developer.skatelescope.org Documentation

Release 0.1.1-beta

Marco Bartolini

Oct 17, 2020

CONTENTS

This repository contains a set of packages for deploying a minimal SDP system capable of configuring and executing workflows.

CHAPTER

ONE

GETTING STARTED

1.1 I want to..

1.1.1 Understand the design of the SDP prototype

Documentation can be found at:

- *Design Overview*
- *Components*
- *Module View*

1.1.2 Set up the SDP prototype in a local development environment

Instructions can be found at *Setting up a local development environment*.

1.1.3 Run the SDP prototype stand-alone

First you need to make sure the local development environment is set up. The details on how to run the prototype stand-alone can be found at *Running the SDP Prototype stand-alone*.

1.1.4 Run the SDP Prototype in the integration environment

Details can be found at *Running the SDP Prototype in the integration environment*.

1.1.5 Find out about the SDP Tango devices

Details on the interface and Python API for the SDP Master device can be found at *SDP Master Device*.

Details on the interface and Python API for the SDP Subarray device can be found at *SDP Subarray Device*.

1.1.6 Know more about the SDP configuration database

An overview on how to access SKA SDP configuration information can be found at *Configuration Database*.

Details on the configuration schema can be found at *Configuration Schema*.

API details of the configuration database can be found here *Configuration API*.

1.1.7 Understand the design of the services

The documentation on the processing controller service can be found at *Processing Controller*.

The documentation on the Helm deployer service can be found at *Helm Deployer*.

1.1.8 Run workflows

Instructions on how to run the visibility receive workflow can be found at *Visibility Receive Workflow*.

Details on how to run the PSS receive can be found at *PSS Receive Workflow*.

Instructions on how to run the test workflows can be found at *Test Workflows*.

1.1.9 Develop a workflow

Instructions on how to develop and test a workflow can be found at *Workflow Development*.

CHAPTER

TWO

DESIGN OVERVIEW

2.1 Introduction

This prototype is a partial implementation of the SDP software architecture adopted by the SKA. Its purpose is to implement and test parts of the architecture to de-risk the construction of the SDP.

The most recent version of the complete SDP architecture can be found in the SDP Consortium close-out documentation. The architecture is intended to be a living document that evolves alongside its implementation, so it will eventually be available in a form that can more readily be changed.

CHAPTER

THREE

COMPONENTS

Fig. 1: Component and connector diagram of the prototype implementation.

Execution Control:

- The **SDP Master Tango Device** is intended to provide the top-level control of SDP services. The present implementation does very little, apart from executing internal state transitions in response to Tango commands. As shown in the diagram, it does not yet have a connection to the Configuration Database.
- The **SDP Subarray Tango Devices** control the processing associated with SKA Subarrays. When a Processing Block is submitted to SDP through one of the devices, it is added to the Configuration Database. During the execution of the Processing Block, the device publishes the status of the Processing Block through its attributes.
- The **Processing Controller** controls the execution of Processing Blocks. It detects them by monitoring the Configuration Database. To execute a Processing Block, it requests the deployment of the corresponding Workflow by creating an entry in the Configuration Database.
- The **Configuration Database** is the central store of configuration information in the SDP. It is the means by which the components communicate with each other.

Platform:

- The **Helm Deployer** is the service that the Platform uses to respond to deployment requests in the Configuration Database. It makes deployments by installing Helm charts (a collection of files that describe a related set of Kubernetes resources) into a Kubernetes cluster.
- **Kubernetes** is the underlying mechanism for making dynamic deployments of Workflows and Execution Engines.

Processing Block Deployment:

- A **Workflow** controls the execution of a Processing Block (in the architecture it is called the Processing Block Controller). Workflows connect to the Configuration Database to retrieve the parameters defined in the Processing Block and to request the deployment of Execution Engines.
- **Execution Engines** are the means by which Workflows process the data. They typically enable distributed execution of processing functions, although Workflows may use a single process as a serial Execution Engine.

CHAPTER

FOUR

MODULE VIEW

CHAPTER

FIVE

SETTING UP A LOCAL DEVELOPMENT ENVIRONMENT

5.1 Kubernetes

You will need Kubernetes installed. Docker for Desktop includes a workable one-node Kubernetes installation - you just need to activate it in the settings. Alternatively, you can install Minikube or microk8s.

5.1.1 Docker and Minikube

Docker runs containers in a VM on Windows and macOS, and by default Minikube does this on all systems. The VM needs at least 3 GB of memory to run the SDP prototype. In Docker this can be found in the settings. For Minikube you need to specify the amount of memory on the command line when starting a new instance:

```
$ minikube start --memory='4096m'
```

5.1.2 Micro8ks

Canonical supports microk8s for Ubuntu Linux distributions - and it is also available for many other distributions (42 according to this). It gives a more-or-less 'one line' Kubernetes installation

- To install type `sudo snap install microk8s --classic`
- `microk8s.start` will start the Kubernetes system
- `microk8s.enable dns` is required for the SDP prototype
- `microk8s.status` should show that things are active
- `microk8s.inspect` shows the report in more detail
- microk8s will install `kubectl` as `microk8s.kubectl`. Unless you have another Kubernetes installation in parallel you may wish to set up an alias with `sudo snap alias microk8s.kubectl kubectl`

If you have problems with pods not communicating, you may need to do `sudo iptables -P FORWARD ACCEPT` (`microk8s.inspect` should be able to diagnose this for you).

5.2 Helm

Furthermore you will need to install the Helm utility. It is available from most typical package managers, see Introduction to Helm. We recommend using Helm 3.

If you are using Helm 3 for the first time, you need to add the stable chart repository:

```
$ helm repo add stable https://kubernetes-charts.storage.googleapis.com/
```

5.3 Setting up on Windows

5.3.1 Install and configure tools

The hypervisor Hyper-V is built-in on Windows 10 and just needs to be enabled via settings and a reboot.

Install Minikube, kubectl, Helm 3 and put the executables in the path.

Configure Minikube to use 4GB of memory:

```
> minikube config set memory 4096
```

This creates a file `.minikube/config/config.json` that looks like this:

```
{
  "dashboard": true,
  "memory": 4096
}
```

5.3.2 Fix the line ends

A git clone will by default automatically convert all line ends to Windows format. This causes the SDP devices pod to fail to start. The command to see the error and the resulting message looks like this:

```
> kubectl logs test-sdp-prototype-sdp-devices-845969f6b8-s9nhf -c dsconfig

wait-for-it.sh: waiting 30 seconds for databaseds-tango-base-sdp-prototype:10000
wait-for-it.sh: databaseds-tango-base-sdp-prototype:10000 is available after 0 seconds
Traceback (most recent call last):
File "/usr/local/bin/json2tango", line 11, in <module>
sys.exit(main())
File "/usr/local/lib/python2.7/dist-packages/dsconfig/json2tango.py", line 88, in main
with open(json_file) as f:
IOError: [Errno 2] No such file or directory: 'data/sdp-devices.json\r'
data/sane-dsconfig.sh: line 7: syntax error near unexpected token `fi'
data/sane-dsconfig.sh: line 7: `fi'
```

To fix this, create a file .gitattributes in the top-level project with these contents:

```
*.json text eol=lf
*.sh text eol=lf
*.yml text eol=lf
*.yaml text eol=lf
```

Then run the commands:

```
> git rm --cached -r .
> git reset --hard
> git add --renormalize .
```

CHAPTER

SIX

RUNNING THE SDP PROTOTYPE STAND-ALONE

6.1 Installing the etcd operator

The SDP configuration database is implemented on top of etcd, a strongly consistent, distributed key-value store that provides a reliable way to store data that needs to be accessed by a distributed system or cluster of machines.

Before deploying the SDP itself, you need to install the `etcd-operator` Helm chart. This provides a convenient way to create and manage etcd clusters in other charts.

If you have a fresh install of Helm, you need to add the `stable` repository:

```
$ helm repo add stable https://kubernetes-charts.storage.googleapis.com/
```

The `sdp-prototype` charts directory contains a file called `etcd-operator.yaml` with settings for the chart. This turns off parts which are not used (the backup and restore operators).

First go to the charts directory:

```
$ cd [sdp-prototype]/charts
```

Then install the `etcd-operator` chart with:

```
$ helm install etcd stable/etcd-operator -f etcd-operator.yaml
```

If you now execute:

```
$ kubectl get pod --watch
```

You should eventually see an pod called `etcd-etcd-operator-etcd-operator-[...]` in 'Running' state (yes, Helm is exceedingly redundant with its names). If not wait a bit, if you try to go to the next step before this has completed there is a chance it will fail.

6.2 Deploying the SDP

At this point you should be able to deploy the SDP. Install the `sdp-prototype` chart with the release name `test`:

```
$ helm install test sdp-prototype
```

You can again watch the deployment in progress using `kubectl`:

```
$ kubectl get pod --watch
```

Pods associated with Tango might go down a couple times before they start correctly, this seems to be normal. You can check the logs of pods (copy the full name from `kubectl` output) to verify that they are doing okay:

```
$ kubectl logs test-sdp-prototype-lmc-[...] sdp-subarray-1
1|2020-08-06T15:17:41.369Z|INFO|MainThread|init_device|subarray.py
↪#110|SDPSubarray|Initialising SDP Subarray: mid_sdp/elt/subarray_1
...
1|2020-08-06T15:17:41.377Z|INFO|MainThread|init_device|subarray.py
↪#140|SDPSubarray|SDP Subarray initialised: mid_sdp/elt/subarray_1
$ kubectl logs test-sdp-prototype-processing-controller-[...]
...
1|2020-08-06T15:14:30.068Z|DEBUG|MainThread|main|processing_controller.py
↪#192||Waiting...
$ kubectl logs test-sdp-prototype-helm-deploy-[...]
...
1|2020-08-06T15:14:31.662Z|INFO|MainThread|main|helm_deploy.py#146||Found 0 existing␣
↪deployments.
```

If it looks like this, there is a good chance everything has been deployed correctly.

6.3 Testing it out

6.3.1 Connecting to the configuration database

By default the `sdp-prototype` chart deploys a 'console' pod which enables you to interact with the configuration database. You can start a shell in the pod by doing:

```
$ kubectl exec -it deploy/test-sdp-prototype-console -- /bin/bash
```

This will allow you to use the `sdpcfg` command:

```
# sdpcfg ls -R /
Keys with / prefix:
```

Which correctly shows that the configuration is currently empty.

6.3.2 Starting a workflow

Assuming the configuration is prepared as explained in the previous section, we can now add a processing block to the configuration:

```
# sdpcfg process batch:test_dask:0.2.1
OK, pb_id = pb-sdpcfg-20200425-00000
```

The processing block is created with the `/pb` prefix in the configuration:

```
# sdpcfg ls values -R /pb
Keys with /pb prefix:
/pb/pb-sdpcfg-20200425-00000 = {
  "dependencies": [],
  "id": "pb-sdpcfg-20200425-00000",
  "parameters": {},
  "sbi_id": null,
  "workflow": {
```

(continues on next page)

(continued from previous page)

```
    "id": "test_dask",
    "type": "batch",
    "version": "0.2.0"
  }
}
/pb/pb-sdpcfg-20200425-00000/owner = {
  "command": [
    "testdask.py",
    "pb-sdpcfg-20200425-00000"
  ],
  "hostname": "proc-pb-sdpcfg-20200425-00000-workflow-7pfkl",
  "pid": 1
}
/pb/pb-sdpcfg-20200425-00000/state = {
  "resources_available": true,
  "status": "RUNNING"
}
```

The processing block is detected by the processing controller which deploys the workflow. The workflow in turn deploys the execution engines (in this case, Dask). The deployments are requested by creating entries with `/deploy` prefix in the configuration, where they are detected by the Helm deployer which actually makes the deployments:

```
# sdpcfg ls values -R /deploy
Keys with /deploy prefix:
/deploy/proc-pb-sdpcfg-20200425-00000-dask = {
  "args": {
    "chart": "stable/dask",
    "values": {
      "jupyter.enabled": "false",
      "scheduler.serviceType": "ClusterIP",
      "worker.replicas": 2
    }
  },
  "id": "proc-pb-sdpcfg-20200425-00000-dask",
  "type": "helm"
}
/deploy/proc-pb-sdpcfg-20200425-00000-workflow = {
  "args": {
    "chart": "workflow",
    "values": {
      "env.SDP_CONFIG_HOST": "test-sdp-prototype-etcd-client.default.svc.cluster.local
↪",
      "env.SDP_HELM_NAMESPACE": "sdp",
      "pb_id": "pb-sdpcfg-20200425-00000",
      "wf_image": "nexus.engageska-portugal.pt/sdp-prototype/workflow-test-dask:0.2.0"
    }
  },
  "id": "proc-pb-sdpcfg-20200425-00000-workflow",
  "type": "helm"
}
```

The deployments associated with the processing block have been created in the `sdp` namespace, so to view the created pods we have to ask as follows (on the host):

```
$ kubectl get pod -n sdp
NAME                                                              READY   STATUS    ␣
↪RESTARTS   AGE
```

(continues on next page)

(continued from previous page)

```
proc-pb-sdpcfg-20200425-00000-dask-scheduler-78b4974ddf-w4x8x   1/1     Running   0  ␣
↪       4m41s
proc-pb-sdpcfg-20200425-00000-dask-worker-85584b4598-p6qpw      1/1     Running   0  ␣
↪       4m41s
proc-pb-sdpcfg-20200425-00000-dask-worker-85584b4598-x2bh5      1/1     Running   0  ␣
↪       4m41s
proc-pb-sdpcfg-20200425-00000-workflow-7pfkl                    1/1     Running   0  ␣
↪       4m46s
```

6.3.3 Cleaning up

Finally, let us remove the processing block from the configuration (in the SDP console shell):

```
# sdpcfg delete -R /pb/pb-sdpcfg-20200425-00000
/pb/pb-sdpcfg-20200425-00000
/pb/pb-sdpcfg-20200425-00000/owner
/pb/pb-sdpcfg-20200425-00000/state
OK
```

If you re-run the commands from the last section you will notice that this correctly causes all changes to the cluster configuration to be undone as well.

6.4 Accessing Tango

By default the sdp-prototype chart installs the iTango shell pod from the tango-base chart. You can access it as follows:

```
$ kubectl exec -it itango-tango-base-sdp-prototype -- /venv/bin/itango3
```

You should be able to query the SDP Tango devices:

```
In [1]: lsdev
Device                                  Alias                     Server           ␣
↪       Class
---------------------------------------- ------------------------- ------------------
↪------ --------------------
mid_sdp/elt/master                                                SdpMaster/1      ␣
↪       SdpMaster
mid_sdp/elt/subarray_1                                            SdpSubarray/1    ␣
↪       SdpSubarray
mid_sdp/elt/subarray_2                                            SdpSubarray/2    ␣
↪       SdpSubarray
sys/access_control/1                                              TangoAccessControl/
↪1      TangoAccessControl
sys/database/2                                                    DataBaseds/2     ␣
↪       DataBase
sys/rest/0                                                        TangoRestServer/
↪rest      TangoRestServer
sys/tg_test/1                                                     TangoTest/test   ␣
↪       TangoTest
```

This allows direct interaction with the devices, such as querying and and changing attributes and issuing commands:

```
In [2]: d = DeviceProxy('mid_sdp/elt/subarray_1')

In [3]: d.state()
Out[3]: tango._tango.DevState.OFF

In [4]: d.On()

In [5]: d.state()
Out[5]: tango._tango.DevState.ON

In [6]: d.obsState
Out[6]: <obsState.EMPTY: 0>

In [7]: config_sbi = '''
    ...: {
    ...:   "id": "sbi-mvp01-20200425-00000",
    ...:   "max_length": 21600.0,
    ...:   "scan_types": [
    ...:     {
    ...:       "id": "science",
    ...:       "channels": [
    ...:         {"count": 372, "start": 0, "stride": 2, "freq_min": 0.35e9, "freq_max
↪": 0.358e9, "link_map": [[0,0], [200,1]]}
    ...:       ]
    ...:     }
    ...:   ],
    ...:   "processing_blocks": [
    ...:     {
    ...:       "id": "pb-mvp01-20200425-00000",
    ...:       "workflow": {"type": "realtime", "id": "test_realtime", "version": "0.
↪1.0"},
    ...:       "parameters": {}
    ...:     },
    ...:     {
    ...:       "id": "pb-mvp01-20200425-00001",
    ...:       "workflow": {"type": "realtime", "id": "test_realtime", "version": "0.
↪1.0"},
    ...:       "parameters": {}
    ...:     },
    ...:     {
    ...:       "id": "pb-mvp01-20200425-00002",
    ...:       "workflow": {"type": "batch", "id": "test_batch", "version": "0.1.0"},
    ...:       "parameters": {},
    ...:       "dependencies": [
    ...:         {"pb_id": "pb-mvp01-20200425-00000", "type": ["visibilities"]}
    ...:       ]
    ...:     },
    ...:     {
    ...:       "id": "pb-mvp01-20200425-00003",
    ...:       "workflow": {"type": "batch", "id": "test_batch", "version": "0.1.0"},
    ...:       "parameters": {},
    ...:       "dependencies": [
    ...:         {"pb_id": "pb-mvp01-20200425-00002", "type": ["calibration"]}
    ...:       ]
    ...:     }
    ...:   ]
    ...: }
```

(continues on next page)

(continued from previous page)

```
    ...: '''

In [8]: d.AssignResources(config_sbi)

In [9]: d.obsState
Out[9]: <obsState.IDLE: 0>

In [10]: d.Configure('{"scan_type": "science"}')

In [11]: d.obsState
Out[11]: <obsState.READY: 2>

In [12]: d.Scan('{"id": 1}')

In [13]: d.obsState
Out[13]: <obsState.SCANNING: 3>

In [14]: d.EndScan()

In [15]: d.obsState
Out[15]: <obsState.READY: 2>

In [16]: d.End()

In [17]: d.obsState
Out[17]: <obsState.IDLE: 0>

In [18]: d.ReleaseResources()

In [19]: d.obsState
Out[19]: <obsState.EMPTY: 0>

In [20]: d.Off()

In [21]: d.state()
Out[21]: tango._tango.DevState.OFF
```

6.5 Removing the SDP

To remove the SDP deployment from the cluster, do:

```
$ helm uninstall test
```

and to remove the etcd operator, do:

```
$ helm uninstall etcd
```

6.6 Troubleshooting

6.6.1 etcd doesn't start (DNS problems)

Something that often happens on home set-ups is that `test-sdp-prototype-etcd` does not start, which means that quite a bit of the SDP system will not work. Try executing `kubectl logs` on the pod to get a log. You might see something like this as the last three lines:

```
... I | pkg/netutil: resolving sdp-prototype-etcd-9s4hbbmmvw.k8s-sdp-prototype-etcd.
↪default.svc:2380 to 10.1.0.21:2380
... I | pkg/netutil: resolving sdp-prototype-etcd-9s4hbbmmvw.k8s-sdp-prototype-etcd.
↪default.svc:2380 to 92.242.132.24:2380
... C | etcdmain: failed to resolve http://sdp-prototype-etcd-9s4hbbmmvw.sdp-
↪prototype-etcd.default.svc:2380 to match --initial-cluster=sdp-prototype-etcd-
↪9s4hbbmmvw=http://sdp-prototype-etcd-9s4hbbmmvw.sdp-prototype-etcd.default.svc:2380␣
↪("http://10.1.0.21:2380"(resolved from "http://sdp-prototype-etcd-9s4hbbmmvw.sdp-
↪prototype-etcd.default.svc:2380") != "http://92.242.132.24:2380"(resolved from
↪"http://sdp-prototype-etcd-9s4hbbmmvw.sdp-prototype-etcd.default.svc:2380"))
```

This informs you that etcd tried to resolve its own address, and for some reason got two different answers both times. Interestingly, the `92.242.132.24` address is not actually in-cluster, but from the Internet, and re-appears if we attempt to `ping` a nonexistent DNS name:

```
$ ping does.not.exist
Pinging does.not.exist [92.242.132.24] with 32 bytes of data:
Reply from 92.242.132.24: bytes=32 time=25ms TTL=242
```

What is going on here is that that your ISP has installed a DNS server that redirects unknown DNS names to a server showing a 'helpful' error message complete with a bunch of advertisements. For some reason this seems to cause a problem with Kubernetes' internal DNS resolution.

How can this be prevented? Theoretically it should be enough to force the DNS server to one that does not have this problem (like Google's `8.8.8.8` and `8.8.4.4` DNS servers), but that is tricky to get working. Alternatively you can simply restart the entire thing until it works. Unfortunately this is not quite as straightforward with `etcd-operator`, as it sets the `restartPolicy` to `Never`, which means that any `etcd` pod only gets once chance, and then will remain `Failed` forever. The quickest way seems to be to delete the `EtcdCluster` object, then `upgrade` the chart in order to re-install it:

```
$ kubectl delete etcdcluster test-sdp-prototype-etcd
$ helm upgrade test sdp-prototype
```

This can generally be repeated until by pure chance the two DNS resolutions return the same result and `etcd` starts up.

CHAPTER

SEVEN

RUNNING THE SDP PROTOTYPE IN THE INTEGRATION ENVIRONMENT

The SDP prototype is being integrated with the prototypes of the other telescope sub-systems as part of the so-called *Minimum Viable Product* (MVP).

The integration is done in the SKA MVP Prototype Integration (SKAMPI) repository.

Instructions for installing and running the MVP can be found in the SKAMPI documentation.

To then deploy an SDP workflow without using WebJive etc., follow the instructions in the standalone SDP documentation. However, you will need to append the namespace in which the SDP is running.

```
$ kubectl exec -it deploy/test-sdp-prototype-console -- /bin/bash -n <namespace>
```

The default namespace into which *skampi* deploys is *-n integration*.

Alternatively, you may run

```
$ kubectl config set-context --current --namespace=integration
```

to allow you to run the commands without alteration.

CHAPTER

EIGHT

SDP MASTER DEVICE

8.1 Introduction

The SDP Master Tango device is designed to provide the overall control of the SDP. The commands it receives cause the other SDP services to be stopped or started, and its attributes report on the overall state of the system.

The present implementation of the SDP Master device does very little apart from performing the state transitions in response to commands.

8.2 Interface

8.2.1 Attributes

Device attributes:

Attribute	Type	Read/Write	Values	Description
version	String	Read	Semantic version	Master device server version
heathState	Enum	Read	*healthState values*	SDP health state

healthState values

healthState	Description
OK (0)	
DEGRADED (1)	
FAILED (2)	
UNKNOWN (3)	

8.2.2 Commands

The commands change the device state as described below, but at present they have no other effect on SDP.

Command	Argument type	Return type	Action
On	None	None	Set device state to ON
Disable	None	None	Set device state to DISABLE
Standby	None	None	Set device state to STANDBY
Off	None	None	Set device state to OFF

8.3 Python API

CHAPTER

NINE

SDP SUBARRAY DEVICE

9.1 Introduction

The SDP Subarray Tango device is the principal means by which processing is initiated in SDP.

9.2 State Model

The present implementation is shown in the diagram below. Here the state is the combination of the Tango device state and the observing state (obsState).

9.3 Behaviour

The interaction between TMC (Telescope Manager Control) and the SDP Subarray device is shown below. The SDP Subarray device receives commands from the TMC SDP Subarray leaf node, and the consequent changes to the state of SDP are reported in the device attributes.

9.4 Interface

9.4.1 Attributes

Attribute	Type	Read/Write	Values	Description
version	String	Read	Semantic version	Subarray device server version
obsState	Enum	Read	*obsState values*	Subarray observing state
adminMode	Enum	Read-write	*adminMode values*	Subarray admin mode
healthState	Enum	Read	*healthState values*	Subarray health state
receiveAddresses	String	Read	JSON object	Host addresses for receiving visibilities
schedulingBlockInstance	String	Read	JSON object	State of Scheduling Block Instance
processingBlockState	String	Read	JSON object	State of associated real-time Processing Blocks

obsState values

obsState	Description
EMPTY (0)	No receive and real-time processing resources are assigned to the subarray
RESOURCING (1)	Resources are being assigned or released
IDLE (2)	Receive and real-time processing resources are assigned to the subarray as specified in the Scheduling Block Instance
CONFIGURING (3)	Scan type is being configured
READY (4)	Scan type is configured and the subarray is ready to scan
SCANNING (5)	Scanning
ABORTING (6)	Current activity is being aborted
ABORTED (7)	Most recent activity has been aborted
RESETTING (8)	Resetting to IDLE obsState
FAULT (9)	A fault has occurred in observing
RESTARTING (10)	Restarting in EMPTY obsState

adminMode values

adminMode	Description
OFFLINE (0)	
ONLINE (1)	
MAINTENANCE (2)	
NOT_FITTED (3)	
RESERVED (4)	

healthState values

healthState	Description
OK (0)	
DEGRADED (1)	
FAILED (2)	
UNKNOWN (3)	

9.4.2 Commands

Command	Argument type	Return type	Action
On	None	None	Sets the device state to ON and obsState to EMPTY.
Off	None	None	Sets the device state to OFF.
AssignResources	String (JSON)	None	*Assigns processing resources to the SBI. Sets obsState to IDLE*.
ReleaseResources	None	None	Releases all real-time processing in the SBI. Sets obsState to EMPTY.
Configure	String (JSON)	None	*Configures scan type for the next scans. Sets obsState to READY*.
Scan	String (JSON)	None	*Begins a scan of the configured type. Sets obsState to SCANNING*.
EndScan	None	None	Ends the scan. Sets obsState to READY.
End	None	None	Clears the scan type. Sets obsState to IDLE.
Abort	None	None	Aborts current activity. Sets obsState to ABORTED.
ObsReset	None	None	Resets to last known stable state. Sets obsState to IDLE.
Restart	None	None	Restarts the subarray device. Sets obsState to EMPTY.

AssignResources command

The argument of the AssignResources command is a JSON object describing the processing to be done for the scheduling block instance (SBI). It contains a set of scan types and processing blocks. The scan types contain information about the frequency channels output by CSP, which is important for configuring the receive processes in SDP. The processing blocks define the workflows to be run and the parameters to be passed to the workflows.

An example of the argument is below. Note that:

- `max_length` specifies the maximum length of the SBI in seconds.
- In `scan_types`, the channel information is for example only.
- In `processing_blocks`, the workflow parameters will not actually be empty. Each workflow will have its own schema for its parameters.

```
{
  "id": "sbi-mvp01-20200425-00000",
  "max_length": 21600.0,
  "scan_types": [
    {
      "id": "science",
      "channels": [
        {"count": 372, "start": 0, "stride": 2, "freq_min": 0.35e9, "freq_max": 0.
↪358e9, "link_map": [[0,0], [200,1]]}
      ]
    },
    {
      "id": "calibration",
      "channels": [
        {"count": 372, "start": 0, "stride": 2, "freq_min": 0.35e9, "freq_max": 0.
↪358e9, "link_map": [[0,0], [200,1]]}
      ]
    }
  ],
```

(continues on next page)

(continued from previous page)

```
  "processing_blocks": [
    {
      "id": "pb-mvp01-20200425-00000",
      "workflow": {"type": "realtime", "id": "test_receive_addresses", "version": "0.
↪3.2"},
      "parameters": {}
    },
    {
      "id": "pb-mvp01-20200425-00001",
      "workflow": {"type": "realtime", "id": "test_realtime", "version": "0.2.0"},
      "parameters": {}
    },
    {
      "id": "pb-mvp01-20200425-00002",
      "workflow": {"type": "batch", "id": "test_batch", "version": "0.2.0"},
      "parameters": {},
      "dependencies": [
        {"pb_id": "pb-mvp01-20200425-00000", "type": ["visibilities"]}
      ]
    },
    {
      "id": "pb-mvp01-20200425-00003",
      "workflow": {"type": "batch", "id": "test_batch", "version": "0.2.0"},
      "parameters": {},
      "dependencies": [
        {"pb_id": "pb-mvp01-20200425-00002", "type": ["calibration"]}
      ]
    }
  ]
}
```

Configure command

The argument of the Configure command is a JSON object specifying the scan type for the next scans. `new_scan_types` is optional, it is only present if a new scan type needs to be declared. This would only happen for special SBIs (and underlying SDP workflows) meant to support dynamic reconfiguration.

An example of the argument:

```
{
  "new_scan_types": [
    {
      "id": "new_calibration",
      "channels": [
        {"count": 372, "start": 0, "stride": 2, "freq_min": 0.35e9, "freq_max": 0.
↪358e9, "link_map": [[0,0], [200,1]]}
      ]
    }
  ],
  "scan_type": "new_calibration"
}
```

Scan command

The argument of the Scan command is a JSON object which specifies the scan ID.

An example of the argument:

```
{
  "id": 1
}
```

9.5 Python API

CHAPTER

TEN

BUILDING AND TESTING

Placeholder for the merged documentation on building and testing the Tango devices.

CHAPTER

ELEVEN

CONFIGURATION DATABASE

This is the frontend module for accessing SKA SDP configuration information. It provides ways for SDP controller and processing components to discover and manipulate the intended state of the system.

At the moment this is implemented on top of `etcd`, a highly-available database. This library provides primitives for atomic queries and updates to the stored configuration information.

11.1 Installation

Install from PyPI:

```
pip install ska-sdp-config
```

11.2 Basic Usage

Make sure you have a database backend accessible (etcd3 is supported at the moment). Location can be configured using the `SDP_CONFIG_HOST` and `SDP_CONFIG_PORT` environment variables. The defaults are `127.0.0.1` and `2379`, which should work with a local `etcd` started without any configuration.

This should give you access to SDP configuration information, for instance try:

```
import ska_sdp_config

config = ska_sdp_config.Config()

for txn in config.txn():
    for pb_id in txn.list_processing_blocks():
        pb = txn.get_processing_block(pb_id)
        print("{} ({}:{})".format(pb_id, pb.workflow['id'], pb.workflow['version']))
```

To read a list of currently active processing blocks with their associated workflows.

11.3 Command line

This package also comes with a command line utility for easy access to configuration data. For instance run:

```
sdpcfg list values /pb/
```

To query all processing blocks.

CHAPTER

TWELVE

CONFIGURATION SCHEMA

This is the schema of the configuration database, effectively the control plane of the SDP.

12.1 Design Principles

- Uses a key-value store
- Uses watches on a key or range of keys to monitor for any updates
- Objects are represented as JSON
- We will likely want to define schemas and validation eventually, but for the moment this will be by example

12.2 Scheduling Block

Path `/sb/[sbi_id]`

Dynamic state information of the scheduling block instance.

Contents:

```
{
    "id": "sbi-mvp01-20200425-00000",
    "max_length": 21600.0
    "scan_types": [
        { "id": "science", ... },
        { "id": "calibration", ... }
    ]
    "pb_realtime": [ "pb-mvp01-20200425-00000", ... ]
    "pb_batch": [ ... ]
    "pb_receive_addresses": "pb-mvp01-20200425-00000"
    "current_scan_type": "science"
    "status": "SCANNING"
    "scan_id": 12345
}
```

When the scheduling block instance is being executed, the `status` field is set to the observation state (`obsState`) of the subarray. When the scheduling block is ended, `status` is set to `FINISHED`.

12.3 Processing Block

Path: `/pb/[pb_id]`

Static definition of processing block information.

Contents:

```
{
    "id": "pb-mvp01-20200425-00000",
    "sbi_id": "sbi-mvp01-20200425-00000",
    "workflow": {
        "type": "realtime",
        "id": "vis_receive",
        "version": "0.2.0"
    }
    "parameters": { ... }
}
```

There are two types of processing, real-time processing and batch (offline) processing. Real-time processing starts immediately, as it directly corresponds to an observation that is about to start. Batch processing will be inserted into a scheduling queue managed by the SDP, where it will typically be executed according to resource availability.

Valid types are `realtime` and `batch`. The workflow tag identifies the workflow script version as well as the required underlying software (e.g. execution engines, processing components). `...` stands for arbitrary workflow-defined parameters.

12.3.1 Processing Block State

Path: `/pb/[pb_id]/state`

Dynamic state information of the processing block. If it does not exist, the processing block is still starting up.

Contents:

```
{
    "resources_available": True
    "status": "RUNNING",
    "receive_addresses": [
        { "scan_type": "science", ... },
        { "scan_type": "calibration", ... },
    ]
}
```

Tracks the current state of the processing block. This covers both the SDP-internal state (as defined by the Execution Control Data Model) as well as information to publish via Tango for real-time workflows, such as the status and receive addresses (for ingest).

12.3.2 Processing Block Owner

Path: `/pb/[pb_id]/owner`

Identifies the process executing the workflow. Used for leader election/lock as well as a debugging aid.

Contents:

```
{
  "command": [
    "vis_receive.py",
    "pb-mvp01-20200425-00000"
  ],
  "hostname": "pb-mvp01-20200425-00000-workflow-2kxfz",
  "pid": 1
}
```

CHAPTER

THIRTEEN

CONFIGURATION API

13.1 High-Level API

High-level API for SKA SDP configuration.

class ska_sdp_config.config.**Config**(*backend=None*, *global_prefix=''*, *owner=None*, ***cargs*)
Connection to SKA SDP configuration.

property backend
Get the backend database object.

property client_lease
Return the lease associated with the client.

It will be kept alive until the client gets closed.

close()
Close the client connection.

lease(*ttl=10*)
Generate a new lease.

Once entered can be associated with keys, which will be kept alive until the end of the lease. At that point a daemon thread will be started automatically to refresh the lease periodically (default seems to be TTL/4).

Parameters **ttl** – Time to live for lease

Returns lease object

txn(*max_retries=64*)
Create a *Transaction* for atomic configuration query/change.

As we do not use locks, transactions might have to be repeated in order to guarantee atomicity. Suggested usage is as follows:

```
for txn in config.txn():
    # Use txn to read+write configuration
    # [Possibly call txn.loop()]
```

As the *for* loop suggests, the code might get run multiple times even if not forced by calling *Transaction.loop()*. Any writes using the transaction will be discarded if the transaction fails, but the application must make sure that the loop body has no other observable side effects.

Parameters **max_retries** – Number of transaction retries before a RuntimeError gets raised.

class ska_sdp_config.config.**Transaction**(*config*, *txn*)
High-level configuration queries and updates to execute atomically.

create_deployment (*dpl: ska_sdp_config.entity.deployment.Deployment*)
Request a change to cluster configuration.

Parameters **dpl** – Deployment to add to database

create_processing_block (*pb: ska_sdp_config.entity.pb.ProcessingBlock*)
Add a new `ProcessingBlock` to the configuration.

Parameters **pb** – Processing block to create

create_processing_block_state (*pb_id: str*, *state: dict*)
Create processing block state.

Parameters

- **pb_id** – Processing block ID
- **state** – Processing block state to create

create_scheduling_block (*sb_id: str*, *state: dict*)
Create scheduling block.

Parameters

- **sb_id** – scheduling block ID
- **state** – scheduling block state

create_subarray (*subarray_id: str*, *state: dict*)
Create subarray.

Parameters

- **subarray_id** – subarray ID
- **state** – subarray state

delete_deployment (*dpl: ska_sdp_config.entity.deployment.Deployment*)
Undo a change to cluster configuration.

Parameters **dpl** – Deployment to remove

get_deployment (*deploy_id: str*) → ska_sdp_config.entity.deployment.Deployment
Retrieve details about a cluster configuration change.

Parameters **deploy_id** – Name of the deployment

Returns Deployment details

get_processing_block (*pb_id: str*) → ska_sdp_config.entity.pb.ProcessingBlock
Look up processing block data.

Parameters **pb_id** – Processing block ID to look up

Returns Processing block entity, or None if it doesn't exist

get_processing_block_owner (*pb_id: str*) → dict
Look up the current processing block owner.

Parameters **pb_id** – Processing block ID to look up

Returns Processing block owner data, or None if not claimed

get_processing_block_state (*pb_id: str*) → dict
Get the current processing block state.

Parameters **pb_id** – Processing block ID

Returns Processing block state, or None if not present

get_scheduling_block(*sb_id: str*) → dict
Get scheduling block.

Parameters **sb_id** – scheduling block ID

Returns scheduling block state

get_subarray(*subarray_id: str*) → dict
Get subarray.

Parameters **subarray_id** – subarray ID

Returns subarray state

is_processing_block_owner(*pb_id: str*) → bool
Check whether this client is owner of the processing block.

Parameters **pb_id** – Processing block ID to look up

Returns Whether processing block exists and is claimed

list_deployments(*prefix=''*)
List all current deployments.

Returns Deployment IDs

list_processing_blocks(*prefix=''*)
Query processing block IDs from the configuration.

Parameters **prefix** – If given, only search for processing block IDs with the given prefix

Returns Processing block ids, in lexicographical order

list_scheduling_blocks(*prefix=''*)
Query scheduling block IDs from the configuration.

Parameters **prefix** – if given, only search for scheduling block IDs with the given prefix

Returns scheduling block IDs, in lexicographical order

list_subarrays(*prefix=''*)
Query subarray IDs from the configuration.

Parameters **prefix** – if given, only search for subarray IDs with the given prefix

Returns subarray IDs, in lexicographical order

loop(*wait=False*, *timeout=None*)
Repeat transaction regardless of whether commit succeeds.

Parameters

- **wait** – If transaction succeeded, wait for any read values to change before repeating it.
- **timeout** – Maximum time to wait, in seconds

new_processing_block_id(*generator: str*)
Generate a new processing block ID that is not yet in use.

Parameters **generator** – Name of the generator

Returns Processing block ID

property raw
Return transaction object for accessing database directly.

take_processing_block(*pb_id: str*, *lease*)
Take ownership of the processing block.

Parameters **pb_id** – Processing block ID to take ownership of

Raises backend.ConfigCollision

update_processing_block(*pb: ska_sdp_config.entity.pb.ProcessingBlock*)
Update a `ProcessingBlock` in the configuration.

Parameters **pb** – Processing block to update

update_processing_block_state(*pb_id: str*, *state: dict*)
Update processing block state.

Parameters

- **pb_id** – Processing block ID
- **state** – Processing block state to update

update_scheduling_block(*sb_id: str*, *state: dict*)
Update scheduling block.

Parameters

- **sb_id** – scheduling block ID
- **state** – scheduling block state

update_subarray(*subarray_id: str*, *state: dict*)
Update subarray.

Parameters

- **subarray_id** – subarray ID
- **state** – subarray state

class ska_sdp_config.config.**TransactionFactory**(*config*, *txn*)
Helper object for making transactions.

ska_sdp_config.config.**dict_to_json**(*obj*)
Format a dictionary for writing it into the database.

Parameters **obj** – Dictionary object to format

Returns String representation

13.2 Entities

Processing block configuration entities.

class ska_sdp_config.entity.pb.**ProcessingBlock**(*id*, *sbi_id*, *workflow*, *parameters={}*, *dependencies=[]*, ***kwargs*)
Processing block entity.

Collects configuration information relating to a processing job for the SDP. This might be either real-time (supporting a running observation) or batch (to process data after the fact).

Actual execution of processing steps will be performed by a (parameterised) workflow interpreting processing block information.

property dependencies
Return dependencies on other processing blocks.

property id
Return the processing block ID.

property parameters
Return workflow-specific parameters.

property sbi_id
Return scheduling block instance ID, if associated with one.

to_dict()
Return data as dictionary.

property workflow
Return information identifying the workflow.

13.3 Backend

Backends for SKA SDP configuration DB.

CHAPTER

FOURTEEN

PROCESSING CONTROLLER

14.1 Introduction

The processing controller (PC) is the SDP service responsible for the controlling the execution of processing blocks (PBs).

Each scheduling block instance (SBI) that SDP is configured to execute contains a number of PBs, either real-time or batch. The real-time PBs run simultaneously for the duration of the SBI. Batch PBs run after the SBI is finished, and they may have dependencies on other PBs, both real-time and batch.

The SDP architecture requires the PC to use a model of the available resources to determine if a PB can be executed. This has not been implemented yet, so real-time PBs are always executed immediately, and batch processing ones when their dependencies are finished.

14.2 Processing block and its state

A PB and its state are located at the following paths in the configuration database:

```
/pb/[pb_id]
/pb/[pb_id]/state
```

The PB is created by the subarray Tango device when starting a SBI. Once it is created it does not change. The state is created by the PC when deploying the workflow, and it is subsequently updated by the PC and the workflow.

The entries in the PB state relevant to the PC are `status` and `resources_available`, for example:

```
{
    "status": "WAITING",
    "resources_available": false
}
```

`status` is a string indicating the status of the workflow. Possible values are:

- `STARTING`: set by the PC when it deploys the workflow, hereafter the workflow is responsible for setting `status`
- `WAITING`: workflow has started, but is waiting for resources to be available to execute its processing
- `RUNNING`: workflow is executing its processing
- `FINISHED`: workflow has finished its processing
- `FAILED`: set by the PC if it fails to deploy the workflow, or by the workflow in the case of a non-recoverable error

`resources_available` is a boolean set by the PC to inform the workflow whether it has the resources available to start its processing. Although the resource model is not implemented yet, this is used to control when batch PBs with dependencies start.

14.3 Behaviour

The behaviour of the PC is summarised as follows:

1. The PC uses HTTPS to retrieve the workflow definition file. This is a JSON file that specifies the mapping between the workflow ID and version, and a Docker container image. The workflow definition file is updated at regular intervals (the default is every 5 minutes).
2. If a PB is new, the PC will create the workflow deployment for it. A PB is deemed to be new if the PB state does not exist. The PC creates the state and sets `status` to `STARTING` and `resources_available` to `false`. If the workflow ID and version is not found in the definition file, the PC still creates the state, but sets `status` to `FAILED`.
3. If a PB's dependencies are all `FINISHED`, the PC sets `resources_available` to `true` to allow it to start executing. Real-time PBs do not have dependencies, so they start executing immediately.
4. The PC removes processing deployments (workflows and execution engines) not associated with any existing PB. This is used to clean up if a PB is deleted from the configuration DB. At present there is no mechanism for doing this (other than manually), but it might be used in future to abort a workflow execution.

CHAPTER

FIFTEEN

HELM DEPLOYER

CHAPTER

SIXTEEN

WORKFLOW DEVELOPMENT

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

- Clone the sdp-prototype repository from GitLab and create a new branch for your work.
- Create a directory for your workflow in `src/workflows`:

```
$ mkdir src/workflows/<my-workflow>
$ cd src/workflows/<my-workflow>
```

- Write the workflow script (`<my-workflow>.py`). See the existing workflows for examples of how to do this.
- Create a `Dockerfile` for building the workflow image, e.g.

```
FROM python:3.7

RUN pip install ska_sdp_config

WORKDIR /app
COPY <my-workflow>.py .
ENTRYPOINT ["python", "<my-workflow>.py"]
```

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

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

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

- Build the workflow image:

```
$ make build
```

- Push the image to the Nexus repository:

```
$ make push
```

- Add the workflow to the workflow definition file `src/workflows/workflows.json`.
- Commit the changes to your branch and push to GitLab.
- You can then test the workflow by starting SDP with the processing controller workflows URL pointing to your branch in GitLab:

```
$ 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
```

- Then create a processing block to run the workflow, either via the Tango interface, or by creating it directly in the config DB with `sdpcfg`.

16.1 Additional steps to build a custom execution engine

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>
```

CHAPTER

SEVENTEEN

VISIBILITY RECEIVE WORKFLOW

This is a simple C code for a visibility receiver capable of receiving UDP-based SPEAD (Streaming Protocol for Exchanging Astronomical Data) streams containing the item identifiers specified in the of the SDP-CSP ICD.

This code was originally written for the SKA Science Data Processor Integration Prototype and copied to this repository.

More information about SPEAD can be found here https://casper.ssl.berkeley.edu/astrobaki/images/9/93/SPEADsignedRelease.pdf

17.1 Dependencies

- CASACORE >= 2.0.0 : https://github.com/casacore/casacore
- OSKAR measurement set library : https://github.com/OxfordSKA/OSKAR
  - The OSKAR ms library can be installed as a standalone library from the `oskar/ms` folder of the repo. eg.: .. code-block:: guess

    git clone https://github.com/OxfordSKA/OSKAR.git mkdir OSKAR/oskar/ms/release cd OSAKR/oskar/ms/release cmake .. make make install

17.2 Build Instructions

To build the code on a local machine, ensure `make` and `CMake` are both installed and give the following commands from the current directory:

```
mkdir build
cd build
cmake ..
make
```

To run the unit tests from the build directory, either run

```
ctest
```

or run the unit test binary directly using:

```
./test/recv_test
```

17.3 Test Instructions

17.3.1 Starting the receiver

Run natively with:

```
./recv -d .
```

or with Docker:

```
docker run -t --rm \
    -p 41000:41000/udp \
    -v $(pwd)/output:/app/output \
    --env USER=orca \
    nexus.engageska-portugal.pt/sdp-prototype/vis-receive:latest
```

17.3.2 Staring the sender

```
python3 send.py
```

CHAPTER

EIGHTEEN

PSS RECEIVE WORKFLOW

This is a simple python code for a PSS (Pulsar Search Sub-element) receiver capable of receiving UDP-based SPEAD streams.

18.1 Dependencies

- Numpy>=1.16.2 : https://numpy.org/
- SPEAD2==2.0.2 : https://spead2.readthedocs.io/en/latest/

18.2 Description

There are two simple python programmes (receive and send). The sender is a dummy sender that is capable of loading a single-pulse search candidate file and sending the contents of that file, with some dummy metadata over UDP. The receiver listens for these UDP streams and terminates once they are received.

18.3 Running send and receive standalone

To demonstrate their functionality it is possible to run the send and receive codes natively.

18.3.1 Start the receiver

```
cd [sdp-prototype]/src/pss_receive/pss-receive
python receive.py
```

The console will show that the receiver is listening on port 9012

```
$ python receive.py
Listening on port 9021..
```

18.3.2 Start the sender

We are sending the contents of a single pulse search candidate file test.spccl. In a separate terminal...

```
cd [sdp-prototype]/src/pss_receive/pss-send
python send.py -f test.spccl
```

The console will show some details of the data that we've send.

```
$ python send.py -f test.spccl
INFO 2020-02-03 13:32:36,595 Start of stream
INFO 2020-02-03 13:32:36,595 Sending stream with id=EEK8V39g0JEP5Dmh, name=test.spccl
INFO 2020-02-03 13:32:36,595 Sending stream with id=EEK8V39g0JEP5Dmh, nbytes=5421
INFO 2020-02-03 13:32:36,595 Sending stream with id=EEK8V39g0JEP5Dmh, nlines=34
INFO 2020-02-03 13:32:36,595 End of stream
File test.spccl sent
```

Returning to the terminal in which we started the receiver we should see the data sent by send.py. The data has been saved to a file in the subdirectory 'output'.

18.4 Deploying receive as an sdp component

These instructions assume you have created the etcd cluster and deployed the sdp components. In other words, in the instructions for ''Running the SDP Prototype stand-alone'', you have done everything up to, but not including, the ''start a worflow" section.

Now we add a pss_receive processing block to the configuration database. We first connect to a console pod which allows us to interact with the configuration database.

```
$ kubectl exec -it deploy/sdp-prototype-console -- /bin/bash
$ sdpcfg process realtime:pss_receive:0.1.0
OK, pb_id = realtime-20200203-0000
```

We can watch the tasks that are being deployed in the sdp namespace by running..

```
$ kubectl get all -n sdp
NAME                                                    READY   STATUS              ␣
↪RESTARTS   AGE
pod/pss-receive-6p2dx                                   0/1     ContainerCreating   0 ␣
↪          45s
pod/realtime-20200203-0000-workflow-78fb74d48d-f6pgm    1/1     Running             0 ␣
↪          56s

NAME                    TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)     AGE
service/pss-receive     ClusterIP   10.96.224.5    <none>        9021/UDP    45s

NAME                                              READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/realtime-20200203-0000-workflow   1/1     1            1           56s

NAME                                                        DESIRED   CURRENT   ␣
↪READY   AGE
replicaset.apps/realtime-20200203-0000-workflow-78fb74d48d   1         1         1    ␣
↪    57s

NAME                    COMPLETIONS   DURATION   AGE
job.batch/pss-receive   0/1           46s        46s
```

. . . in which we see the workflow pod, created by the processing controller and the receive pod. Looking at the logs of the

processing controller we can see that processing block realtime-20200203-0000 has been deployed and is in a 'waiting' state as we haven't sent it any data yet.

```
$ kubectl logs sdp-prototype-processing-controller-[...]
processing_controller.py#105||('pss_receive', '0.1.0'): nexus.engageska-portugal.pt/
↪sdp-prototype/workflow-pss-receive:0.1.0
processing_controller.py#106||Batch workflows:
processing_controller.py#158||Current PBs: ['realtime-20200203-0000']
processing_controller.py#159||Current deployments: ['realtime-20200203-0000-pss-
↪receive', 'realtime-20200203-0000-workflow']
processing_controller.py#160||Current PBs with deployment: ['realtime-20200203-0000']
processing_controller.py#207||Waiting...
```

Looking at the output of the workflow pod in the sdp namespace we can see that the processing controller has claimed the processing block and deployed the pss-receive container.

```
$ kubectl logs realtime-20200203-0000-workflow-[...] -n sdp
INFO:pss_recv:Claimed processing block ProcessingBlock(pb_id='realtime-20200203-0000',
     sbi_id=None, workflow={'id': 'pss_receive', 'type': 'realtime', 'version': '0.1.0
↪'},
     parameters={}, scan_parameters={})
INFO:pss_recv:Deploying PSS Receive...
INFO:pss_recv:Done, now idling...
```

Finally, we can see the output of the receive pod, which shows the same console output as would be seen were we to be running the receive code standalone.

```
$ kubectl logs pss-receive-[...] -n sdp
Listening on port 9021..
```

18.4.1 Sending some data

```
$ cd [sdp-prototype]/src/pss_receive/pss-send
```

In this directory there is a K8s deployment manifest that will start a send job in the sdp namespace. To do this..

```
$ kubectl apply -f deploy-sender.yaml -n sdp
job.batch/sender created
```

Looking at the logs in the sdp namespace we see that..

- A sender pod was created and has completed
- The receiver pod shows as completed
- A sender job has been deployed under 'jobs'

```
$  kubectl get all -n sdp
NAME                                                READY   STATUS      RESTARTS
↪AGE
pod/pss-receive-q6dpd                               0/1     Completed   0
↪27s
pod/realtime-20200203-0000-workflow-78fb74d48d-662rb   1/1     Running     0
↪33s
pod/sender-rvxgh                                    0/1     Completed   0
↪11s

NAME                  TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
service/pss-receive   ClusterIP   10.96.242.53   <none>        9021/UDP   27s
```

(continues on next page)

(continued from previous page)

```
NAME                                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/realtime-20200203-0000-workflow     1/1     1            1           33s

NAME                                                           DESIRED   CURRENT
↪READY   AGE
replicaset.apps/realtime-20200203-0000-workflow-78fb74d48d     1         1         1
↪   33s

NAME                       COMPLETIONS   DURATION   AGE
job.batch/pss-receive      1/1           19s        27s
job.batch/sender           1/1           3s         11s
```

As before, looking at the logs for the sender and receiver pods, we can see the data that was sent/received.

18.4.2 Tidying up

Now we can stop the sender job and remove the processing block from the configuration

```
$ sdpcfg delete /pb/realtime-20200203-0000
$ kubectl delete job sender -n sdp
```

CHAPTER

NINETEEN

DEPLOYING THE SDP VIA TANGO

(More generalised documentation can be found at https://developer.skatelescope.org/projects/sdp-prototype/en/latest/running/running_standalone.html#accessing-tango

Connect to the TANGO interface

```
kubectl exec -it itango-tango-base-sdp-prototype -- /venv/bin/itango3
```

Create an interface to the SDP-subarray-1 TANGO device

```
d = DeviceProxy('mid_sdp/elt/master')
```

We can check the obsState of the sub-array which is currently IDLE.

```
d.obsState
```

We can then set the configuration of the Scheduling Block Instance by defining a JSON string containing information about the scan parameters and the workflow(s) to be deployed.

```
   config = '''
   {
  "id": "sbi-test-20200715-00000",
  "max_length": 600.0,
  "scan_types": [
    {
      "id": "science",
      "channels": [
        {"count": 372, "start": 0, "stride": 2, "freq_min": 0.35e9, "freq_max": 0.
↪358e9, "link_map": [[0,0], [200,1]]}
      ]
    }
  ],
  "processing_blocks": [
    {
      "id": "pb-test-20200715-00000",
      "workflow": {"type": "realtime", "id": "test_receive_addresses", "version": "0.
↪3.2"},
      "parameters": {}
    },
    {
      "id": "pb-test-20200715-00001",
      "workflow": {"type": "realtime", "id": "pss_receive", "version": "0.2.0"},
      "parameters": {}
    }
```

(continues on next page)

(continued from previous page)

```
  ]
}'''
```

In this case the workflows are test_receive_addresses and pss_receive. If we look at the processes that are running in the sdp namespace we can see that the two workflows have been deployed and the pss-receive workflow has deployed the pss-receive container (the test_receive_addresses workflow does not trigger any deployments).

```
[bshaw@dokimi ~]$ kubectl get all -n sdp
NAME                                                READY   STATUS    RESTARTS   AGE
pod/proc-pb-test-20200715-00000-workflow-dm48x      1/1     Running   0          2m
pod/proc-pb-test-20200715-00001-workflow-j8kbj      1/1     Running   0          2m
pod/pss-receive-sbwxz                               1/1     Running   0          1m

NAME                    TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
service/pss-receive     ClusterIP   10.96.84.108   <none>        9021/UDP   1m

NAME                                                COMPLETIONS   DURATION   AGE
job.batch/proc-pb-test-20200715-00000-workflow      0/1           42m        2m
job.batch/proc-pb-test-20200715-00001-workflow      0/1           42m        2m
job.batch/pss-receive
```

We can then set the scan type using the Configure() method.

```
d.Configure('{"scan_type": "science"}')
```

At this point our subarray has entered a READY state. We then set the scan running with the Scan() method.

```
d.Scan('{"id": 1}')
```

Now the subarray has entered a SCANNING state and data is being acquired. We can emulate the flow of data into the SDP from PSS by deployed the dummer sender by the same method as above. In a separate terminal,

```
cd /[...]/sdp-prototype/src/pss_receive/pss-send
kubectl apply -f deploy-sender.yaml
```

This will deploy a sender job (called sender) into the sdp namespace. This has sent some candidate data to the receiver, after which both the sender and receiver enter a "completed" state.

```
[bshaw@dokimi ~]$ kubectl get all -n sdp
NAME                                                READY   STATUS      RESTARTS   AGE
pod/proc-pb-test-20200715-00000-workflow-dm48x      1/1     Running     0          71m
pod/proc-pb-test-20200715-00001-workflow-j8kbj      1/1     Running     0          71m
pod/pss-receive-sbwxz                               0/1     Completed   0          69m
pod/sender-jrm6t                                    0/1     Completed   0          4m37s

NAME                    TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
service/pss-receive     ClusterIP   10.96.84.108   <none>        9021/UDP   69m

NAME                                                COMPLETIONS   DURATION   AGE
job.batch/proc-pb-test-20200715-00000-workflow      0/1           71m        71m
job.batch/proc-pb-test-20200715-00001-workflow      0/1           71m        71m
job.batch/pss-receive                               1/1           64m        69m
job.batch/sender
```

Now we can end the scan

```
d.EndScan()
```

and the subarray reverts to the READY state. We then clear the scan-type with.

```
d.Reset()
```

and the subarray returns to the IDLE state. The SDP resources can then be freed using

```
d.ReleaseResources()
```

which sets the workflows into a "completed" state.

19.1 Ongoing work

CHAPTER

TWENTY

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.

20.1 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",
  "buffer_img": "buff-pb-mvp01-20200523-00001-img"
}
```

20.2 Running the workflow

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

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

Once the sdp-prototype is running, start a iTango shell with:

```
kubectl exec -it itango-tango-base-sdp-prototype -- /venv/bin/itango3
```

First, obtain a handle to a subarray device with:

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

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": "0.2.0"},
      "parameters": {}
    },
    {
      "id": "pb-mvp01-20200523-00001",
      "workflow": {"type": "batch", "id": "batch_imaging", "version": "0.1.0"},
      "parameters": {
        "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",
        "buffer_img": "buff-pb-mvp01-20200523-00001-img"
      },
      "dependencies": [
        {"pb_id": "pb-mvp01-20200523-00000", "type": ["none"]}
      ]
    }
  ]
```

(continues on next page)

(continued from previous page)

```
}
'''
```

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

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

You can run the subarray commands as normal, but the batch processing does not start until you end the real-time processing with the `ReleaseResources` command:

```
d.ReleaseResources()
```

You can watch the pods and persistent volume clams (for the buffer reservations) being deployed with:

```
watch kubectl 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
...
```

20.3 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
```

CHAPTER

TWENTYONE

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.

21.1 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
}
```

21.2 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 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.
```

21.3 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
```

CHAPTER

TWENTYTWO

TEST WORKFLOWS

22.1 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.

22.2 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.

22.3 Test Receive Addresses Workflow

22.3.1 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.

22.3.2 Testing

Start the sdp prototype with (Helm 3 syntax):

```
helm install test sdp-prototype
```

Once all the pods are running, connect to the Tango interface using the following command:

```
kubectl exec -it itango-tango-base-test /venv/bin/itango3
```

Obtain a handle to the device with:

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

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]]
       }]
```

(continues on next page)

(continued from previous page)

```
    }
  ],
  "processing_blocks": [
    {
      "id": "pb-mvp01-20200318-0001",
      "workflow": {"type": "realtime", "id": "test_receive_addresses", "version": "0.
↪3.2"},
      "parameters": {}
    },
    {
      "id": "pb-mvp01-20200318-0002",
      "workflow": {"type": "realtime", "id": "test_realtime", "version": "0.2.0"},
      "parameters": {}
    }
  ]
} '''
```

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 deploy/test-sdp-prototype-console bash` and from there to see the full list run `sdpcfg ls -R /`

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

```
sdpcfg list values /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
        ],
        [
          2000,
          9000,
          1
        ]
      ]
```

(continues on next page)

(continued from previous page)


```
    },
    "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 `sdpcfg list values /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]]}}'
```


22.4 Test Dask Workflow

22.5 Test Daliuge Workflow

CHAPTER

TWENTYTHREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

R

S

T

U

W