ska-sdp-lmc-queue-connector Documentation

Release 4.0.4

Julian Carrivick, Callan Gray and Matteo Di Carlo

Apr 11, 2024

QUEUE CONNECTOR DEVICE

This project implements a Tango device intended to provide services to exchange data between various components internal and external to the Science Data Processor (SDP). Initially, this will essentially only be exchanging data between Tango devices external to the SDP and Kafka topics only available internally to SDP components, though it is reasonable to expect this may expand to other sources and sinks in future.

The LMC Queue Connector Device will sit in the Science Data Processor and interact with various other components both within and outside that boundary. These components will directly interact by reading from or subscribing to attributes on the Queue Connector device. They will also indirectly interact with the Data Exchange device by publishing data as a Tango attribute (if it's a Tango device) or to a Kafka topic, which the Queue Connector device will subscribe to.

In the diagram below, we describe expected flow of data between each of these components, with the Queue Connector device essentially acting as a middle-man between Tango and Kafka.

One example workflow which this is expected to be used for is to calculate and publish pointing offsets to the telescope. Our Queue Connector Device will subscribe to the telescope sub-array device to obtain pointing data and write it to a Kafka topic, this will then be consumed by a processor in the realtime receive pipeline which will perform some calculations using the pointing data and the visibilities it is receiving. It will then write these offsets to another Kafka topic, which the the Queue Connector Device will be subscribed to, which is then exposed using a Tango attribute. The telescope sub-array will subscribe to this attribute to obtain the pointing offsets, completing the feedback loop.

CHAPTER

ONE

ARCHITECTURE

1.1 Exchange Model

The LMC Queue Connector Device processes data in the form of one or more configurable exchanges/streams that continuously process when the device is in the `ON` state.

The simplified class diagram for this can be expressed by the following:

1.2 Data Flow

An *Exchange* is fundamentally an object that asynchronously pulls data from one or more sources streams (*DataSource*), through a pipe operator (*DataPipe*), and then asynchronously pushes to a sink stream (*DataSink*). The asynchronous interface between sources and sinks allows for a single device to indefinately process multiple exchanges/streams, without threads being blocked waiting for I/O, and until instructed to stop.

Data into an exchange each use the same single dtype and input shape, and pipe operators may change the output data shape.

1.2.1 Example

Two common configurations for exchanges are:

- Publishing a Tango subscription to a Kafka topic
- Publishing a Kafka topic as a Tango attribute

In this example there are 2 runtime implementations for each of *DataSource* and *DataSink*. Data is sourced from either another Tango device attribute or a Kafka topic. This data is then be propagated to either a local attribute on the Queue Connector device or to a Kafka topic. It is reasonable to expect the need for other implementations too (e.g. An in-memory Source and Sink for unit testing) and this is accomplished by implementing the *DataSource* and *DataSink* interfaces.

CHAPTER

TWO

TANGO DEVICE

2.1 Properties

Property	Type	Default Value	Example Value
`exchanges_json`	String	""	`'{"exchanges": []}'`
`exchanges_config_path`	String	""	`'/component/lmc-queueconnector-01/owner'`

2.2 Commands

Command	Args	Return Type
`IsMonitoringDB()`	Void	Bool
`Configure()`	String	Void
`Reset()`	Void	Void

Note: Additional commands `Standby()`, `Start()`, `Stop()`, `Abort()` are intended for testing purposes and may be disabled or removed in future releases.

2.3 State Model

The present implementation utilizes the following native Tango operational states that can be read using the `state` attribute.

- **INIT**: the device is currently initialising.
- **FAULT**: the device has experienced an unexpected error from which it needs to be reset.
- **STANDBY**: the device is not configured.
- **OFF**: the device is configured, but not currently in use.
- **OPEN**: the device is opening stream connections.
- **ON**: the device is streaming data.
- **CLOSE**: the device is closing stream connections.

Note: Transition arrows are triggered by a corresponding tango command, with the exception of underlined transitions, which are triggered automatically.

2.4 Configuring

The Queue Connector Tango Device only streams data when it has been configured at runtime. This can be performed via multiple approaches.

2.4.1 Configure by Property

To configure via properties, set either the `exchanges_json` and *exchanges_config_path* properties in the tango database and the device will configure on initialization.

Note: If both `exchanges_json` and *exchanges_config_path* properties are provided, then *exchanges_config_path* will take priority.

Configuration Database Path

To configure via SDP Config, set the *exchanges_config_path* property to a SDP Config path (including leading slash). The device will fetch the JSON value at the given path parsed as a *QueueConnectorDescriptor*.

JSON

To configure via JSON, set the `exchanges_json` property to JSON that will be parsed as a *QueueConnectorDescriptor*.

2.4.2 Configure by Command

Configuration may be performed post initialization using the `Configure()` command.

JSON

Pass a raw JSON string of the `QueueConnectorDescriptor` to the `Configure()` when in the **STANDBY** state. Configuring the device successfully will transition the device to the **OFF** state.

Configuration Database Path

Pass a configuration database path string to the `Configure()` command.

Note: The command will check for a leading '/' as a heuristic for detecting a config_path to configure from, otherwise attempting to parse the string as JSON.

Note: Configure by command is deprecated and estimated for remove in version 4.0.0

CHAPTER

THREE

CONFIGURATION

The LMC Queue Connector is configured using a *QueueConnectorDescriptor* containing one or more *ExchangeDescriptor* instances in JSON, where each exchange represents independent streams for processing.

The tango device can be configured after initializing via multiple approaches, see *Tango Device*.

3.1 Queue Connector Descriptor

A typical *QueueConnectorDescriptor* with a single *ExchangeDescriptor* consists of:

```
{
    "exchanges": [
        {
            "dtype": "<data type>",
            "shape": [ ... ],
            "source": { "type": "<source type>" } | [ ... ],
            "pipe": { "type": "<pipe type>" },
            "sink": { "type": "<sink type>" },
        }
    ]
}
```

Where:

- **dtype**: Numpy dtype or Python type to deserialize to.
- **shape**: Optional data dimensions of the source if data is arraylike.
- **source**: Exchange input description for reading data.
- **pipe**: Optional pipe function for stream operations.
- **sink**: Exchange output description for writing data.

A compatible combination of dtype, shape and encoding must be chosen at runtime to initialize the exchange instance ready for streaming.

See below for summaries of each, *Configuration Schema* for schema and *API* for further details on available sources, sinks and pipes.

3.2 Data Dtype

The data type provided to an exchange may be either a python native built-in type (*bool*, *int*, *float*, *str*, *object*, *bytes*) or any name of a supported numpy generic class as listed below:

Descriptor Literal	DType	Tango ArgTy	Notes
`"bytes"`	`tuple [str, dtype('S')]`	DevEn	Shape must be scalar. Data is not validated to ensure it matches the encoding.
`"object_"`	`numpy. dtype('O')`	N/A	Shape must be scalar. This type is reserved largely for container types such *list*, *tuple* and *dict*, but is compatible with all dtypes. For Tango, only compatible with sink *TangoObjectScatterAttributeSink*.
`"str_"`	`numpy. dtype('<U')`	DevSt	Uses UTF-8 encoding.
`"bool"`	`numpy. dtype('bool')`	DevBo	
`"uint8"`	`numpy. dtype('uint8'`	DevUC	
`"uint16"`	`numpy. dtype('uint16`	DevUS	
`"uint32"`	`numpy. dtype('uint32`	DevUL	
`"uint64"`	`numpy. dtype('uint64`	DevUL	
`"int16"`	`numpy. dtype('int16'`	DevSh	
`"int32"`	`numpy. dtype('int32'`	DevLo	
`"int64"` or `"int"`	`numpy. dtype('int64'`	DevLo	
`"float32"`	`numpy. dtype('float3`	DevFl	
`"float64"` or `"float"`	`numpy. dtype('float6`	DevDo	
`"datetime64"` or `datetime64[**ti precision**]`	`numpy. dtype('<M8[** precision**]'`	None	
structured datatype (`List[tuple(str, Scalar)]`)	`numpy. dtype('[. ..]')`	*Any*	For Tango, only compatible with sink `TangoPointingSubscriptionSource`.

Note: Tango *DevEnum* and *DevPipeBlob* are currently not supported.

In addition, the dtype may be specified as a numpy structured datatype, though this is limited to the *In-Memory* and *Kafka* data sources and sinks. For example, a list of Alt/Az coordinates might have descriptor dtype of `[["alt", "float64"], ["az", "float64"]]`.

Note: When using a structured datatype and serialising to Kafka with `"carray"` encoding, a limited subset of field

data types is allowed. Notably, `datetime64` types are not supported (See this issue). There is no such limitation with `"npy"` encoding.

3.3 Data Shape

The optional exchange *shape* describes the maximum dimensions size for an array of values. An empty list (default) indicates a scalar, and up to a single negative shape value may be used to indicate a dynamically sized dimension.

Using a Tango source or sink will limit the number of dimensions to 2 as Tango only supports the `SCALAR` format, the `SPECTRUM` format for 1D arrays, and the `IMAGE` format or 2D arrays. Specialized sink *TangoArrayScatterAttributeSink* can work around this limitation by using array flattening and generating a separate attribute of the original shape.

Note: Tango attributes require a maximum dimension size, thus Tango sinks currently do not support negative values in the data shape.

Note: When working with multi-dimensional `str` data, the Tango DeviceProxy will return data as a `list` rather than a `numpy.ndarray`.

3.4 Data Source(s)

A *DataSource* is an extensible component of the LMC Queue Connector representing a location the data can be asynchronously pulled from. The data source will convert incoming data to the dtype specified in the exchange descriptor.

Multiple data sources are supported on a single exchange and are merged into a single stream.

3.5 Data Pipe

An optional *DataPipe* is an extensible component of the LMC Queue Connector representing a stream operation performed on data in-between the source and sink. Outside of simple functions, this interface also allows for changing the data shape itself using stream buffering, aggregation, splitting functions.

3.6 Data Sink(s)

A *DataSink* is an extensible component of the LMC Queue Connector representing a location that data can be asynchronously (or synchronously) pushed to. The type data into the sink is determined by the dtype specified in the exchange descriptor and it's shape is calculated by the Data Pipe.

Note: Multiple data sinks are currently not supported on a single exchange (but trivial to implement). A temporary workaround to this is to use duplicate exchanges differing by sink.

3.7 Source/Sink Compatibility

Below is a complete compatibility matrix currently supported by the various sink and source combinations.

Sink/Sourc	bytes	object	int16 int32 int64	uint8 uint16 uint32 uint64	float32 float64	complex64 complex12	datetime64	*struct*
KafkaConsumerSource	✓	✓	✓	✓	✓	✓	✓	✓
TangoSubscriptionSource	✓		✓	✓	✓			
TangoPointingSubscriptionSource								✓
KafkaProducerSink	✓	✓	✓	✓	✓	✓	✓	✓
TangoLocalAttributeSink	✓		✓	✓	✓			
TangoArrayScatterSink			✓	✓	✓			
TangoObjectScatterAttributeSink		✓	✓	✓	✓			✓

3.8 Data Encoding

Certain sinks, sources and data using raw byte buffers may require selecting a suitable `Encoding` string.

3.8.1 Bytes Data Encoding

The *bytes* data format is unique in that an encoding string is paired and travels with the bytes value, similar to *tango.DevEncoded*.

Note: The encoding parameter is optional for `KafkaProducerSink` when using the *bytes* data type. No encoding or decoding is performed when used by `KafkaConsumerSource` or `KafkaProducerSink`.

3.8.2 Kafka Encoding

Since Kafka Topics only store raw bytes, `KafkaProducerSink` and `KafkaConsumerSource` must provide an encoding to describe how to read from/write to raw bytes.

Supported `encoding`/*`dtype`*/*`shape`* combinations are described in the below table:

Encoding	Compatible Dtypes	Compatible Shapes	Notes
"pytł (default)	"str", "dict", "bytes", "bool", "int[16|32|64]", "uint[8|16|32|64]", "float[32|64]"	[], [x], [x, y] (only [] for bytes and dict types)	Data is serialized using `repr(data).encode('utf-8')` and deserialized using `ast.literal_eval(data.decode('utf-8'))`.
"utf-	"str" "bytes", "bool", "int[16|32|64]", "uint[8|16|32|64]", "float[32|64]"	[]	Data is serialized using `str(data).encode('utf-8')` and deserialized using `dtype(data)`.
"asci	"str", "bytes", "bool", "int[16|32|64]", "uint[8|16|32|64]", "float[32|64]"	[]	Data is serialized using `str(data).encode('ascii')` and deserialized using `dtype(data)`.
"jsor	"str", "dict", "bytes", "bool", "int[16|32|64]", "uint[8|16|32|64]", "float[32|64]"	[], [x], [x, y] (only [] for bytes and dict types)	Data is serialized using `json.dumps(data).encode('utf-8')` and deserialized using `json.loads(data)`.
"msgp	"str", "dict", "bytes", "bool", "int[16|32|64]", "uint[8|16|32|64]", "float[32|64]"	[], [x], [x, y] (only [] for bytes and dict types)	Data is serialized using `msgpack.packb(value, default=msgpack_numpy.encode)` and deserialized using `msgpack.unpackb(value, object_hook=msgpack_numpy.decode)`
"carı	"bool", "int[16|32|64]", "uint[8|16|32|64]", "float[32|64]", List[tuple(str, Scalar)]	[], [x], [x, y]	Writes and zero-copy reads raw numpy buffer in row-major order, without any dimension information. `KafkaConsumerSource` will attempt to reshape according to the provided *shape* information.
"npy'	"str", "bool", "int[16|32|64]", "uint[8|16|32|64]", "float[32|64]", List[tuple(str, Scalar)]	[], [x], [x, y]	Uses `np.save()` and `np.load()`, includes dimension information.
any string	"bytes"	[]	If specified, the `KafkaProducerSink` will validate that the encoding of data matches the encoding from associated `DataSource`. For the `TangoSubscriptionSource`, this is set by Tango clients when they write a value. For `KafkaConsumerSource` this is specified in the configuration.
None	"bytes"	[]	

CHAPTER

FOUR

CONFIGURATION SCHEMA

The JSON configuration provided to the Queue Connector Device via Configuration Database, Property or Command Parameter must conform to the schema outlined by *QueueConnectorDescriptor*.

Note: Whilst it is valid to configure with no exchanges, doing so will keep the device in the **STANDBY** state. This behaviour may change in future.

4.1 Schema

4.1.1 QueueConnectorDescriptor

Primary JSON serializable descriptor for configuring a queue connector device. Note: exchanges as dictionary of groups is experimental and not fully supported via configuration database.

type	*object*			
properties				
• exchanges	*Exchanges*			
	default	null		
	anyOf	type	*object*	
		additionalProperties	type	*array*
			items	*ExchangeDescriptor*
		type	*array*	
		items	*ExchangeDescriptor*	
		type	*null*	
additionalProperties	False			

BufferWithTimePipeDescriptor

type	*object*			
properties				
• type	*Type*			
	default	BufferWithTimePipe		
	const	BufferWithTimePipe		
• timespan	*Timespan*			
	type	*number*		
	default	0		
additionalProperties	False			

DefaultPipeDescriptor

type	*object*			
properties				
• type	*Type*			
	default	DefaultPipe		
	const	DefaultPipe		
additionalProperties	False			

ExchangeDescriptor

Descriptor for instantiating an exchange.

type	*object*			
properties				
• dtype	*Dtype*			
	default	str		
	anyOf	type	*string*	
		type	*array*	
		items	type	*array*
			items	
			maxItems	3
			minItems	2
• shape	*Shape*			
	type	*array*		
	default			
	items			
• **source**	*Source*			
	anyOf	oneOf	*InMemorySourceDescriptor*	
			TangoSubscriptionSourceDescriptor	
			TangoPointingSubscriptionSourceDescriptor	
			KafkaConsumerSourceDescriptor	
		type	*array*	
		items	oneOf	*InMemorySourceDescriptor*

continues on next page

Table 1 – continued from previous page

			TangoSubscription-SourceDescriptor
			TangoPointing-Subscription-SourceDescriptor
			KafkaConsumer-SourceDescriptor
• **sink**	*Sink*		
	oneOf	*InMemorySinkDescriptor*	
		TangoLocalAttributeSinkDescriptor	
		TangoArrayScatterAttributeSinkDescriptor	
		TangoObjectScatterAttributeSinkDescriptor	
		TangoRemoteAttributeSinkDescriptor	
		KafkaProducerSinkDescriptor	
• pipe	*Pipe*		
	default	type DefaultPipe	
	oneOf	*DefaultPipeDescriptor*	
		BufferWithTimePipeDescriptor	
additionalProperties	False		

InMemorySinkDescriptor

Descriptor for instantiating an InMemorySink

type	*object*	
properties		
• type	*Type*	
	default	InMemorySink
	const	InMemorySink
• **key**	*Key*	
	type	*string*
additionalProperties	False	

InMemorySourceDescriptor

Descriptor for instantiating an InMemorySource

type	*object*			
properties				
• type	*Type*			
	default	InMemorySource		
	const	InMemorySource		
• **data**	*Data*			
	type	*array*		
	items	anyOf	type	*array*
			maxItems	2
			minItems	2
			type	*array*
			items	
			type	*object*
			type	*integer*
			type	*number*
			type	*boolean*
			type	*string*
• delay	*Delay*			
	type	*number*		
	default	0		
additionalProperties	False			

KafkaConsumerSourceDescriptor

type	*object*			
properties				
• type	*Type*			
	default	KafkaConsumerSource		
	const	KafkaConsumerSource		
• **servers**	*Servers*			
	anyOf	type	*string*	
		type	*array*	
		items	type	*string*
• **topic**	*Topic*			
	type	*string*		
• encoding	*Encoding*			
	type	*string*		
	enum	utf-8, ascii, python, json, msgpack_numpy, npy, carray		
	default	python		
additionalProperties	False			

KafkaProducerSinkDescriptor

type	*object*			
properties				
• type	*Type*			
	default	KafkaProducerSink		
	const	KafkaProducerSink		
• **servers**	*Servers*			
	anyOf	type	*string*	
		type	*array*	
		items	type	*string*
• **topic**	*Topic*			
	type	*string*		
• encoding	*Encoding*			
	type	*string*		
	enum	utf-8, ascii, python, json, msgpack_numpy, npy, carray		
	default	python		
• message_max_byte	*Message Max Bytes*			
	type	*integer*		
	default	1048576		
• timestamp_options	default	null		
	anyOf	*TimestampOptions*		
		type	*null*	
additionalProperties	False			

TangoArrayScatterAttributeSinkDescriptor

A Tango Attribute Sink for splitting and scattering of ndarray data.

type	*object*			
properties				
• type	*Type*			
	type	*string*		
	enum	TangoArrayScatterAttributeSink, TangoSplitAttributeSink		
	default	TangoArrayScatterAttributeSink		
• **attribute_names**	*Attribute Names*			
	type	*array*		
	items	type	*string*	
• axis	*Axis*			
	type	*integer*		
	default	0		
• default_value	*Default Value*			
	default			
	anyOf	type	*array*	
		maxItems	2	
		minItems	2	
		type	*array*	
		items		
		type	*object*	
		type	*integer*	

continues on next page

Table 2 – continued from previous page

		type	*number*	
		type	*boolean*	
		type	*string*	
• at-	*Attribute Shape Names*			
tribute_shape_n	default	null		
	anyOf	type	*array*	
		items	type	*string*
		type	*null*	
• indices	*Indices*			
	default	null		
	anyOf	type	*array*	
		items	type	*integer*
		type	*null*	
additionalProperties	False			

TangoAttributeDescriptor

An attribute descriptor describing a filter and path to to a value in JSON.

type	*object*			
properties				
• **at-**	*Attribute Name*			
tribute_name	type	*string*		
• **dtype**	*Dtype*			
	anyOf	type	*string*	
		type	*array*	
		items	type	*array*
			items	
			maxItems	3
			minItems	2
• shape	*Shape*			
	type	*array*		
	default			
	items	type	*integer*	
• path	*Path*			
	type	*string*		
	default	@		
• filter	*Filter*			
	default	null		
	anyOf	type	*string*	
		type	*null*	
• default_value	*Default Value*			
	default			
	anyOf	type	*array*	
		maxItems	2	
		minItems	2	
		type	*array*	
		items		
		type	*object*	
		type	*integer*	

continues on next page

Table 3 – continued from previous page

		type	*number*
		type	*boolean*
		type	*string*
additionalProperties	False		

TangoLocalAttributeSinkDescriptor

type	*object*		
properties			
• type	*Type*		
	default	TangoLocalAttributeSink	
	const	TangoLocalAttributeSink	
• **attribute_name**	*Attribute Name*		
	type	*string*	
• default_value	*Default Value*		
	default		
	anyOf	type	*array*
		maxItems	2
		minItems	2
		type	*array*
		items	
		type	*object*
		type	*integer*
		type	*number*
		type	*boolean*
		type	*string*
additionalProperties	False		

TangoObjectScatterAttributeSinkDescriptor

A Tango Attribute Sink for splitting and scattering object heirarchy data.

type	*object*	
properties		
• type	*Type*	
	default	TangoObjectScatterAttributeSink
	const	TangoObjectScatterAttributeSink
• **attributes**	*Attributes*	
	type	*array*
	items	*TangoAttributeDescriptor*
additionalProperties	False	

TangoPointingSubscriptionSourceDescriptor

type	*object*	
properties		
• type	*Type*	
	default	TangoPointingSubscriptionSource
	const	TangoPointingSubscriptionSource
• **device_name**	*Device Name*	
	type	*string*
• **attribute_name**	*Attribute Name*	
	type	*string*
• etype	*Etype*	
	type	*integer*
	default	0
• stateless	*Stateless*	
	type	*boolean*
	default	True
additionalProperties	False	

TangoRemoteAttributeSinkDescriptor

type	*object*	
properties		
• type	*Type*	
	default	TangoRemoteAttributeSink
	const	TangoRemoteAttributeSink
• **device_name**	*Device Name*	
	type	*string*
• **attribute_name**	*Attribute Name*	
	type	*string*
additionalProperties	False	

TangoSubscriptionSourceDescriptor

type	*object*	
properties		
• type	*Type*	
	default	TangoSubscriptionSource
	const	TangoSubscriptionSource
• **device_name**	*Device Name*	
	type	*string*
• **attribute_name**	*Attribute Name*	
	type	*string*
• etype	*Etype*	
	type	*integer*
	default	0
• stateless	*Stateless*	
	type	*boolean*
	default	True
additionalProperties	False	

TimestampOptions

A set of kafka producer options related to extracting Kafka timestamps from dynamic data. Timestamps in dynamic data must be one of:

- An offset to unix epoch in milliseconds.
- A numpy datetime64 on TAI scale.

type	*object*						
properties							
• slices	*Slices*						
	type	*array*					
	default						
	items	anyOf	type	*integer*			
			type	*array*			
			items	anyOf	type	*integer*	
					type	*null*	
			maxItems	3			
			minItems	3			
• key	*Key*						
	default	null					
	anyOf	type	*string*				
		type	*null*				
• reducer	*Reducer*						
	default	null					
	anyOf	type	*string*				
		enum	min, max, mean				
		type	*null*				
additional-Properties	False						

Note: **bold** indicates a required key-value pair.

4.2 Examples

4.2.1 Empty Config

```
{}
```

4.2.2 Image Config

```
{
    "exchanges": [
        {
            "dtype": "float32",
            "shape": [2,2],
            "source": {
                "type": "InMemorySource",
                "data": [
                    [[0,0],
                    [0,1]],
                    [1.0, 2.0],
                    2.0,
                    2.1,
                    2.5,
                ],
                "delay": 0.5
            },
            "sink": {
                "type": "KafkaProducerSink",
                "topic": "test-topic",
                "servers": "localhost",
                "encoding": "carray"
            }
        },
        {
            "dtype": "float32",
            "shape": [2,2],
            "source": {
                "type": "KafkaConsumerSource",
                "topic": "test-topic",
                "servers": "localhost",
                "encoding": "carray"
            },
            "sink": {
                "type": "TangoLocalAttributeSink",
                "attribute_name": "matrix",
                "default_value": 0
```

(continues on next page)

(continued from previous page)

```
            }
        }
    ]
}
```

CHAPTER

FIVE

API

5.1 SDP Queue Connector Device

class ska_sdp_lmc_queue_connector.sdp_queue_connector.**SDPQueueConnector**(**args: Any*, ***kwargs: Any*)

A dynamically configured tango device for sending and receiving data to and from data services.

exchanges_config_path: str

Configuration Database Path to a JSON configuration

generic_read(*attr: tango.Attr*)

Generic method for reading the internal device values into the output attribute argument.

Parameters

attr (*tango.Attr*) – output tango attribute

async group_configure(*configuration: str*, *group_name_override: str | None = None*)

Commands the device to load exchanges using a provided config json string.

Parameters

- **configuration** (*str | None*) – QueueConnectorDescriptor json. None
- **config.** (*value is considered as an empty*) –
- **group_name_override** (*str | None*) – When provided, overrides the
- **QueueConnectorDescriptor.** (*group name inside the*) –

async reconfigure(*group_name: str*, *config_value: str | None*)

Tries commanding to stop, reconfigure and start the device depending on the current state.

Parameters

config (*str | None*) – the new configuration json string.

set_attr(*name: str*, *value: Tuple[str, bytes] | numpy.ndarray | Dict[str, object] | int | float | bool | str*, *push_event=True*)

Sets the internal attribute and optionally pushes a change event.

Parameters

- **name** (*str*) – name of attribute to set
- **value** (*DataType*) – value to set
- **push_event** (*bool | None, optional*) – Whether to push an event.
- **to** (*Setting to None will push if polling is not active. Defaults*) –

- **None.** –

async watch_database_and_reconfigure(*database_path: str*)

Long running awaitable that watches the configuration database and automatically tries reconfiguring and the device.

Parameters
database_path (*str*) – database path to watch

class ska_sdp_lmc_queue_connector.sdp_queue_connector.**QueueConnectorDescriptor**(**args: Any*, ***kwargs: Any*)

Primary JSON serializable descriptor for configuring a queue connector device. Note: exchanges as dictionary of groups is experimental and not fully supported via configuration database.

exchanges: Dict[str | None, List[*ExchangeDescriptor*]] | List[*ExchangeDescriptor*] | None = None

Mapping or list of exchanges for a queue connector to process when running.

5.2 Exchange Interfaces

class ska_sdp_lmc_queue_connector.exchange.**ExchangeDescriptor**(**args: Any*, ***kwargs: Any*)

Descriptor for instantiating an exchange.

dtype: numpy.dtype = 'str'

Python primitive, numpy dtype or tango dtype of the dynamic attribute

pipe: DefaultPipeDescriptor | BufferWithTimePipeDescriptor

A pipe operator to be applied between source and sink read and write

shape: list = []

Data shape used by numpy and tango

sink: *InMemorySinkDescriptor* | *TangoLocalAttributeSinkDescriptor* | *TangoArrayScatterAttributeSinkDescriptor* | *TangoObjectScatterAttributeSinkDescriptor* | TangoRemoteAttributeSinkDescriptor | *KafkaProducerSinkDescriptor*

A data sink descriptor to be written to by the exchange

source: *InMemorySourceDescriptor* | *TangoSubscriptionSourceDescriptor* | *TangoPointingSubscriptionSourceDescriptor* | *KafkaConsumerSourceDescriptor* | list[*InMemorySourceDescriptor* | *TangoSubscriptionSourceDescriptor* | *TangoPointingSubscriptionSourceDescriptor* | *KafkaConsumerSourceDescriptor*]

One or more data source descriptors to be read by the exchange

class ska_sdp_lmc_queue_connector.exchange.**Exchange**(*sources: Sequence[*DataSource*]*, *sink:* DataSink, *pipe:* DataPipe)

A container representing a connection between a source and sink and handles asynchronous streaming between them.

async run()

Asynchronously runs the exchange connecting source payloads to the sink.

Running this to completion without exceptions guarentees no payloads are lost between sources and sinks.

async start()

Invokes start on the sinks and sources.

async stop()

Invokes stop on the sinks and sources.

5.3 Data Source, Sink and Pipe Interfaces

class ska_sdp_lmc_queue_connector.sourcesink.**DataSource**

Interface for an object containing data that can be asynchronously read.

abstract async start()

Asynchronously performs all additional initialization before reading (e.g. connecting to socket endpoints)

abstract async stop()

Asynchronously performs all end of stream destruction after reading (e.g. closing sockets)

class ska_sdp_lmc_queue_connector.sourcesink.**DataSink**

Interface for an object that receives data that can be asynchronously written to.

abstract async awrite(*value: Tuple[str, bytes] | numpy.ndarray | Dict[str, object] | int | float | bool | str*)

Writes a single data entry to the sink

abstract async start()

Asynchronously performs all additional initialization before writing (e.g. waiting for socket connections)

abstract async stop()

Asynchronously performs all end of stream destruction after reading (e.g. closing sockets)

class ska_sdp_lmc_queue_connector.sourcesink.**DataPipe**

Functor interface for pipe operators to perform on python data between a DataSource and DataSink.

abstract property output_dtype: numpy.dtype

Dtype of the functor stream output

abstract property output_shape: list[int]

Shape of the functor stream output

5.4 Tango

5.4.1 TangoSubscriptionSource

class ska_sdp_lmc_queue_connector.tango_sourcesink.**TangoSubscriptionSourceDescriptor**(**args: Any*, ***kwargs: Any*)

attribute_name: str

Attribute name the subscription is to

device_name: str

Device name containing the attribute the subscription is to

etype: SchemaEventType

The type of attribute event to listen for

stateless: bool = True

When True will retry subscribing every 10 seconds if failed subscription, otherwise will raise an exception on start if False.

class ska_sdp_lmc_queue_connector.tango_sourcesink.**TangoSubscriptionSource**(*desc:* TangoSubscriptionSourceDescriptor, *device:* SDPQueueConnector, *dtype: np.dtype*, *shape: list[int]*)

A DataSource populated from a subscription to tango attribute events.

start()

Asynchronously performs all additional initialization before reading (e.g. connecting to socket endpoints)

stop()

Asynchronously performs all end of stream destruction after reading (e.g. closing sockets)

5.4.2 TangoPointingSubscriptionSource

class ska_sdp_lmc_queue_connector.tango_pointing_source.**TangoPointingSubscriptionSourceDescriptor**(**args: Any*, ***kwa Any*)

class ska_sdp_lmc_queue_connector.tango_pointing_source.**TangoPointingSubscriptionSource**(*desc:* TangoPointingSubscriptionSourceDescriptor, *device:* SDPQueueConnector, *dtype: np.dtype*, *shape: list[int]*)

A specialized tango attribute source for converting pointings to a structured type.

5.4.3 TangoAttributeSink

class ska_sdp_lmc_queue_connector.tango_sourcesink.**TangoLocalAttributeSinkDescriptor**(**args: Any*, ***kwargs: Any*)

attribute_name: str

Attribute name to dynamically add to the SDPQueueConnector

default_value: SchemaDataType = ''

Starting value of the dynamic attribute before a source is read

class ska_sdp_lmc_queue_connector.tango_sourcesink.**TangoLocalAttributeSink**(*desc:* TangoLocalAttributeSinkDescriptor, *device:* SDPQueueConnector, *dtype: np.dtype*, *shape: list[int]*)

A DataSink that publishes data to a tango attribute on the Tango Device member.

awrite(*value: Tuple[str, bytes] | numpy.ndarray | Dict[str, object] | int | float | bool | str*)

Writes a single data entry to the sink

start()

Asynchronously performs all additional initialization before writing (e.g. waiting for socket connections)

stop()

Asynchronously performs all end of stream destruction after reading (e.g. closing sockets)

5.4.4 TangoArrayScatterAttributeSink

class ska_sdp_lmc_queue_connector.tango_array_scatter_sink.**TangoArrayScatterAttributeSinkDescriptor**(**arg*
Any,
***k*
Any)

A Tango Attribute Sink for splitting and scattering of ndarray data.

attribute_names: list[str]

Ordered attributute names to dynamically add the SDPQueueConnector

attribute_shape_names: list[str] | None = None

Optional attribute shapes names

axis: int = 0

Axis to split on. Non-zero values may reduce performance.

default_value: SchemaDataType = ''

Starting value of the dynamic attribute before a source is read

indices: list[int] | None = None

Optional ordered indexes of the split boundaries

class ska_sdp_lmc_queue_connector.tango_array_scatter_sink.**TangoArrayScatterAttributeSink**(*desc: TangoArrayScatterAttributeSinkDescriptor*, *device: SDPQueueConnector*, *dtype: np.dtype*, *shape: list[int]*)

awrite(*value: Tuple[str, bytes] | numpy.ndarray | Dict[str, object] | int | float | bool | str*)

Writes a single data entry to the sink

start()

Asynchronously performs all additional initialization before writing (e.g. waiting for socket connections)

stop()

Asynchronously performs all end of stream destruction after reading (e.g. closing sockets)

5.4.5 TangoObjectScatterAttributeSink

class ska_sdp_lmc_queue_connector.tango_object_scatter_sink.**TangoObjectScatterAttributeSinkDescriptor**(*

A

**

A

A Tango Attribute Sink for splitting and scattering object heirarchy data.

attributes: list[*TangoAttributeDescriptor*]

Attribute names to dynamically add the SDPQueueConnector

class ska_sdp_lmc_queue_connector.tango_object_scatter_sink.**TangoAttributeDescriptor**(**args: Any*, ***kwargs: Any*)

An attribute descriptor describing a filter and path to to a value in JSON.

attribute_name: str

Attribute name to dynamically add to the SDPQueueConnector

default_value: SchemaDataType = ''

Starting value of the dynamic attribute before a source is read

dtype: DType

Python primitive, numpy dtype or tango dtype of the dynamic attribute

filter: str | None = None

JMESPath predicate expression for whether to update the attribute

path: str = '@'

JMESPath expression for the location of the data to extract

shape: list[int] = []

Maximum shape of the dynamic attribute

class ska_sdp_lmc_queue_connector.tango_object_scatter_sink.**TangoObjectScatterAttributeSink**(*desc: TangoObjectScatterAttributeSinkDescriptor*, *device: SDPQueueConnector*, *dtype: np.dtype*, *shape: list[int]*)

awrite(*value: Tuple[str, bytes] | numpy.ndarray | Dict[str, object] | int | float | bool | str*)

Writes a single data entry to the sink

start()

Asynchronously performs all additional initialization before writing (e.g. waiting for socket connections)

stop()

Asynchronously performs all end of stream destruction after reading (e.g. closing sockets)

5.5 Kafka

5.5.1 KafkaConsumerSource

class ska_sdp_lmc_queue_connector.kafka_sourcesink.**KafkaConsumerSourceDescriptor**(**args: Any*, ***kwargs: Any*)

encoding: `Literal['utf-8', 'ascii', 'python', 'json', 'msgpack_numpy', 'npy', 'carray'] = 'python'`

Encoding of the message bytes

servers: `str | list[str]`

The Kafka broker(s) to query for metadata and setup the connection

topic: `str`

The Kafka topic to read from

class ska_sdp_lmc_queue_connector.kafka_sourcesink.**KafkaConsumerSource**(*descriptor: KafkaConsumerSourceDescriptor*, *dtype: numpy.dtype*, *shape: list[int]*)

A DataSource which consumes messages from a Kafka topic

start()

Asynchronously performs all additional initialization before reading (e.g. connecting to socket endpoints)

async stop()

Asynchronously performs all end of stream destruction after reading (e.g. closing sockets)

5.5.2 KafkaProducerSink

class ska_sdp_lmc_queue_connector.kafka_sourcesink.**KafkaProducerSinkDescriptor**(**args: Any*, ***kwargs: Any*)

class **TimestampOptions**(**args: Any*, ***kwargs: Any*)

A set of kafka producer options related to extracting Kafka timestamps from dynamic data. Timestamps in dynamic data must be one of:

- An offset to unix epoch in milliseconds.
- A numpy datetime64 on TAI scale.

key: `str | None = None`

Timestamp key for dictionary-like data types.

reducer: `Literal['min', 'max', 'mean'] | None = None`

Axes reduce operation for timestamps.

slices: `tuple[int | slice, ...] = ()`

Timestamp slice location for multidimensional data. Size must match number of dimensions.

encoding: `Literal['utf-8', 'ascii', 'python', 'json', 'msgpack_numpy', 'npy', 'carray'] = 'python'`

The encoding of the kafka message.

message_max_bytes: `int = 1048576`

The max size of encoded messages in bytes. NOTE: The 1MiB default is recommended by Kafka, sending larger messages higher than this will linearly increase latency.

servers: `str | list[str]`

The Kafka broker(s) to query for metadata and setup the connection

timestamp_options: *TimestampOptions* | None = None

An optional group of settings related to extracting Kafka timestamps from dynamic data. None results in using the current time.

topic: str

The Kafka topic to write messages to

class ska_sdp_lmc_queue_connector.kafka_sourcesink.**KafkaProducerSink**(*descriptor: KafkaProducerSinkDescriptor*, *dtype: numpy.dtype*, *shape: list[int]*)

A DataSink which produces messages for a Kafka topic

awrite(*value: Tuple[str, bytes] | numpy.ndarray | Dict[str, object] | int | float | bool | str*)

Writes a single data entry to the sink

start()

Asynchronously performs all additional initialization before writing (e.g. waiting for socket connections)

stop()

Asynchronously performs all end of stream destruction after reading (e.g. closing sockets)

5.6 In-Memory

5.6.1 InMemorySource

class ska_sdp_lmc_queue_connector.in_memory_sourcesink.**InMemorySourceDescriptor**(**args: Any*, ***kwargs: Any*)

Descriptor for instantiating an InMemorySource

data: list[Tuple[str, bytes] | numpy.ndarray | Dict[str, Any] | int | float | bool | str]

Data values to be read from the source to a sink

delay: float = 0

Time delay in seconds before the next data value is available to read

class ska_sdp_lmc_queue_connector.in_memory_sourcesink.**InMemorySource**(*desc: InMemorySourceDescriptor*, *dtype: numpy.dtype*, *shape: list[int] | None = None*)

An in-memory implementation of a DataSource for testing.

start()

Asynchronously performs all additional initialization before reading (e.g. connecting to socket endpoints)

stop()

Asynchronously performs all end of stream destruction after reading (e.g. closing sockets)

5.6.2 InMemorySink

class ska_sdp_lmc_queue_connector.in_memory_sourcesink.**InMemorySinkDescriptor**(**args: Any*, ***kwargs: Any*)

Descriptor for instantiating an InMemorySink

key: str

Key for accessing the stored data queue via classmethod

class ska_sdp_lmc_queue_connector.in_memory_sourcesink.**InMemorySink**(*desc:* InMemorySinkDescriptor)

An in-memory implementation of a DataSink for testing. Instances of
sink data queues exist as class members referenced via a lookup key.

awrite(*value: Tuple[str, bytes] | numpy.ndarray | Dict[str, object] | int | float | bool | str*)

Writes a single data entry to the sink

start()

Asynchronously performs all additional initialization before writing (e.g. waiting for socket connections)

stop()

Asynchronously performs all end of stream destruction after reading (e.g. closing sockets)

5.7 Pipes

class ska_sdp_lmc_queue_connector.pipe.default_pipe.**DefaultPipe**(*dtype: numpy.dtype*, *shape: list[int]*)

Interface for pipeline operators to perform on python data between a DataSource and DataSink

property output_dtype: numpy.dtype

Dtype of the functor stream output

property output_shape: list[int]

Shape of the functor stream output

class ska_sdp_lmc_queue_connector.pipe.buffer_pipe.**BufferWithTimePipe**(*desc: BufferWithTimePipeDescriptor*, *dtype: numpy.dtype*, *shape: list[int]*)

Buffers data with time window by dynamically by adding a dynamic dimension to the ouput shape and appending data until the time window closes.

property output_dtype: numpy.dtype

Dtype of the functor stream output

property output_shape: list[int]

Shape of the functor stream output

CHAPTER

SIX

GETTING STARTED

This page outlines instructions for how to setup an example pipeline using Queue Connector Device in the SDP Integration context.

6.1 Configuration

A JSON instantiation of *QueueConnectorDescriptor* serves as the configuration for the LMC QueueConnector is required to turn the device on. A minimal example of this is follows:

```
{
    "exchanges": [
        {
            "dtype": "string",
            "shape": [],
            "source": {
                "type": "KafkaConsumerSource",
                "servers": "localhost:9092",
                "topic": "test_topic",
                "encoding": "utf-8"
            },
            "sink": {
                "type": "TangoLocalAttributeSink",
                "attribute_name": "message"
            }
        }
    ]
}
```

This indicates to the queue connector to stream string values read from the test_topic topic on the Kafka server running on `localhost:9092` to a new scalar tango attribute named `message`.

6.2 Write to Configuration Database

The configuration database is the recommended approach to configuring the LMC Queue Connector by populating the *exchanges_config_path* device property. The SDP Integration Helm chart is setup to create one Queue Connector per subarray, each of which is configured to monitor JSON configs at direct child paths of `'/component/lmc-queueconnector-<subarray-id>/owner'` in the database.

Using `ska-sdp-config`, write the above configuration to the database location from a container inside the same cluster as the device, e.g.:

```
import ska_sdp_config
import json
configdb = ska_sdp_config.Config()
subarray_id = 1
for txn in configdb.txn():
    txn._create(
        f'/component/lmc-queueconnector-{subarray_id:02d}/owner/my_id',
        json.dumps({
            "exchanges": [
                {
                    "dtype": "string",
                    "shape": [],
                    "source": {
                        "type": "KafkaConsumerSource",
                        "servers": "localhost:9092",
                        "topic": "test_topic",
                        "encoding": "utf-8"
                    },
                    "sink": {
                        "type": "TangoLocalAttributeSink",
                        "attribute_name": "message"
                    }
                }
            ]
        })
    )
```

Note: This approach does not require the software performing configuration to install and use the tango API, only *ska-sdp-config*.

6.3 Subscribe to Exchange Sinks

Since the Queue Connector indefinitely streams data until instructed to stop via providing a new configuration, a component intended to receive data from the Queue Connector should subscribe to the endpoint the device writes to. In the case of a Tango attribute this can be done using tango subscriptions:

```
>>> import tango
>>> proxy = tango.DeviceProxy('test-sdp/queueconnector/01')
>>> proxy.subscribe_event("message", tango.EventType.CHANGE_EVENT, print)
```

6.4 Stream Data

For this example, the specified Kafka topic now automatically pushes data to the subscribed Queue Connector, and will start immediately after the new config is detected and read.

```
>>> import aiokafka
>>> import asyncio
>>> async def stream():
...     async with aiokafka.AIOKafkaProducer(boostrap_servers="test_topic") as producer:
...         for message in ["Hello", "world!"]:
...             await producer.send_and_wait(message)
...             await asyncio.sleep(1)
...
>>> asyncio.run(stream())
```

As data is sent to Kafka, the Tango subscription handler will trigger on background thread and print.

6.5 End Streaming

When the stream becomes idle (in this example after 2 seconds), any Queue Connector attributes will remain on the device, e.g.

```
>>> print(proxy.read_attribute("message"))
"World!"
```

At this point the device can either be stopped by writing an empty config to the configuration database or be reconfigured by writing a new configuration. Any existing tango attributes from the previous configuration will be removed (even if a newly detected config also defines them).

CHAPTER

SEVEN

DEVELOPER GUIDE

7.1 Get Started

7.1.1 Install dependencies

You will need:

- Python >=3.10
- Poetry >=1.2.2
- Docker

Before running `poetry install` to install the Python dependencies you will need a system tango library installed on your system (which is required by `pytango`).

For Debian/Ubuntu:

```
$ sudo apt update
$ sudo apt install -y curl git build-essential libboost-python-dev libtango-dev
```

Please note that:

- The `libtango-dev` will install an old version of the TANGO-controls framework (9.2.5);
- The best way to get the latest version of the framework is compiling it (instructions can be found here)
- MacOS is not supported
- Windows users will need to use WSL
- The above script has been tested with Ubuntu 22.04.

During this step, `libtango-dev` installation might ask for the Tango Server IP:PORT. Just accept the default proposed value.

Once you have that available you can install the python dependencies. Note that on some systems, you may need to explicitly provide the path to the tango C++ headers:

```
CPPFLAGS=-I/usr/include/tango poetry install
```

7.1.2 Run linting and testing

Since this project supports interfacing with Kafka, we need to spin up a instance for testing. For this we use Docker Compose so you will need to install docker engine, and docker compose.

When these are available you can run the tests using

```
$ poetry run make python-tests
```

Linting can be run in a similar way:

```
$ poetry run make python-lint
```

7.2 Other

7.2.1 Makefile targets

This project contains a Makefile which acts as a UI for building Docker images, testing images, and for launching interactive developer environments. For the documentation of the Makefile run `make help`.

7.2.2 TANGO References

- https://pytango.readthedocs.io/en/stable/contents.html
- https://pytango.readthedocs.io/en/stable/green_modes/green_modes_server.html
- https://pytango.readthedocs.io/en/stable/testing.html
- https://pytango.readthedocs.io/en/stable/client_api/index.html
- https://pytango.readthedocs.io/en/stable/server_api/server.html

CHAPTER

EIGHT

CHANGE LOG

All notable changes to this project will be documented in this file. This project adheres to Semantic Versioning.

8.1 [Development]

8.1.1 Added

- Added thread-safe async stream utilities.
- Enabled more warnings as errors in tests.

8.1.2 Changed

- **BREAKING** Renamed `Exchange.stream` to `Exchange.run`.
- Updated to pydantic ^2.6
- Updated to pytest ^8.1

8.2 [4.0.4]

8.2.1 Fixed

- Fixed dimensionality of 2D attributes created by the `TangoLocalAttributeSink`, dimensions were swapped.

8.3 [4.0.3]

8.3.1 Fixed

- `TangoArrayScatterSink` now correctly creating value attributes.

8.4 [4.0.2]

8.4.1 Fixed

- Device now automatically leaves INIT state when monitoring is configured.

8.5 [4.0.1]

8.5.1 Fixed

- Python logging setup now correctly follows the `-v` command line option.

8.6 [4.0.0]

8.6.1 Changed

- **BREAKING** Tango attributes representing exchange sinks are always read-only. The `access` option that allowed them to become read-write has been removed.
- `TangoObjectScatterAttributeSink` can now process arrays.

8.6.2 Removed

- **BREAKING** Removed all device commands
- **BREAKING** Removed *exchanges_json* tango property
- **BREAKING** Removed `poll_period`, `abs_change` and `rel_change` parameters on `TangoLocalAttributeSink` as polled Tango attributes are unable to be removed from the device.

[3.0.1]

8.6.3 Changed

- Changed configuration database state format to JSON dictionary
- Changed configuration database fault format to JSON dictionary

8.7 [3.0.0] [YANKED]

8.7.1 Added

- Added exchange groups for supporting multiple configs
- Added configuration database watching from multiple keys
- Added separate state machine per exchange group
- Added forwarding of exchange group state and exceptions to configuration database

8.7.2 Changed

- **BREAKING** Changed *Configure()* to no longer accept a SDP configuration path. Use *Monitor()* instead.
- **BREAKING** Changed config monitoring to monitor only direct child paths of *exchanges_config_path*.

8.8 [2.0.0]

8.8.1 Added

- Added *TimestampOptions* to *KafkaProducerSink*
- Added *python* encoding option
- Added *DataPipe* interface with *DefaultPipe* and *BufferWithTimePipe*
- Added *msgpack_numpy* to the list of available kakfa encodings
- Added Getting Started section to online documentation

8.8.2 Changed

- **BREAKING** Changed *TangoJsonScatterAttributeSink* to *TangoObjectScatterAttributeSink*
- **BREAKING** Changed to *python* as default encoding for *KafkaProducerSink* and *KafkaConsumerSource*
- Updated `ska-sdp-config` dependency from `0.5.1` to `0.5.2` for improved etcd performance. This removes the dependency on *git*.

8.8.3 Removed

- **BREAKING** Removed *"bytearray"* as an alias for *"bytes"* dtype
- **BREAKING** Removed *json* as a supported dtype. Use *object* with encoding *json*
- **BREAKING** Removed *"dict"* and *"list"* as supported dtypes. Use *object* instead.

8.9 [1.2.0]

8.9.1 Added

- Added *json* encoding type

8.9.2 Changed

- Updated `ska-sdp-config` dependency to `0.5.1` for improved etcd performance. This raises the minimum support version of etcd from `3.3` to `3.4`.

8.9.3 Fixed

- Fixed *TangoJsonScatterAttributeSink* default filter behaviour

8.10 [1.1.1]

8.10.1 Fixed

- Fixed default string values on *TangoJsonScatterAttributeSink*

8.11 [1.1.0]

8.11.1 Added

- Add limited support for numpy structured arrays when using InMemory and Kafka sources and sinks. Specify the structured dtype in the `exchanges[].dtype` field
- Added JSON/Dictionary types to exchanges
- Added TangoJsonScatterAttributeSink
- Added changelog to sphinx docs

8.11.2 Deprecated

- Deprecated *TangoSplitAttributeSink* in favor of *TangoArrayScatterAttributeSink*

8.11.3 Removed

- Removed ska-tango-base dependency

8.12 [1.0.0]

8.12.1 Added

- Added TangoSplitAttributeSink
- Added etcd3 path monitoring
- Added IsMonitoringDB command
- Added Tango command docstrings
- Added thread-safe SubscribeEventConditionContext

8.12.2 Changed

- Set correct server logging format
- Set default server logging level to INFO
- Updated Configure command behaviour
- Renamed "ExchangeDescriptorList" to "QueueConnectorDescriptor"

8.12.3 Fixed

- Bugfix for docs not displaying correctly

8.13 [0.2.2]

8.13.1 Added

- Support manual pushing of change events when polling not defined

8.13.2 Fixed

- Kafka consumer source performance improvements

8.14 [0.2.1]

8.14.1 Fixed

- Entrypoint module path bugfix
- Only log warning if a kafka sink is written to after being stopped

8.15 [0.2.0]

8.15.1 Added

- Extended documentation around configuration and data types

8.15.2 Changed

- Renamed project from ska-sdp-data-exchange to ska-sdp-lmc-queue-connector
- Renamed device class from DynamicExchangeDevice to SDPQueueConnector
- Renamed entrypoint from SDPExchange to SDPQueueConnector

8.15.3 Removed

- Removed "Data" and "Descriptor" from config type descriminators
- Removed "Data" from long class names

8.15.4 Fixed

- Bugfix for when stopping a kafka sink whilst it is sending

8.16 [0.1.0]

8.16.1 Added

- string, bytes, primitive and nparray data type support
- Tango subscription source and tango attribute sink
- Kafka Consumer source and Kafka Producer sink
- In-memory source and sink
- **Dynamic Exchange Device with configuration via a**
  Tango property or using the `Configure()` command providing JSON or a SDP Config path
- Empty Python project directory structure

INDEX

T

W