developer.skatelescope.org Documentation

Release 1.2

Marco Bartolini

Nov 17, 2020

Contents

1 Installation **3**
- 1.1 Dependencies . . . 3
- 1.2 This package . . . 4

2 Receiver **5**
- 2.1 Configuration options . . . 5
- 2.2 Running . . . 6
- 2.3 Running Multiple Receivers . . . 6

3 Payload Consumers **7**
- 3.1 `mswriter` . . . 7
- 3.2 `plasma_writer` . . . 7
- 3.3 Adding Custom Consumers . . . 7

4 Sender **9**
- 4.1 Configuration options . . . 9

5 API documentation **11**
- 5.1 Packetisers . . . 11
- 5.2 Transmitters . . . 12
- 5.3 Payloads . . . 12
- 5.4 Receivers . . . 13
- 5.5 Others . . . 13

6 Running the Receive Workflow **15**
- 6.1 Tests and Quick Start . . . 15
- 6.2 Deploying the Receive Workflow in the SDP Prototype . . . 16
- 6.3 Deploying the Receive Workflow Behind a Proxy (PSI deployments) . . . 18

7 Running the Emulator **19**

Python Module Index **21**

Index **23**

This is an interface emulator for the Correlator Beamformer and SDP recieve workflow interface. It is an extensible and configurable package that has been designed to support multiple communication protocols and technolgies and provide a platform for testing consumers of CBF data payloads.

CHAPTER 1

Installation

1.1 Dependencies

Optionally, first create a virtual environment.

This package uses the OSKAR MS libraries for doing Measurement Set reading and writing. The OSKAR project contains numerous other functionality though which can be more expensive to build and install. To install **only** this part of the C++ library follow these instructions:

```
# Install dependencies and build tool
apt-get -y update
apt-get -y install cmake libblas-dev liblapack-dev casacore-dev

# Get a copy of OSKAR
git clone https://github.com/OxfordSKA/OSKAR.git
mkdir OSKAR/oskar/ms/release
cd OSKAR/oskar/ms/release

# Add -DCMAKE_INSTALL_PREFIX=$VIRTUAL_ENV if installing into a virtual
# environment
cmake -DCASACORE_LIB_DIR=/usr/lib/x86_64-linux-gnu ..

# Build and install
make -j4
make install

# Got back to the root directory where you started
cd ../../../..
```

Our package does not use the C++ OSKAR MS library directly but its python bindings. However, the OSKAR python bindings package does not support out of the box to install **only** the MS part of the bindings. Thus, the following steps will be required:

```
# Get a copy of this package
git clone https://gitlab.com/ska-telescope/cbf-sdp-emulator

# Use this package's copy of setup.py to build OSKAR's python bindings
cd OSKAR/python
mv ../../cbf-sdp-emulator/3rdParty/setup_oskar_ms.py ./setup.py
python3 ./setup.py build
python3 ./setup.py install
cd ../../
```

1.2 This package

This is a standard setuptools-based program so the usual installation methods should work. The only caveat is that some of the dependencies are found in the EngageSKA's Nexus server instead of PyPI, so we need to point `pip` to them:

```
# Go into the top-level directory of this repository
cd cbf-sdp-emulator

# Using "pip" will automatically install any dependencies from PyPI
pip install --extra-index-url=https://nexus.engageska-portugal.pt/repository/pypi/
↪simple .

# Use pip in editable mode if you are actively changing the code
pip install --extra-index-url=https://nexus.engageska-portugal.pt/repository/pypi/
↪simple -e .
```

This package contains code to interact with a Plasma store. This is declared as an extra `plasma` dependency, so if you are planning to use this support you will need to install the extra dependency:

```
pip install --extra-index-url=https://nexus.engageska-portugal.pt/repository/pypi/
↪simple .[plasma]
```

CHAPTER 2

Receiver

An `emu-recv` program should be available after installing the package. This program receives packets. It is an extensible package that is built to be agnostic to the actual mode of reception.

2.1 Configuration options

Configuration options can be given through a configuration file or through the command-line. See `emu-recv -h` for details. An example configuration file is given in 'example.conf'.

The following configuration categories/names are supported for the SPEAD2 receivers - other receivers can be added that are not SPEAD2 compliant. But currently as of this initial version we are only supplying the SPEAD2 UDP receiver. The API for adding more receivers in this framework is supplied:

- `reception`: these are configuration options applied when reading the input Measurement Set.
- `bind_hostname`: The IP address or hostname of the interface to which to bind for reception.
- `receiver_port_start`: The initial port number to which to bind.
- `num_ports`: [optional] The number of ports to which to bind. This can also be calculated from the 'channels_per_stream' option and the number of channels in the data-model.
- `consumer`: The consumer class to attach to the receiver, defaults to `mswriter`. If this a simple name, then it is assumed to be a module under the `cbf_sdp.consumers` package which inside must contain a `consumer` class. User-provided code can also be specified by giving the full path to a class (e.g., `my.python.package.module.classname`). For more details on consumers see *Payload Consumers*.
- `datamodel`: This is important. We have no interface with the telescope model (TM) and therefore have to obtain all the observation metadata from somewhere. We have decided to use the Measurement Set as the basis for this. So you should supply a measurement set that contains the same metadata as that which is being sent. it does not have to be the same file - but the output measurement set parameters and the UVW will be taken from this file. We open and close this file quickly so there should be no issue with multiple open files. Providing the receiver is started before the sender.
- `transmission`: these are options that generally apply to the transmission method and are mostly ignored by the receiver except:

- channels_per_stream: The number of channels for which data will be sent in a single stream. This is used in the case where multiple ports are required with multiple channels per port. You dont actually need this - but without it you have to set num_ports appropriately

2.2 Running

The application is really simple to run. It is installed as an entrypoint when you install the package and will run as simply as: emu_recv -c <configuration_file> Or you can specify the various options on the command line. A typical configuratio file is supplied and looks like this:

```
[transmission]

method = spead2_transmitters
target_host = 127.0.0.1
target_port_start = 41000
channels_per_stream = 1
rate = 247000

[reader]

[payload]

method = icd

[reception]

method = spead2_receivers
receiver_port_start = 41000
consumer = spead2_mswriter
datamodel = tests/data/gleam-model.ms
outputfilename = tests/data/recv-vis.ms
```

Note by default we use a spead2 transmitter and receiver and a icd payload. We have designed this package to be extensible and if you want to add different transmitters and receivers you should be able to

In practical terms it makes sense to start the receiver(s) before the transmitter so they are waiting for data. But you do not have to - the protocols and consumers are flexible enough to be started when the data-stream is already running.

2.3 Running Multiple Receivers

In many places we have made design decisions that are common to the SKA-SA systems. In the case of multiple consumers we are simply using UDP multicast. THis means that multiple consumers can access the same transmitted stream if they bind to the same multicast IP address and ports. We have examples of this operations in the example configurations.

This multiple-receiver operation is required when displays and monitoring is required to sample the data in transit - before it becomes a measurement set.

CHAPTER 3

Payload Consumers

Upon the reception and decoding of a payload, a receiver passes it to a *consumer*. Consumers are a simple mechanism for decoupling data reception from any further data processing.

The `cbf-sdp-emulator` package currently comes with 2 built-in consumers, but arbitrary consumers can be used as well. The mechanism used to choose a consumer can be found *here* (see the *reception.consumer* option).

3.1 `mswriter`

The `mswriter` consumer, as derived from its name, writes incoming payloads into a Measurement Set. If payloads are missing the resulting Measurement Set will still have the missing rows, but with invalid data.

3.2 `plasma_writer`

The `plasma_writer` consumer puts the incoming payloads into a shared plasma store using the sdp-dal-prototype. The `sdp-dal-prototype` implements an RPC-like API; from its standpoint our `plasma_writer` is a *Caller*, and the payload is written into plasma representing a remote method invocation. In order to demonstrate the full cycle of writing and reading data into/from plasma we also provide a corresponding *Processor* under `cbf_sdp.plasma_processor.SimpleProcessor`, which corresponds to the callee. Upon invocation, this processor takes incoming payloads and writes them into a Measurement Set, similarly to how `mswrite` does.

3.3 Adding Custom Consumers

Third-party consumers are also supported, which users can provide within their own code bases. Consumers are implemented as classes with the following signatures:

- An `__init__(config, tm)` method for initialization. The `config` parameter contains the full receiver configuration dictionary, as loaded from its command-line and configuration file. The `tm` is an instance of *`cbf_sdp.utils.FakeTM`* containing most metadata about the observation.

- An `async def consume(self, payload)` method for payload consumption. The `payload` parameter is an instance of *cbf_sdp.icd.Payload*. Note that this is a coroutine, so potentially long-running tasks should be spawned off using executors to avoid hanging the event loop.

CHAPTER 4

Sender

An `emu-send` program should be available after installing the package. This program takes a Measurement Set and transmits it over the network using the preferred transmission method.

4.1 Configuration options

Configuration options can be given through a configuration file or through the command-line. See `emu-send -h` for details.

The following configuration categories/names are supported:

- `reader`: these are configuration options applied when reading the input Measurement Set.
- `start_chan`: the first channel for which data is read. Channels before this one are skipped. If `start_chan` is bigger than the actual number of channels in the input MS an error is raised.
- `num_chan`: number of channels for which data is read. If `num_chan` + `start_chan` are bigger than the actual number of channels in the input MS then `num_chan` is adjusted.
- `num_repeats`: number of times a single set of visibilities should be sent after being read, defaults to `1`. Bigger values will send the same data over and over, which is less realistic but imposes less stress on the file-system.
- `transmission`: these are options that apply to the transmission method.
- `method`: the transmission method to use, defaults to `spead2`.
- `target_host`: the host where data will be sent to.
- `target_port_start`: the first port where data will be sent to.
- `channels_per_stream`: number of channels for which data will be sent in a single stream.
- `max_packet_size`: the maximum size of packets to build, used by `spead2`.
- `rate`: the maximum send data rate, in bytes/s. Used by `spead2`, defaults to 1 GB/s.

CHAPTER 5

API documentation

This section describes requirements and guidelines.

5.1 Packetisers

We begin with the packetiser we have written as a default `emulator` this is a pretty simple package that uses the transmitter and payload classes defined in the configuration to send data.

At the moment we have an assumption that the ICD payload is being used. but minor changes to the packetise method would remove that requirement. Very minimal work is needed to replicate this with another payload.

The actual transmission protocol is abstracted into the `transmitters` and this is currently defaulting to SPEAD2 and UDP. But as this is almost completely abstracted should be easy to change.

Default Emulator

Reads data from a MS and creates SPEAD2 packets.

This package is designed to be easily extensible to different payloads and protocols. This module is currently installed as emu_send when the package is installed.

`cbf_sdp.packetiser.`**`packetise`**(*config*, *ms*, *loop=None*)

Reads data off a Measurement Set and transmits it using the transmitter specified in the configuration.

Uses the vis_reader get data from the measurement set then gives it to the transmitter for packaging and transmission. This code is transmission protocol agnostic.

Parameters

- **`config`** – The configuration
- **`ms`** – The measurement set

5.2 Transmitters

The transmitters are envisaged to be at least as diverse as UDP, IBV and perhaps ROCE we have only implemented the UDP transmitter. But extensions should be trivial

spead2_transmitters

Class that manages transmission of a SPEAD2 HEAP via UDP and of a content defined by the payload class

class cbf_sdp.transmitters.spead2_transmitters.**Spead2SenderPayload**(*num_baselines=None*, *num_channels=None*)

SPEAD2 payload following the CSP-SDP interface document

class cbf_sdp.transmitters.spead2_transmitters.**transmitter**(*config*, *num_baselines*, *num_chan*, *loop*)

SPEAD2 transmitter

This class uses the spead2 library to transmit visibilities over multiple spead2 streams. Each visiblity set given to this class' *send* method is broken down by channel range (depending on the configuration parameters), and each channel range is sent through a different stream.

close()

Sends the end-of-stream message

send(*ts*, *ts_fraction*, *vis*)

Send a visibility set through all SPEAD2 streams

Parameters

- **ts** – the integer part of the visibilities' timestamp
- **ts_fraction** – the fractional part of the visibilities' timestamp
- **vis** – the visibilities

5.3 Payloads

class cbf_sdp.icd.**Payload**

A payload as specified by the ICD

channel_count

The number of channels contained in this payload

channel_id

The ID of the first channel of this payload

correlated_data_fraction

The fraction of data on this payload that was correlated

hardware_id

The ID of the hardware source of this payload

mjd_time

The timestamp of the payload in MJD seconds

phase_bin_count

The number of phase bins of this payload

phase_bin_id

The ID of the first phase bin of this payload

polarisation_id
The ID of the polarisation of this payload

scan_id
The ID of the scan of this payload

time_centroid_indices
The time centroids for each visibility of this payload

timestamp_count
The timestamp of the visibilities, as (integer) seconds since UNIX epoch

timestamp_fraction
The fractional timestamp of the visibilities, as an integer with units of 1/2**32 seconds

unix_time
The timestamp as fractional seconds since UNIX epoch

visibilities
The correlator visibilities of this payload

5.4 Receivers

UDP Protocol Multi-stream SPEAD2 receiver

class cbf_sdp.receivers.spead2_receivers.**Spead2ReceiverPayload**
A Payload that updates itself from data coming from spead2 heaps

class cbf_sdp.receivers.spead2_receivers.**receiver**(*config*, *tm*, *loop*)
SPEAD2 receiver

This class uses the spead2 library to receive a multiple number of streams, each using a single UDP reader. As heaps are received they are given to a single consumer.

run()
Receive all heaps, passing them to the consumer

5.5 Others

class cbf_sdp.utils.**FakeTM**(*ms*)
TelescopeManager-like class that reads its model information from a Measurement Set.

freq_inc_hz
The frequency increment between channels, in Hz

freq_start_hz
The frequency of the first channel, in Hz

get_freq_inc_hz()
The frequency increment between channels, in Hz

get_freq_start_hz()
The frequency of the first channel, in Hz

get_is_autocorrelated()
Whether the current observation is used autocorrelation or not

get_matching_data(*current_mjd_utc*) → cbf_sdp.utils.DataObject
Like get_nearest_data, but if no exact match is found an *ValueError* exception is raised.

get_nearest_data(*time*) → cbf_sdp.utils.DataObject
Returns the (meta)data associated with correlator dumps happening at a given point in time. If no exact match is found the nearest is returned.

get_num_baselines()
The number of baselines used by the current observation

get_num_channels()
The number of channels of the current observation

get_num_pols()
The number of polarisations used by the current observation

get_num_stations()
The number of stations used by the current observation

get_phase_centre_radec_rad()
Return the RA/DEC phase centre in radians

is_autocorrelated
Whether the current observation is used autocorrelation or not

num_baselines
The number of baselines used by the current observation

num_channels
The number of channels of the current observation

num_pols
The number of polarisations used by the current observation

num_stations
The number of stations used by the current observation

phase_centre_radec_rad
Return the RA/DEC phase centre in radians

CHAPTER 6

Running the Receive Workflow

There are many ways to deploy this workflow, standalone on a local machine for testing. Distributed across a local cluster, or installed as a Kubernetes chart. This interface simlualtor sits across 2 domains in the SKA. The emulator (sender) is a synthesiser of the Correlator Beamformer (CBF) which is a device within the Central Signal Processor (CSP) domain. The receiver is an example of a Science Data Processor workflow and as such resides in the SDP regime.

We have developed a number of mechanisms by which these two elements can be deployed. But they essentailly fall into tow simple groups. A kubernetes deployment - be it in a general Kubernetes environment or more specifically the SKAMPI prototype of the SKA. This section of the documentation deals with a few example deployments.

6.1 Tests and Quick Start

Ok so you don't want to read all the documentation, or just want to get something running straight away. Open in the quickstart directory and you will find some simple configuration files.

There are quickstart examples for the following situations:

1) A simple send and receive pair for a small number of channels on a single stream. The basic and simplest scheme - this will not expected to scale beyond a few hundred channels.

2) A simple send and receive pair for a larger number of channels using multiple streams but a single output file. This employs a multi-threaded asynchronous receive and should scale. Although the performance may be limited by disk performance - both sending and receiving.

Each experiment is in its own directory - example data sets are included in the tests/data directory. You should not need to install anything other than this package to get them to work.

6.1.1 Running the examples

The example directories include a working configuration and run script ('run.sh'). Just executing the run script will run a receiver in the background and a sender in the foreground. It will transfer the visibility data weights and flags and transfer the meta-data from the mock-TM interface.

6.2 Deploying the Receive Workflow in the SDP Prototype

6.2.1 Setting up the Prototype

In these instructions we are assuming you have a deployed Kubernetes environment. Either minikube or the integration environment and the etcd and sdp-prototype charts are installed:

```
> helm list
NAME            NAMESPACE       REVISION        UPDATED                                 ␣
↪  STATUS          CHART                                   APP VERSION
etcd            default         1               2020-10-27 10:20:42.192392 +1100 AEDT␣
↪  deployed        etcd-operator-0.11.0                    0.9.4
sdp-prototype   default         1               2020-10-27 10:20:58.656668 +1100 AEDT␣
↪  deployed        sdp-prototype-0.4.0                     1.0
```

Then ensure that all your pods are running:

```
> kubectl get pods
NAME                                                READY   STATUS      RESTARTS   AGE
databaseds-tango-base-sdp-prototype-0               1/1     Running     1          ␣
↪162m
etcd-etcd-operator-etcd-operator-796f6fd5bb-52qsf   1/1     Running     0          ␣
↪163m
itango-tango-base-sdp-prototype                     1/1     Running     2          ␣
↪162m
sdp-prototype-console-86dc9bb7d6-4fxfc              1/1     Running     0          ␣
↪162m
sdp-prototype-etcd-vnkfzzwzgk                       1/1     Running     0          ␣
↪162m
sdp-prototype-helmdeploy-5cbdbd9d48-68rs2           1/1     Running     0          ␣
↪162m
sdp-prototype-lmc-cb799bbdf-5p5dg                   3/3     Running     0          ␣
↪162m
sdp-prototype-proccontrol-854779ff7d-m5s6w          1/1     Running     0          ␣
↪162m
tangodb-tango-base-sdp-prototype-0                  1/1     Running     0          ␣
↪162m
```

The pods may take a while to start running.

6.2.2 Configuring the Workflow

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

The configuration of the receive workflow is managed via adding to the configuration of the processing block. The processing block can be created using the *sdpcfg* utility or via the iTango interface. In this example we will assume *sdpcfg* is being used:

```
> sdpcfg process realtime:test_new_receive:0.1.4
```

This will start up a default deployment. Without arguments this is a test deployment. If will launch a number of containers and both a sender and receiver in the same pod. We typically use this for testing purposes. The behaviour and the chart deployed can be altered by adding a JSON blob to the command line, for example:

```
>sdpcfg process realtime:test_new_receive:0.1.4 "{ transmit.model : false, reception.
↪ring_heaps : 133 }"
```

In the above example you can see there are two key value pairs in the JSON blob. The first `transmit.model : false` tells the receive workflow not to start a sender/emulator container. In the future we may make this the default state. The second `reception.ring_heaps : 133` is an example of a configuration setting for the receive workflow. All the options supported by the receiver are supported by the chart deployment. The defaults set by the workflow currently are:

```
'model.pull' : 'true',
'model.url' : 'https://gitlab.com/ska-telescope/cbf-sdp-emulator/-/raw/master/data/
↪sim-vis.ms.tar.gz',
'model.name' : 'sim-vis.ms',
'transmit.model' : 'true',
'reception.outputfilename' : 'output.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'
```

For more information on the configuration of the receivers see *Receiver*. The will also be some default configurations for the chosen `consumer` in *Payload Consumers*.

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 neccesary to provide a scheme 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.

Once *sdpcfg* has be run 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)

6.2.3 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:

```
>sdpcfg process realtime:test_new_receive:0.1.4 "{ 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.

6.3 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 kbernetes environment itself is behind a firewall. The SDP prototype deployment can be thought of as charts thant 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 in an integration environment which is managed by CSIRO and behind a web-proxy. When the sdp-prototype is deployed all the elements of the prototype need to be informed of the proxy

6.3.1 Configuring Workflow to Use The Proxy

Firstly the sdp prototype needs to be deployed with a proxy setting exposed. This is an install line which will expose the CSIRO proxy to the helm charts of the sdp prototype:

```
helm install sdp-prototype sdp-prototype  --set proxy.server=delphinus.atnf.csiro.
↪au:8888 --set proxy.noproxy='{}'
```

This will ensure the prototype itself is launched with the correct proxy settings.

But as you would expect this does not neccesarily pass the proxy settings on to workflows. Ih the case of the receive workflow.

This is the equivalent `sdpcfg` line with the proxy information:

```
> sdpcfg process realtime:test_new_receive:0.1.4 "{proxy.server : delphinus.atnf.
↪csiro.au:8888 ,
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 ,
proxy.use : true }"
```

This command line would launch the receive workflow on the PSI, behind a proxy, configured to push the results to a Google Cloud Services bucket.

CHAPTER 7

Running the Emulator

Now we can configure the receive workflow in a number of different environments - we should consider how to deploy the emulator.

The emulator can be ran simply as a standalone application and configured on the command line. This is how the straightforward deployments work. However we have also built a tango device to control the application and it is documented here: CBF-SDP Emulator TANGO Devices

These is a chart included in the tango-device repository *https://gitlab.com/ska-telescope/cbf-sdp-emulator-tango-device.git* and that is deployed in a similar way to the kubernetes deployment of the receive workflow:

```
> cd charts
> helm install emulator cbf-sdp-emulator-tango-device  --set proxy.use=true --set␣
↪proxy.server=delphinus.atnf.csiro.au:8888
```

Then the device can be configured and controlled via an LMC container as described in CBF-SDP Emulator TANGO Devices.

Python Module Index

.

../cbf_sdp/transmitters/spead2_transmitters.py, 8

C

cbf_sdp.packetiser, 11

cbf_sdp.receivers.spead2_receivers, 13

cbf_sdp.transmitters.spead2_transmitters, 12

Index

Symbols

../cbf_sdp/transmitters/spead2_transmitters.py (*module*), 8

C

cbf_sdp.packetiser (*module*), 11
cbf_sdp.receivers.spead2_receivers (*module*), 13
cbf_sdp.transmitters.spead2_transmitters (*module*), 12
channel_count (*cbf_sdp.icd.Payload attribute*), 12
channel_id (*cbf_sdp.icd.Payload attribute*), 12
close() (*cbf_sdp.transmitters.spead2_transmitters.transmitter method*), 12
correlated_data_fraction (*cbf_sdp.icd.Payload attribute*), 12

F

FakeTM (*class in cbf_sdp.utils*), 13
freq_inc_hz (*cbf_sdp.utils.FakeTM attribute*), 13
freq_start_hz (*cbf_sdp.utils.FakeTM attribute*), 13

G

get_freq_inc_hz() (*cbf_sdp.utils.FakeTM method*), 13
get_freq_start_hz() (*cbf_sdp.utils.FakeTM method*), 13
get_is_autocorrelated() (*cbf_sdp.utils.FakeTM method*), 13
get_matching_data() (*cbf_sdp.utils.FakeTM method*), 13
get_nearest_data() (*cbf_sdp.utils.FakeTM method*), 14
get_num_baselines() (*cbf_sdp.utils.FakeTM method*), 14
get_num_channels() (*cbf_sdp.utils.FakeTM method*), 14
get_num_pols() (*cbf_sdp.utils.FakeTM method*), 14
get_num_stations() (*cbf_sdp.utils.FakeTM method*), 14
get_phase_centre_radec_rad() (*cbf_sdp.utils.FakeTM method*), 14

H

hardware_id (*cbf_sdp.icd.Payload attribute*), 12

I

is_autocorrelated (*cbf_sdp.utils.FakeTM attribute*), 14

M

mjd_time (*cbf_sdp.icd.Payload attribute*), 12

N

num_baselines (*cbf_sdp.utils.FakeTM attribute*), 14
num_channels (*cbf_sdp.utils.FakeTM attribute*), 14
num_pols (*cbf_sdp.utils.FakeTM attribute*), 14
num_stations (*cbf_sdp.utils.FakeTM attribute*), 14

P

packetise() (*in module cbf_sdp.packetiser*), 11
Payload (*class in cbf_sdp.icd*), 12
phase_bin_count (*cbf_sdp.icd.Payload attribute*), 12
phase_bin_id (*cbf_sdp.icd.Payload attribute*), 12
phase_centre_radec_rad (*cbf_sdp.utils.FakeTM attribute*), 14
polarisation_id (*cbf_sdp.icd.Payload attribute*), 12

R

receiver (*class in cbf_sdp.receivers.spead2_receivers*), 13
run() (*cbf_sdp.receivers.spead2_receivers.receiver method*), 13

S

scan_id (*cbf_sdp.icd.Payload attribute*), 13

send() (*cbf_sdp.transmitters.spead2_transmitters.transmitter method*), 12
Spead2ReceiverPayload (*class in cbf_sdp.receivers.spead2_receivers*), 13
Spead2SenderPayload (*class in cbf_sdp.transmitters.spead2_transmitters*), 12

T

time_centroid_indices (*cbf_sdp.icd.Payload attribute*), 13
timestamp_count (*cbf_sdp.icd.Payload attribute*), 13
timestamp_fraction (*cbf_sdp.icd.Payload attribute*), 13
transmitter (*class in cbf_sdp.transmitters.spead2_transmitters*), 12

U

unix_time (*cbf_sdp.icd.Payload attribute*), 13

V

visibilities (*cbf_sdp.icd.Payload attribute*), 13