# developer.skao.int Documentation

*Release 0.5.0*

Marco Bartolini

Mar 13, 2024

# CONTENTS

This repository contains the library 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.

CHAPTER

ONE

# INSTALLATION AND USAGE

## 1.1 Install with pip

```
pip install ska-sdp-config --extra-index-url https://artefact.skao.int/repository/pypi-
↪internal/simple
```

## 1.2 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.

You can find `etcd` pre-built binaries, for Linux, Windows, and macOS, here: https://github.com/etcd-io/etcd/releases.

You can also use homebrew to install `etcd` on macOS:

```
brew install etcd
```

If you encounter issues follow: https://brewinstall.org/install-etcd-on-mac-with-brew/

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.script['name'], pb.script['version']))
```

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

## 1.3 Command line

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

*SDP command-line interface*

## 1.4 Running unit tests locally

You will need to have a database backend to run the tests as well. See "Basic usage" above for instructions on how to install an `etcd` backend on your machine.

Once you started the database (run `etcd` in the command line), you will be able to run the tests using pytest.

Alternative way is by using the two shell scripts in the scripts directory:

`docker_run_etcd.sh` -> Which runs etcd in a Docker container for testing the code. `docker_run_python.sh` -> Runs a python container and connects to the etcd instance.

Run the scripts from the root of the repository:

```
bash scripts/docker_run_etcd.sh
bash scripts/docker_run_python.sh
```

Once the container is started and mounted to the local directory.

Since the dependencies are managed by poetry, either run a poetry install, or pip install the repository (from the root):

```
pip install -e .
```

Then run the tests:

```
pytest tests/
```

CHAPTER

TWO

# DESIGN AND BEST PRACTICES

Quick points:

- Uses a key-value store
- Objects are represented as JSON
- Uses watchers on a key or range of keys to monitor for any updates

## 2.1 Transaction Basics

The SDP configuration database interface is built around the concept of transactions, i.e. blocks of read and write queries to the database state that are guaranteed to be executed atomically. For example, consider this code:

```
for txn in config.txn():
   a = txn.get('a')
   if a is None:
       txn.create('a', '1')
   else:
       txn.update('a', str(int(a)+1))
```

It is guaranteed that we increment the *'a'* key by exactly one here, no matter how many other processes might be operating on it. How does this work?

The way transactions are implemented follows the philosophy of Software Transactional Memory as opposed to a lock-based implementation. The idea is that all reads are performed, but all writes are actually delayed until the end of the transaction. So in the above example, *'a'* is actually read from the database, but the 'put' call is not performed immediately.

Once the transaction finishes (the end of the *for* loop), the transaction commit sends a single request to the database that updates all written values **only if** none of the read values have been written in the meantime. If the commit fails, we repeat the transaction (that's why it is a loop!) until it succeeds. The idea is that this is fairly rare, and repeating the transaction should typically be cheap.

## 2.2 Usage Guidelines

What does this mean for everyday usage? Transactions should be as self-contained as possible - i.e. they should explicitly contain all assumptions about the database state they are making. If we wrote the above transaction as follows:

```
for txn in config.txn():
   a = txn.get('a')

for txn in config.txn():
   if a is None:
       txn.create('a', '1')
   else:
       txn.update('a', str(int(a)+1))
```

A whole number of things could happen between the first and the second transaction:

1. The *'a'* key could not exist in the first transaction, but could have been created by the second (which would cause us to fail)
2. The *'a'* key could exist in the first transaction, but could have been deleted by the second (which would also cause the above to fail)
3. Another transaction might have updated the *'a'* key with a new value (which would cause that update to be lost)

A rule of thumb is that you should assume **nothing** about the database state at the start of a transaction. If you rely on something, you need to (re)query it after you enter it. If for some reason you couldn't merge the transactions above, you should write something like:

```
for txn in config.txn():
   a = txn.get('a')

for txn in config.txn():
   assert txn.get('a') == a, "database state independently updated!"
   if a is None:
       txn.create('a', '1')
   else:
       txn.update('a', str(int(a)+1))
```

This would especially catch case (3) above. This sort of approach can be useful when we want to make sub-transactions that only depend on a part of the overall state:

```
for txn in config.txn():
    keys = txn.list_keys('/as/')
for key in keys:
    for txn in config.txn():
        a = txn.get(key)
        # Safety check: Path might have vanished in the meantime!
        if a is None:
            break
        # ... do something that depends solely on existance of "key" ...
```

This can especially be combined with watchers (see below) to keep track of many objects without requiring huge transactions.

## 2.3 Wrapping transactions

The safest way to work with transactions is to make them as "large" as possible, spanning all the way from getting inputs to writing outputs. This should be the default unless we have a strong reason to do it differently (examples for such reasons would be transactions becoming too large, or transactions taking so long that they never finish - but either should be extremely rare).

However, in the context of a program with complex behaviour this might appear cumbersome: This means we have to pass the transaction object to every single method that could either read or write the state. An elegant way to get around this is to move such methods to a "model" class that wraps the transaction itself:

```
def IncrementModel(Transaction):
    def __init__(self, txn):
        self._txn = txn
    def increase(key):
        a = self._txn.get(key)
        if a is None:
            self._txn.create(key, '1')
        else:
            self._txn.update(key, str(int(a)+1))

# ...
for txn in config.txn():
   model = IncrementModel(txn)
   model.increase('a')
```

In fact, we can provide factory functions that entirely hide the transaction object from view:

```
def increment_txn(config):
    for txn in config.txn():
        yield IncrementModel(txn)

# ...
for model in increment_txn(config):
   model.increase('a')
```

We could wrap this model the same way again to build as many abstraction layers as we want - key is that high-level methods such as "increase" are now directly tied to the existence of a transaction object.

## 2.4 Dealing with roll-backs

Especially as we start wrapping transactions more and more, we must keep in mind that while we can easily "roll back" any writes of the transaction (as they are not actually performed immediately), the same might not be true for program state. So for instance, the following would be unsafe:

```
to_update = ['a','b','c']
for model in increment_txn():
    while to_update:
        model.increase(to_update.pop())
```

Clearly this transaction would work differently the second time around! For this reason it is a good idea to keep in mind that while we expect the *for* to only execute once, it is entirely possible that they would execute multiple times, and the code should be written accordingly.

Fortunately, this sort of occurrence should be relatively rare - the following might be more typical:

```
objects_found = []
for model in increment_txn():
    for obj in model.list_objects():
        if model.some_check(obj):
            LOGGER.debug(f'Found {obj}!')
            objects_found.append(obj)
```

In this case, *objects_found* might contain duplicate objects if the transaction repeats - which could be easily fixed by moving the initialisation into the *for* loop.

On the other hand, note that transaction loops might also lead to duplicated log lines here, which might be seen as confusing. In this case, this is relatively benign and therefore likely acceptable. It might be possible to generate log messages at the start and end of transactions to make this more visible.

Another possible approach could be to replicate the transaction behaviour: for example, we could make the logging calls to *IncrementModel*, which would internally aggregate the logging lines to generate, which *increement_txn* could then emit in one go once the transaction actually goes through.

## 2.5 Watchers

Occasionally we might want to actively track something in the configuration. For sake of example, let's say we want to wait for a key to appear so we can print it. A simple implementation using polling might look like the following:

```
while True:
    for txn in config.txn():
        line = txn.get('/line_to_print')
        if line is not None:
            txn.delete('/line_to_print')
    if line is not None:
        print(line)
    time.sleep(1)
```

(Note that we are making sure to print outside the transaction loop - otherwise lines might get printed multiple times if we were running more than one instance of this program in parallel!)

But clearly this is not very good - it re-queries the database every second, which adds database load *and* is pretty slow. Instead, we can use a watcher loop:

```
for watcher in config.watcher():
    for txn in watcher.txn():
        line = txn.get('/line_to_print')
        if line is not None:
            txn.delete('/line_to_print')
    if line is not None:
        print(line)
```

Note that we are calling *txn* on the *watcher* instead of *config*: What is happening here is that the *watcher* object collects keys read by the transaction, and only iterates once one of them has been written. It is a concept that has a lot in common with the transaction loop, except that while the transaction loop only iterates if the transaction is inconsistent, the watcher loop *always* iterates.

Note that you can have multiple separate transactions within a watcher loop, which however are not guaranteed to be consistent. For example:

```
for watcher in config.watcher():
    for txn in watcher.txn():
        line = txn.get('/line_to_print')
    print('A:', line)
    for txn in watcher.txn():
        line = txn.get('/line_to_print')
    print('B:', line)
```

In this program we might get different results for *A* and *B*. However, the watcher *does* guarantee that the loop will iterate if any of the read values have been invalidated. So if the line was deleted between the two transaction, the following output would be generated:

```
A: something
B: None
A: None
B: None
```

After all, while transaction *B* had a current view of the situation the first time around, the view of transaction *A* became out-of-date.

By default, the watcher only iterates if any values read by a watcher transaction has changed. This may take an arbitrary amount of time (including infinite amount), hence we can "force" the watcher loop to go to its next iteration via two methods. A default *timeout* can be set either upon initiation:

```
for watcher in etcd3.watcher(timeout=60):
    ...
```

or manually with the *watcher.set_timeout(<new_timeout>)* method. The *timeout* is valid for the whole life-cycle of the watcher. Alternatively, you can set a "wake-up call", on a loop-by-loop basis, using the *watcher.set_wake_up_at(<value_of_alarm>)* method. This guarantees that the watcher will wake up at the given time or earlier (specified as an absolute datetime object). This especially means that if the method gets called multiple times, the watcher will wake up at the earliest of the times specified, either by *timeout* or by any of the *wake_up* calls.

## 2.6 New Backend Implementation

We are currently in the process of implementing a new backend using the *python-etcd3* client - https://github.com/etcd-io/etcd/releases. We found out that there are several issues with the current client (*etcd3-py*) and had to implement workarounds and fixes to ensure the SDP Configuration Library keeps working. Also, this client is no longer maintained by its developers. New client is better maintained and is expected to give an enhanced database connection performance.

More details of our investigation into the client can be found here - https://confluence.skatelescope.org/display/SE/Investigation+of+python-etcd3+client+package

The current *etcd3* backend has been renamed to *etcd3_revolution1* which uses *etcd3-py* python client. We will continue to support it until the new backend is ready to be replaced.

CHAPTER

# THREE

# CONFIGURATION SCHEMA

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

## 3.1 Execution Block

Path /eb/[eb_id]

Dynamic state information of the execution block.

Contents:

```
{
    "eb_id": "eb-mvp01-20200425-00000",
    "max_length": 21600.0,
    "scan_types": [
        { "scan_type_id": "science", ... },
        { "scan_type_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,
    "last_updated": "2022-08-01 10:01:12"
}
```

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

## 3.2 Processing Block

Path: /pb/[pb_id]

Static definition of processing block information.

Contents:

```
{
    "pb_id": "pb-mvp01-20200425-00000",
```

(continues on next page)

(continued from previous page)

```
    "eb_id": "eb-mvp01-20200425-00000",
    "script": {
        "kind": "realtime",
        "name": "vis_receive",
        "version": "0.2.0"
    },
    "parameters": { ... }
}
```

There are two kinds of processing, real-time and batch (offline). 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 kinds are `realtime` and `batch`. The script tag identifies the processing script version as well as the required underlying software (e.g. execution engines, processing components). `...` stands for arbitrary processing script-defined parameters.

### 3.2.1 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_id": "science", ... },
        { "scan_type_id": "calibration", ... },
    ],
    "last_updated": "2022-08-01 10:01:12"
}
```

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 processing, such as the status and receive addresses (for ingest).

### 3.2.2 Processing Block Owner

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

Identifies the process executing the script. 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-script-2kxfz",
```

(continues on next page)

(continued from previous page)

```
  "pid": 1
}
```

CHAPTER

FOUR

# CONFIGURATION API

## 4.1 High-Level API

## 4.2 Entities

### 4.2.1 Processing Block

### 4.2.2 Deployment

## 4.3 Backends

### 4.3.1 Common

Common functionality for implementing backends.

**exception** `ska_sdp_config.backend.common.`**`ConfigCollision`**(*path: str*, *message: str*)

Exception generated if key to create already exists.

**exception** `ska_sdp_config.backend.common.`**`ConfigVanished`**(*path: str*, *message: str*)

Exception generated if key to update that does not exist.

`ska_sdp_config.backend.common.`**`depth_of_path`**(*path: str*) → int

Get the depth of a path, this is the number of "/" in it.

**Returns**
the depth

### 4.3.2 Etcd3 backend

### 4.3.3 Etcd3 backend revolution 1

### 4.3.4 Memory backend

CHAPTER

FIVE

# SDP COMMAND-LINE INTERFACE

Command Line Interface: `ska-sdp`

To run the CLI, you must start a shell in the console pod (assuming you have SDP deployed in Kubernetes/Minikube, for instructions follow: SDP standalone).

```
kubectl exec -it ska-sdp-console-0 -n <namespace> -- bash
```

Once in, to access the help window of `ska-sdp`, run:

```
ska-sdp -h
```

## 5.1 Command - SDP Object matrix

This is a table/matrix of the existing commands of `ska-sdp` and what they can do with a specific SDP Object.

Commands:

- list
- get/watch
- create
- update/edit
- end
- delete
- import

SDP Objects:

- pb (processing block)
- script (processing script definition)
- deployment
- eb (execution block)
- controller (Tango controller device)
- subarray (Tango subarray device)

| | pb | script | deployment | eb | other |
|---|---|---|---|---|---|
| **list** | • list all pbs<br>• list pbs for a certain date | • list all script definitions<br>• list a script def of a specific kind (batch or realtime) | list all deployments | list all ebs | • if **-a \| –all**: list all the contents of the Config DB<br>• if **-v \| –values**: list keys with values (or just values)<br>• if **–prefix**: list limited to this prefix (for testing purposes)<br>• if controller, list the device entry if there is one<br>• if subarray, list all subarray device entries |
| **get/watch** | • get the value of a single key<br>• get the values of all pb-related keys for a single pb-id | get the value of a single key | get the value of a single key | get the value of a single key | *Note: rules for get and watch are the same* |
| **create** | • create a pb to **run a processing script**<br>• if **–eb**: add eb parameters for real-time pb | create a key/value pair with prefix of /script | create a deployment of **given deployment-id, kind, and parameters** | create a key/value pair with prefix of /eb | *Not implemented for Tango devices* |

## 5.2 Relevant environment variables

Backend-related:

```
SDP_CONFIG_BACKEND   Database backend (default etcd3)
SDP_CONFIG_HOST      Database host address (default 127.0.0.1)
SDP_CONFIG_PORT      Database port (default 2379)
SDP_CONFIG_PROTOCOL  Database access protocol (default http)
SDP_CONFIG_CERT      Client certificate
SDP_CONFIG_USERNAME  User name
SDP_CONFIG_PASSWORD  User password
```

When running *ska-sdp edit*:

```
EDITOR    Executable of an existing text editor. Recommended: vi, vim, nano (i.e.␣
↪command line-based editors)
```

## 5.3 Usage

```
> ska-sdp --help

Command line utility for interacting with SKA Science Data Processor (SDP).

Usage:
    ska-sdp COMMAND [options] [SDP_OBJECT] [<args>...]
    ska-sdp COMMAND (-h|--help)
    ska-sdp (-h|--help)

SDP Objects:
    pb            Interact with processing blocks
    script        Interact with available processing script definitions
    deployment    Interact with deployments
    eb            Interact with execution blocks
    controller    Interact with Tango controller device
    subarray      Interact with Tango subarray device

Commands:
    list            List information of object from the Configuration DB
    get | watch     Print all the information (i.e. value) of a key in the Config DB
    create          Create a new, raw key-value pair in the Config DB;
                    Run a processing script; Create a deployment
    update          Update a raw key value from CLI
    edit            Edit a raw key value from text editor
    delete          Delete a single key or all keys within a path from the Config DB
    end             Stop/Cancel execution block
    import          Import processing script definitions from file or URL
```

```
> ska-sdp list --help

List keys (and optionally values) within the Configuration Database.
```

(continues on next page)

(continued from previous page)

```
Usage:
    ska-sdp list (-a |--all) [options]
    ska-sdp list [options] pb [<date>]
    ska-sdp list [options] script [<kind>]
    ska-sdp list [options] (deployment|eb|controller|subarray)
    ska-sdp list (-h|--help)

Arguments:
    <date>      Date on which the processing block(s) were created. Expected format:␣
↪YYYYMMDD
                If not provided, all pbs are listed.
    <kind>      Kind of processing script definition. Batch or realtime.
                If not provided, all scripts are listed.

Options:
    -h, --help         Show this screen
    -q, --quiet        Cut back on unnecessary output
    -a, --all          List the contents of the Config DB, regardless of object type
    -v, --values       List all the values belonging to a key in the config db; default:␣
↪False
    --prefix=<prefix>  Path prefix (if other than standard Config paths, e.g. for␣
↪testing)
```

```
> ska-sdp (get|watch) --help

Get/Watch all information of a single key in the Configuration Database.

Usage:
    ska-sdp (get|watch) [options] <key>
    ska-sdp (get|watch) [options] pb <pb-id>
    ska-sdp (get|watch) (-h|--help)

Arguments:
    <key>       Key within the Config DB.
                To get the list of all keys:
                    ska-sdp list -a
    <pb-id>     Processing block id to list all entries and their values for.
                Else, use key to get the value of a specific pb.

Options:
    -h, --help    Show this screen
    -q, --quiet   Cut back on unnecessary output
```

```
> ska-sdp create --help

Create SDP objects (deployment, script, eb) in the Configuration Database.
Create a processing block to run a script.

Usage:
    ska-sdp create [options] pb <script> [<parameters>] [--eb=<eb-parameters>]
```

(continues on next page)

(continued from previous page)

```
    ska-sdp create [options] deployment <item-id> <kind> <parameters>
    ska-sdp create [options] (script|eb) <item-id> <value>
    ska-sdp create (-h|--help)

Arguments:
    <script>            Script that the processing block will run, in the format:
                            kind:name:version
    <parameters>        Optional parameters for a script, with expected format:
                            '{"key1": "value1", "key2": "value2"}'
                        For deployments, expected format:
                            '{"chart": <chart-name>, "values": <dict-of-values>}'
    <eb-parameters>     Optional eb parameters for a real-time script
    <item-id>           Id of the new deployment, script or eb
    <kind>              Kind of the new deployment (currently "helm" only)

Options:
    -h, --help     Show this screen
    -q, --quiet    Cut back on unnecessary output

Example:
    ska-sdp create eb eb-test-20210524-00000 '{"test": true}'
    Result in the config db:
        key: /eb/eb-test-20210524-00000
        value: {"test": true}

Note: You cannot create processing blocks apart from when they are called to run a␣
↪script.
```

```
> ska-sdp (update|edit) --help

Update the value of a single key or processing block state.
Can either update from CLI, or edit via a text editor.

Usage:
    ska-sdp update [options] (script|eb|deployment) <item-id> <value>
    ska-sdp update [options] pb-state <item-id> <value>
    ska-sdp update [options] controller <value>
    ska-sdp update [options] subarray <item-id> <value>
    ska-sdp edit (script|eb|deployment) <item-id>
    ska-sdp edit pb-state <item-id>
    ska-sdp edit controller
    ska-sdp edit subarray <item-id>
    ska-sdp (update|edit) (-h|--help)

Arguments:
    <item-id>   id of the script, eb, deployment, processing block or subarray
    <value>     Value to update the key/pb state with.

Options:
    -h, --help    Show this screen
    -q, --quiet   Cut back on unnecessary output
```

(continues on next page)

(continued from previous page)

```
Note:
    ska-sdp edit needs an environment variable defined:
        EDITOR: Has to match the executable of an existing text editor
                Recommended: vi, vim, nano (i.e. command line-based editors)
        Example: EDITOR=vi ska-sdp edit <key>
    Processing blocks cannot be changed, apart from their state.

Example:
    ska-sdp edit eb eb-test-20210524-00000
        --> key that's edited: /eb/eb-test-20210524-00000
    ska-sdp edit script batch:test:0.0.0
        --> key that's edited: /script/batch:test:0.0.0
    ska-sdp edit pb-state some-pb-id-0000
        --> key that's edited: /pb/some-pb-id-0000/state
```

```
> ska-sdp delete --help

Delete a key from the Configuration Database.

Usage:
    ska-sdp delete (-a|--all) [options] (pb|script|eb|deployment|prefix)
    ska-sdp delete [options] (pb|eb|deployment) <item-id>
    ska-sdp delete [options] script <script>
    ska-sdp delete (-h|--help)

Arguments:
    <item-id>   ID of the processing block, or deployment, or execution block
    <script>    Script definition to be deleted. Expected format: kind:name:version
    prefix      Use this "SDP Object" when deleting with a non-object-specific, user-
↪defined prefix

Options:
    -h, --help              Show this screen
    -q, --quiet             Cut back on unnecessary output
    --prefix=<prefix>       Path prefix (if other than standard Config paths, e.g. for␣
↪testing)
```

```
> ska-sdp end --help

End execution block in the configuration database.
By default it sets the status to FINISHED. If the --cancel flag is set, it sets
the status to CANCELLED.

Usage:
    ska-sdp end eb <eb-id> [options]
    ska-sdp end (-h|--help)

Arguments:
<eb-id>    ID of execution block to end

Options:
```

(continues on next page)

(continued from previous page)

```
    -c, --cancel  Cancel the execution block
    -h, --help    Show this screen
    -q, --quiet   Cut back on unnecessary output
```

```
> ska-sdp import --help

Import processing script definitions into the Configuration Database.

Usage:
    ska-sdp import scripts [options] <file-or-url>
    ska-sdp import (-h|--help)

Arguments:
    <file-or-url>      File or URL to import script definitions from.

Options:
    -h, --help          Show this screen
    --sync              Delete scripts not in the input
```

## 5.4 Example script definitions file

You can also use a script definitions file to import processing scripts into the Config DB. An example script definitions file looks like

```
scripts:
- kind: realtime
  name: test_realtime
  version: 0.2.2
  image: artefact.skao.int/ska-sdp-script-test-realtime:0.2.2
- kind: batch
  name: test_batch
  version: 0.2.2
  image: artefact.skao.int/ska-sdp-script-test-batch:0.2.2
```

Both YAML and JSON files are accepted. After the import, you can check via .. code-block:: bash

> ska-sdp list script

It will output a list of processing scripts that are available to use.

CHAPTER

SIX

# INDICES AND TABLES

- genindex
- modindex

# PYTHON MODULE INDEX

# INDEX