   DURATION   AGE
job.batch/proc-pb-sdpcli-20210217-00003-workflow   0/1           3m10s      3m11s
job.batch/pss-pipeline                              0/1           3s         3s
job.batch/pss-receive                               0/1           3m3s       3m3s
```

We can see that the pss-pipeline is waiting for data from the CBF by running..

```
$ kubectl logs pss-pipeline-wjdm6 -n sdp
  [log][tid=140007832209280][/opt/pss-pipeline/thirdparty/cheetah/cheetah/../cheetah/
↪pipeline/detail/BeamLauncher.cpp:148][1613583537]Creating Beams....
  [log][tid=140007832209280][/opt/pss-pipeline/thirdparty/cheetah/cheetah/rcpt_low/src/
↪UdpStream.cpp:37][1613583537]listening for UDP Low stream from 0.0.0.0:9029
  [debug][tid=140007832209280][/opt/pss-pipeline/thirdparty/cheetah/cheetah/../cheetah/
↪exporters/detail/DataExport.cpp:61][1613583537]Creating sink of type spccl_files␣
↪(id=spccl_files)
  [debug][tid=140007832209280][/opt/pss-pipeline/thirdparty/cheetah/cheetah/../cheetah/
↪exporters/detail/DataExport.cpp:61][1613583537]Creating sink of type spccl_sigproc_
↪files (id=candidate_files)
  [debug][tid=140007832209280][/opt/pss-pipeline/thirdparty/cheetah/cheetah/../cheetah/
↪exporters/detail/DataExport.cpp:61][1613583537]Creating sink of type spccl_spead␣
↪(id=spead_stream)
  [log][tid=140007832209280][/opt/pss-pipeline/thirdparty/cheetah/cheetah/exporters/src/
↪SpCclSpeadStreamer.cpp:46][1613583537]Spead UDP output stream on 10.111.94.131:9021␣
↪(limited to 10000000.000000 bytes/sec)
  [log][tid=140007832209280][/opt/build/thirdparty/panda/install/include/panda/detail/
↪packet_stream/PacketStreamImpl.cpp:125][1613583537]start packet stream listening on:0.
↪0.0.0:9029
  [log][tid=140007832209280][/opt/pss-pipeline/thirdparty/cheetah/cheetah/../cheetah/
↪pipeline/detail/BeamLauncher.cpp:171][1613583537]Finished creating pipelines
  [log][tid=140007832209280][/opt/pss-pipeline/thirdparty/cheetah/cheetah/../cheetah/
↪pipeline/detail/BeamLauncher.cpp:223][1613583537]Starting Beam: 1
```

Now we can deploy the CBF emulator to stream UDP time frequency data using the pss_receive:0.2.1 deploy-cbf-emulator.yaml:

```
$ kubectl apply -f deploy-cbf-emulator.yaml -n sdp
```

This will deploy a cbf-emulator pod. Now let's watch the logs of the pss-receive application and eventually we'll see packets arriving..

```
$ kubectl logs -f pss-receive-9rsbf -n sdp
  .
  .
  [debug][tid=139702737897344][/opt/pss-pipeline/thirdparty/cheetah/cheetah/exporters/
↪src/SpeadLoggingAdapter.cpp:50][1613584774]spead: packet with 1432 bytes of payload at␣
↪offset 211172184 added to heap 2
  [debug][tid=139702737897344][/opt/pss-pipeline/thirdparty/cheetah/cheetah/exporters/
↪src/SpeadLoggingAdapter.cpp:50][1613584774]spead: packet with 1432 bytes of payload at␣
↪offset 211173616 added to heap 2
  [debug][tid=139702737897344][/opt/pss-pipeline/thirdparty/cheetah/cheetah/exporters/
↪src/SpeadLoggingAdapter.cpp:50][1613584774]spead: packet with 1432 bytes of payload at␣
↪offset 211175048 added to heap 2
 .
 .
 .
```

Cleaning up.

We can simply turn off the CBF and PSS will the following..

```
kubectl delete job cbf-emulator -n sdp
kubectl delete job pss-pipeline -n sdp
kubectl delete service cbf-receive -n sdp
```

Removing the pss_receive workflows using the sdp-console. First get a list of entries in the configuration database.

```
$ ska-sdp list -a
Keys with / prefix:
 /deploy/proc-pb-sdpcli-20201126-00000-pss-receive
 /deploy/proc-pb-sdpcli-20201126-00000-workflow
 /master
 /pb/pb-sdpcli-20201126-00000
 /pb/pb-sdpcli-20201126-00000/owner
 /pb/pb-sdpcli-20201126-00000/state
 /subarray/01
 /subarray/02
 /subarray/03
```

Now remove the following entries, and we'll see items disappear from the sdp namespace

```
$ ska-sdp delete deployment proc-pb-sdpcli-20201126-00000-pss-receive
$ ska-sdp delete deployment proc-pb-sdpcli-20201126-00000-workflow
```

Then we can disable the sdp

```
$ helm uninstall test
```

3.2.4 Changelog

0.2.3

- Use dependencies from the central artefact repository and publish the workflow image there.

0.2.2

- use python:3.9-slim as the base docker image

3.3 Batch Imaging Workflow

The `batch_imaging` workflow is a proof-of-concept of integrate a scientific workflow with the SDP prototype. It simulates visibilities and images them using RASCIL with Dask as an execution engine.

The workflow simulates SKA1-Low visibility data in a range of hour angles from -30 to 30 degrees and adds phase errors. The visibilities are then calibrated and imaged using the ICAL pipeline.

The workflow creates buffer reservations for storing the visibilities and images.

3.3.1 Parameters

The workflow parameters are:

- `n_workers`: number of Dask workers to deploy
- `freq_min`: minimum frequency (in hertz)
- `freq_max`: maximum frequency (in hertz)
- `nfreqwin`: number of frequency windows
- `ntimes`: number of time samples
- `rmax`: maximum distance of stations to include from array centre (in metres)
- `ra`: right ascension of the phase centre (in degrees)
- `dec`: declination of the phase centre (in degrees)
- `buffer_vis`: name of the buffer reservation to store visibilities
- `buffer_img`: name of the buffer reservation to store images

For example:

```
{
  "n_workers": 4,
  "freq_min": 0.9e8,
  "freq_max": 1.1e8,
  "nfreqwin": 8,
  "ntimes": 5,
  "rmax": 750.0,
  "ra": 0.0,
  "dec": -30.0,
  "buffer_vis": "buff-pb-mvp01-20200523-00001-vis",
```

(continues on next page)

(continued from previous page)

```
  "buffer_img": "buff-pb-mvp01-20200523-00001-img"
}
```

3.3.2 Running the workflow using iTango

If using Minikube, make sure to increase the memory size (minimum 16 GB):

```
minikube start --memory=16g
```

Once the SDP is running, start an iTango shell.

First, obtain a handle to a subarray device and turn it on:

```
d = DeviceProxy('mid_sdp/elt/subarray_1')
d.On()
```

If you are not sure what devices are available, list them with `lsdev`.

Create a configuration string for the scheduling block instance. This contains one real-time processing block, which uses the `test_realtime` workflow as a placeholder, and one batch processing block containing the `batch_imaging` workflow, which uses the example parameters from above:

```
config_sbi = '''
{
  "id": "sbi-mvp01-20200523-00000",
  "max_length": 21600.0,
  "scan_types": [
    {
      "id": "science",
      "channels": [
        {"count": 8, "start": 0, "stride": 1, "freq_min": 0.9e8, "freq_max": 1.1e8,
↪"link_map": [[0,0]]}
      ]
    }
  ],
  "processing_blocks": [
    {
      "id": "pb-mvp01-20200523-00000",
      "workflow": {"type": "realtime", "id": "test_realtime", "version": "0.2.2"},
      "parameters": {}
    },
    {
      "id": "pb-mvp01-20200523-00001",
      "workflow": {"type": "batch", "id": "batch_imaging", "version": "0.1.1"},
      "parameters": {
        "n_workers": 4,
        "freq_min": 0.9e8,
        "freq_max": 1.1e8,
        "nfreqwin": 8,
        "ntimes": 5,
        "rmax": 750.0,
        "ra": 0.0,
```

(continues on next page)

(continued from previous page)

```
        "dec": -30.0,
        "buffer_vis": "buff-pb-mvp01-20200523-00001-vis",
        "buffer_img": "buff-pb-mvp01-20200523-00001-img"
      },
      "dependencies": [
        {"pb_id": "pb-mvp01-20200523-00000", "type": ["none"]}
      ]
    }
  ]
}
'''
```

Note that each workflow may come with multiple versions. Always use the latest number, unless you know a specific version that suits your needs. (The Changelog at the end of this page may help to decide.)

The scheduling block instance is created by the `AssignResources` command:

```
d.AssignResources(config_sbi)
```

In order for the batch processing to start, you need to end the real-time processing with the `ReleaseResources` command:

```
d.ReleaseResources()
```

You can watch the pods and persistent volume claims (for the buffer reservations) being deployed with the following command or using k9s:

```
kubectl -w get pod,pvc -n sdp
```

At this stage you should see a pod called `proc-pb-mvp01-20200523-00001-workflow-...` and the status is `RUNNING`. To see the logs, run:

```
kubectl logs <pod-name> -n sdp
```

and it should look like this:

```
INFO:batch_imaging:Claimed processing block pb-mvp01-20200523-00001
INFO:batch_imaging:Waiting for resources to be available
INFO:batch_imaging:Resources are available
INFO:batch_imaging:Creating buffer reservations
INFO:batch_imaging:Deploying Dask EE
INFO:batch_imaging:Running simulation pipeline
INFO:batch_imaging:Running ICAL pipeline
...
```

3.3.3 Accessing the data

The buffer reservations are realised as Kubernetes persistent volume claims. They should have persistent volumes created to satisfy them automatically. The name of the corresponding persistent volume is in the output of:

```
kubectl get pvc -n sdp
```

The location of the persistent volume in the filesystem is shown in the output of:

```
kubectl describe pv <pv-name>
```

If you are running Kubernetes with Minikube in a VM, you need to log in to it first to gain access to the files:

```
minikube ssh
```

3.3.4 Running the workflow using the SDP CLI

Deploy SDP and start the console as described at Running SDP stand-alone.

You may also run this workflow directly from the console using the `` `*ska-sdp*` `` CLI <https://developer.skao.int/projects/ska-sdp-config/en/latest/cli.html>`_.

Run the workflow:

```
ska-sdp create pb batch:batch_imaging:0.1.1
```

If you want to change the default parameters, you can run instead as follows (update the JSON string as needed):

```
ska-sdp create pb batch:batch_imaging:0.1.1 '{"n_workers": 4, "freq_min": 0.9e8, "freq_
↪max": 1.1e8}'
```

You can watch the pod being created as before either using

```
kubectl -w get pods -n sdp
```

or k9s. To access the data created by the workflow, follow the steps above at “Accessing the data” in the “Running the workflow using iTango” section.

3.3.5 Changelog

0.1.2

- use latest SDP configuration library

3.4 Delivery workflow

This workflow provides a basic implementation of an SDP Delivery mechanism. It uploads data from SDP buffer reservations to Google Cloud Platform (GCP). It uses Dask as an execution engine.

3.4.1 Parameters

The workflow parameters are:

- `bucket`: name of the GCP storage bucket in which to upload the data
- `buffers`: list of buffers to upload to the storage bucket, each contains
  - `name`: name of the buffer reservation
  - `destination`: location to upload it in the bucket
- `service_account`: location of the GCP service account key (stored in a Kubernetes secret)
  - `secret`: name of the secret
  - `file`: filename of the service account key
- `n_workers`: number of Dask workers to deploy

For example:

```
{
  "bucket": "delivery-test",
  "buffers": [
    {
      "name": "buff-pb-20200523-00000-test",
      "destination": "buff-pb-20200523-00000-test"
    }
  ],
  "service_account": {
    "secret": "delivery-gcp-service-account",
    "file": "service-account.json"
  },
  "n_workers": 1
}
```

3.4.2 Running the workflow using the `ska-sdp` CLI

After SDP is deployed in Minikube, and you started the sdp-console, run:

```
ska-sdp create pb batch:delivery:0.1.0 '<parameters-json>'
```

Replace `<parameters-json>` with the above string and the appropriate values. Once executed, a processing block pod will be created in the `sdp` namespace, which will run the delivery workflow. (See ska-sdp CLI usage.)

Note that each workflow may come with multiple versions. Always use the latest number, unless you know a specific version that suits your needs. (The Changelog at the end of this page may help to decide.)

3.4.3 Creating a GCP storage bucket to receive the data

The steps to create a GCP storage bucket for the delivery workflow are as follows. GCP has an ample documentation, so each step is linked to the relevant section:

1) Create a project.
2) Create a storage bucket in the project.
3) Create a service account and download a key:

```
* The service account must have the role "Storage Object Creator".
* Create and download a key in JSON format.
```

3.4.4 Adding the GCP service account key as a Kubernetes secret

To make the service account key available to the delivery workflow, it needs to be uploaded to the cluster as a Kubernetes secret. The command to do this is:

```
kubectl create secret generic <secret-name> --from-file=<service-account-key> -n <sdp-
↪namespace>
```

Using the values from the example parameters above and assuming the namespace for the SDP dynamic deployments is sdp (the default), the command would be:

```
kubectl create secret generic delivery-gcp-service-account --from-file=service-account.
↪json -n sdp
```

To check the secret has been created, you can use the command:

```
kubectl describe secret delivery-gcp-service-account -n sdp
```

and the output should look like:

```
Name:         delivery-gcp-service-account
Namespace:    sdp
Labels:       <none>
Annotations:  <none>

Type:  Opaque

Data
====
service-account.json:  2382 bytes
```

3.4.5 Changelog

0.1.1

- use python:3.9-slim as the base docker image

3.5 Plasma Pipeline Workflow

This is the integration of the cbf-sdp interface emulation and receive workflow – this includes the plasma interface

3.5.1 Changelog

0.1.4

- Use dependencies from the central artefact repository and publish the workflow image there.

0.1.3

- Ported to use the latest version of workflow library (0.2.4).

0.1.2

- use python:3.9-slim as the base docker image

CHAPTER

FOUR

TEST WORKFLOWS

Workflows that prove the underlying functionality of triggering SDP. These are good starting point for developing different types of workflows.

You can find a detailed description of a batch and a realtime workflow example at *Examples*

4.1 Test Real-Time Workflow

The `test_realtime` workflow is designed to test the processing controller logic concerning processing block dependencies.

The sequence of actions carried out by the workflow is:

- Claims processing block
- Sets processing block `status` to `'WAITING'`
- Waits for `resources_available` to be `True`
  - This is the signal from the processing controller that the workflow can run
- Sets processing block `status` to `'RUNNING'`
- Waits for scheduling block `status` to be set to `FINISHED`
  - This is the signal from the Subarray device that the scheduling block is finished
- Sets processing block `status` to `'FINISHED'`

The workflow makes no deployments.

4.1.1 Changelog

0.2.5

- Use dependencies from the central artefact repository and publish the workflow image there.

0.2.4

- Ported to use the latest version of workflow library (0.2.4).

0.2.3

- use python:3.9-slim as the base docker image

4.2 Test Batch Workflow

The `test_batch` workflow is designed to test the processing controller logic concerning processing block dependencies.

The sequence of actions carried out by the workflow is:

- Claims processing block
- Reads value of `duration` parameter (type: float, units: seconds) from processing block
- Sets processing block `status` to `'WAITING'`
- Waits for `resources_available` to be `True`
    - This is the signal from the processing controller that the workflow can start
- Sets processing block `status` to `'RUNNING'`
- Does some "processing" (i.e. sleeps) for the requested duration
- Sets processing block `status` to `'FINISHED'`

The workflow makes no deployments.

4.2.1 Changelog

0.2.5

- Use dependencies from the central artefact repository and publish the workflow image there.

0.2.4

- Ported to use the latest version of workflow library (0.2.4).

0.2.3

- use python:3.9-slim as the base docker image

4.3 Test Receive Addresses Workflow

4.3.1 Introduction

The purpose of this workflow is to test the mechanism for generating SDP receive addresses from the channel link map for each scan type which is contained in the list of scan types in the SB. The workflow picks it up from there, uses it to generate the receive addresses for each scan type and writes them to the processing block state. It consists of a map of scan type to a receive address map. This address map get publishes to the appropriate attribute once the SDP subarray finishes the transition following AssignResources.

4.3.2 Testing

Deploy SDP and make sure the iTango console pod is also running.

After entering the iTango pod, obtain a handle to a subarray device and turn it on:

```
d = DeviceProxy('mid_sdp/elt/subarray_1')
d.On()
```

If you are not sure what devices are available, list them with `lsdev`.

Here is the configuration string for the scheduling block instance:

```
config = '''
{
  "id": "sbi-mvp01-20200318-0001",
  "max_length": 21600.0,
  "scan_types": [
     {
       "id": "science_A",
       "coordinate_system": "ICRS", "ra": "02:42:40.771", "dec": "-00:00:47.84",
       "channels": [{
          "count": 744, "start": 0, "stride": 2, "freq_min": 0.35e9, "freq_max": 0.368e9,
↪ "link_map": [[0,0], [200,1], [744,2], [944,3]]
       },{
          "count": 744, "start": 2000, "stride": 1, "freq_min": 0.36e9, "freq_max": 0.
↪368e9, "link_map": [[2000,4], [2200,5]]
       }]
     },
     {
       "id": "calibration_B",
       "coordinate_system": "ICRS", "ra": "12:29:06.699", "dec": "02:03:08.598",
       "channels": [{
          "count": 744, "start": 0, "stride": 2, "freq_min": 0.35e9, "freq_max": 0.368e9,
↪ "link_map": [[0,0], [200,1], [744,2], [944,3]]
       },{
          "count": 744, "start": 2000, "stride": 1, "freq_min": 0.36e9, "freq_max": 0.
↪368e9, "link_map": [[2000,4], [2200,5]]
       }]
     }
   ],
  "processing_blocks": [
    {
```

(continues on next page)

(continued from previous page)

```
      "id": "pb-mvp01-20200318-0001",
      "workflow": {"type": "realtime", "id": "test_receive_addresses", "version": "0.3.4
↪"},
      "parameters": {}
    },
    {
      "id": "pb-mvp01-20200318-0002",
      "workflow": {"type": "realtime", "id": "test_realtime", "version": "0.2.2"},
      "parameters": {}
    }
  ]
} '''
```

Note that each workflow may come with multiple versions. Always use the latest number, unless you know a specific version that suits your needs. (The Changelog at the end of this page may help to decide.)

Start the scheduling block instance by the AssignResources command:

```
d.AssignResources(config)
```

You can connect to the configuration database by running the following command:

```
kubectl exec -it sdp-console-0 -- bash
```

and from there to see the full list of entries, run

```
ska-sdp list -a
```

To check if the receive addresses are updated in the processing block state correctly, run the following command:

```
ska-sdp get pb pb-mvp01-20200318-0001/state
```

and the output should look like this:

```
/pb/pb-mvp01-20200318-0001/state = {
  "receive_addresses": {
    "calibration_B": {
      "host": [
        [
          0,
          "192.168.0.1"
        ],
        [
          2000,
          "192.168.0.1"
        ]
      ],
      "port": [
        [
          0,
          9000,
          1
        ],
```

(continues on next page)

(continued from previous page)

```
        [
          2000,
          9000,
          1
        ]
      ]
    },
    "science_A": {
      "host": [
        [
          0,
          "192.168.0.1"
        ],
        [
          2000,
          "192.168.0.1"
        ]
      ],
      "port": [
        [
          0,
          9000,
          1
        ],
        [
          2000,
          9000,
          1
        ]
      ]
    }
  },
  "resources_available": true,
  "status": "RUNNING"
}
```

To access the SBI run this

```
ska-sdp get /sb/sbi-mvp01-20200318-0001
```

In there you should see that pb_receive_addresses is updated with the PB_ID.

This should now update the receiveAddresses attribute with receive addresses map and that can be verified by running d.receiveAddresses and the output should look like this:

```
Out[4]: '{"calibration_B": {"host": [[0, "192.168.0.1"], [2000, "192.168.0.1"]], "port":␣
↪[[0, 9000, 1], [2000, 9000, 1]]}, "science_A": {"host": [[0, "192.168.0.1"], [2000,
↪"192.168.0.1"]], "port": [[0, 9000, 1], [2000, 9000, 1]]}}'
```

4.3.3 Changelog

0.3.7

- Use dependencies from the central artefact repository and publish the workflow image there.

0.3.6

- Ported to use the latest version of workflow library (0.2.4).

0.3.5

- use python:3.9-slim as the base docker image

4.4 Test Dask Workflow

The `test_dask` workflow is designed to test deploying two instances of a Dask execution engine and executing a simple function on each one.

The sequence of actions carried out by the workflow is:

- Claims processing block
- Sets processing block `status` to `'WAITING'`
- Waits for `resources_available` to be `True`
  - This is the signal from the processing controller that the workflow can start
- Sets processing block `status` to `'RUNNING'`
- Deploys two Dask execution engines in parallel
- Does some simple operations. Constructs a graph to add two numbers together and computes the result by calling the 'compute' method.
- Sets processing block `status` to `'FINISHED'`

4.4.1 Changelog

0.2.6

- Use dependencies from the central artefact repository and publish the workflow image there.

0.2.5

- Ported to use the latest version of workflow library (0.2.4).

0.2.4

- use python:3.9-slim as the base docker image
- slimmed down the requirements file as well

4.5 Test Daliuge Workflow

Not yet imported to use the workflow library. To be updated.

4.5.1 Changelog

0.2.2

- Use dependencies from the central artefact repository and publish the workflow image there.

0.2.1

- use python:3.8-slim as the base docker image

CHAPTER

FIVE

TROUBLESHOOTING

Collection of step-by-step articles with solutions to known issues that you may encounter while developing/deploying workflows.

5.1 Device does not finish RESOURCING after running AssignResources()

Problem:

Occurrence:

Solution:

CHAPTER SIX

ARCHIVE

Documentation that is no longer maintained but can be of interest for some.

6.1 Build a custom execution engine

Note, this section is OUTDATED!

If you want to use a custom execution engine (EE) in your workflow, the additional steps you need to do are:

- Create a directory in `src` for your EE.
- Add the EE code.
- Build the EE Docker image(s) and push it/them to the Nexus repository.
- Add a Helm chart to deploy the EE containers in `src/helm_deploy/charts`.
- Add the custom EE deployment to the workflow script.
- Commit changes to your branch and push to GitLab.
- When testing, you also need to point the Helm deployer to your branch of the repository:

```
$ helm install sdp-prototype -n sdp-prototype \
  --set processing_controller.workflows.url=https://gitlab.com/ska-telescope/sdp-
↪prototype/raw/<my-branch>/src/workflows/workflows.json \
  --set helm_deploy.chart_repo.ref=<my-branch>
```