

Embedding a debugger in your PyTango device

Anton Joubert (SARAO)

Matteo Di Carlo (INAF)

Is it possible to debug a device?

Yes and we have tested the following options:

- The pdb debugger
- Remote debugging with debugpy (vscode)

Note that there are other options.

Pdb debugger

Pdb is a python module which defines an interactive source code debugger

It does not have the ability to attach to a running program

So to use it we must go into the container, kill the running device server and restart it with the debugger module

Pdb debugger

Note that by default the pdb debugger works only with the main thread.

```
tango@centralnode-01-0 /app$ python3 -m pdb /usr/local/bin/CentralNodeDS 01
> /usr/local/bin/CentralNodeDS(3)<module>()
-> import re
(Pdb) break /usr/local/lib/python3.7/dist-packages/centralnode/central_node.py:127
Breakpoint 1 at /usr/local/lib/python3.7/dist-packages/centralnode/central_node.py:127
(Pdb) continue
Downloading https://hpiers.obspm.fr/iers/bul/bulc/Leap_Second.dat
|=====
1|2021-03-04T10:53:03.600Z|INFO|MainThread|write_loggingLevel|base_device.py#1001|tango-device:
1|2021-03-04T10:53:03.601Z|INFO|MainThread|update_logging_handlers|base_device.py#305|tango-dev
1|2021-03-04T10:53:03.602Z|DEBUG|MainThread|_init_logging|base_device.py#691|tango-device:ska_m
1|2021-03-04T10:53:03.669Z|INFO|MainThread|exit|core.py#131||Finished processing state UNINITIA
1|2021-03-04T10:53:03.669Z|INFO|MainThread|enter|core.py#125||Finished processing state INIT er
1|2021-03-04T10:53:03.669Z|INFO|MainThread|_update_state|base_device.py#821|tango-device:ska_mi
1|2021-03-04T10:53:03.670Z|INFO|MainThread|callbacks|core.py#1084||Executed callback '<bound me
1|2021-03-04T10:53:03.670Z|DEBUG|MainThread|do|base_device.py#616|tango-device:ska_mid/tm_centra
1|2021-03-04T10:53:03.671Z|DEBUG|MainThread|do|base_device.py#630|tango-device:ska_mid/tm_centra
1|2021-03-04T10:53:03.671Z|INFO|MainThread|do|base_device.py#635|tango-device:ska_mid/tm_centra
> /usr/local/lib/python3.7/dist-packages/centralnode/central_node.py(127)do()
-> self.logger.info("Device initialising...")
(Pdb) continue
1|2021-03-04T10:53:09.141Z|INFO|MainThread|do|central_node.py#127|tango-device:ska_mid/tm_centra
1|2021-03-04T10:53:09.142Z|DEBUG|MainThread|do|central_node.py#139|tango-device:ska_mid/tm_cent
1|2021-03-04T10:53:09.143Z|INFO|MainThread|do|central_node.py#172|tango-device:ska_mid/tm_centra
1|2021-03-04T10:53:09.143Z|INFO|MainThread|_call_do|commands.py#212|tango-device:ska_mid/tm_cer
1|2021-03-04T10:53:09.145Z|INFO|MainThread|exit|core.py#131||Finished processing state INIT exi
1|2021-03-04T10:53:09.146Z|INFO|MainThread|enter|core.py#125||Finished processing state OFF ent
1|2021-03-04T10:53:09.146Z|INFO|MainThread|_update_state|base_device.py#821|tango-device:ska_mi
1|2021-03-04T10:53:09.147Z|INFO|MainThread|callbacks|core.py#1084||Executed callback '<bound me
Ready to accept request
□
```

Pdb debugger: Threading

```
tango@centralnode-01-0:/app$ python3 -m pdb /usr/local/bin/CentralNodeDS 01  
> /usr/local/bin/CentralNodeDS(3)<module>()  
-> import re  
(Pdb) break /usr/local/lib/python3.7/dist-packages/centralnode/central_node.py:314  
Breakpoint 1 at /usr/local/lib/python3.7/dist-packages/centralnode/central_node.py:314  
(Pdb) continue  
> /usr/local/bin/CentralNodeDS(9)<module>()
```

```
# -*- coding: utf-8 -*-  
import re  
import sys  
from centralnode.central_node import main  
if __name__ == '__main__':  
    sys.argv[0] = re.sub(r'(-script\.pyw|\.exe)?$', '', sys.argv[0])  
    import pdb; pdb.Pdb(nosigint=True).set_trace()  
    sys.exit(main())
```

```
def StartUpTelescope(self):  
    """  
    This command invokes SetOperateMode() command on DishLeafNode,  
    SdpMasterLeafNode and SubarrayNode and sets the CentralNode to  
    Operate Mode.  
    """  
    import pdb; pdb.Pdb(nosigint=True).set_trace()  
    handler = self.get_command_object("StartupTelescope")  
    (result_code, message) = handler()  
    return [[result_code], [message]]
```

```
7Z|INFO|MainThread|write_loggingLevel|base_device.py#1001|tango-device:sk  
7Z|INFO|MainThread|update_logging_handlers|base_device.py#305|tango-de  
7Z|DEBUG|MainThread|_init_logging|base_device.py#691|tango-device:sk  
5Z|INFO|MainThread|exit|core.py#131||Finished processing state UNINITI  
5Z|INFO|MainThread|enter|core.py#125||Finished processing state INITI  
5Z|INFO|MainThread|_update_state|base_device.py#821|tango-device:sk  
5Z|INFO|MainThread|callbacks|core.py#1084||Executed callback '<bound m  
1|2021-03-04T11:11:08.077Z|DEBUG|MainThread|do|base_device.py#616|tango-device:sk  
4T11:11:08.077Z|DEBUG|MainThread|do|base_device.py#630|tango-device:sk  
4T11:11:08.077Z|INFO|MainThread|do|base_device.py#635|tango-device:sk  
4T11:11:08.077Z|INFO|MainThread|do|central_node.py#127|tango-device:sk  
4T11:11:08.077Z|DEBUG|MainThread|do|central_node.py#139|tango-device:sk  
4T11:11:08.078Z|INFO|MainThread|do|central_node.py#172|tango-device:sk  
4T11:11:08.078Z|INFO|MainThread|_call_do|commands.py#212|tango-device:sk  
4T11:11:08.078Z|INFO|MainThread|exit|core.py#131||Finished processing state INITI  
4T11:11:08.078Z|INFO|MainThread|enter|core.py#125||Finished processing state OFF  
4T11:11:08.078Z|INFO|MainThread|_update_state|base_device.py#821|tango-device:sk  
1|2021-03-04T11:11:08.078Z|INFO|MainThread|callbacks|core.py#1084||Executed callback '<bound m  
Ready to accept request  
1|2021-03-04T11:11:13.042Z|DEBUG|Dummy-1|log_stream|log4tango.py#138|tango-device:sk  
> /usr/local/lib/python3.7/dist-packages/centralnode/central_node.py(312)StartUpTelescope()  
-> handler = self.get_command_object("StartupTelescope")  
(Pdb) |
```

Remote debugging

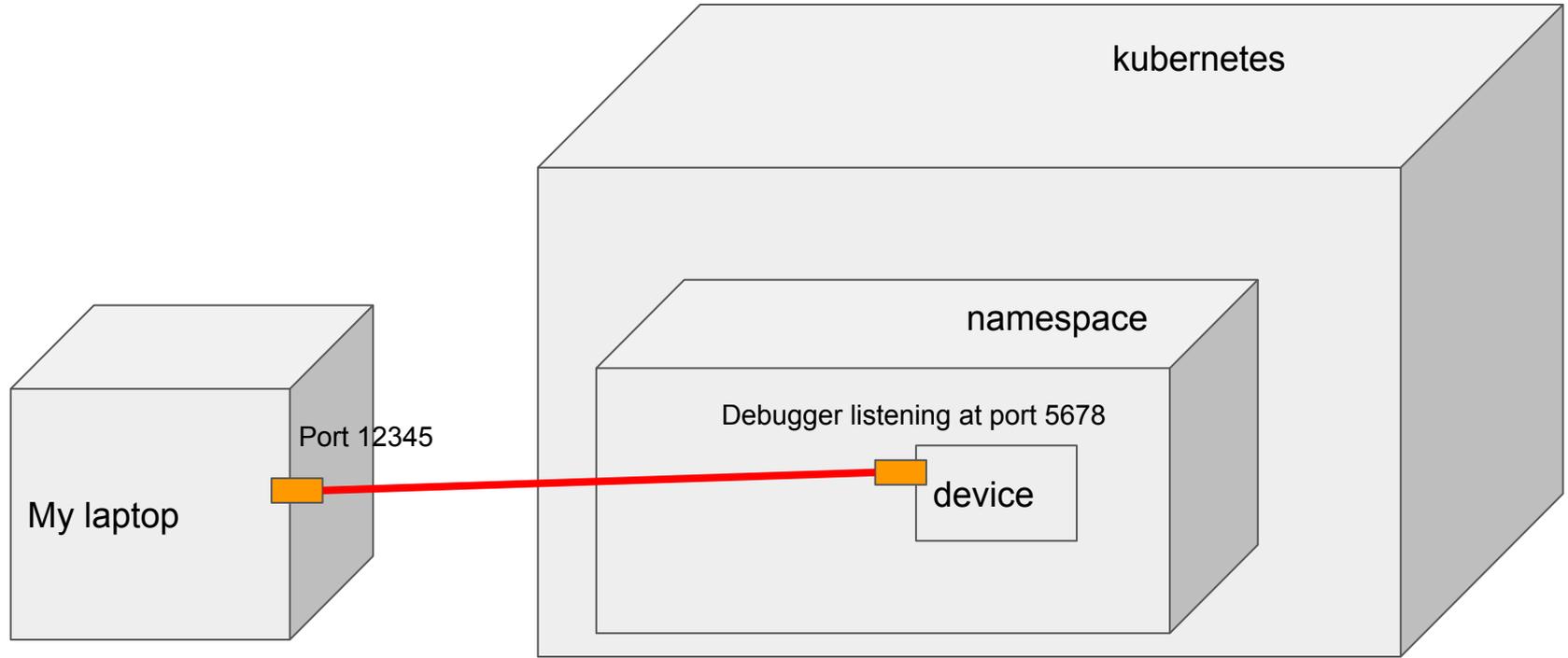
The remote debugging requires a debugger listening from a port in the remote machine (in our case a container)

If the debugger is listening to a port, we need to port-forward it to a local port (feature of kubernetes)

```
$ kubectl port-forward centralnode-01-0 12345:5678 -n  
ci-debug-matt
```

With non-kubernetes machines, ssh could do something similar.

As if the debugger listen to a local port of my laptop



debugpy library

vscode extensions uses this python library for remote debugging:

<https://github.com/microsoft/debugpy>

It is an adapter of the pydevd used in PyCharm

<https://github.com/fabioz/PyDev.Debugger/>

How it works:

- CLI: `python -m debugpy --listen localhost:5678 mydevice.py`
- From code:
 - `import debugpy`
 - `debugpy.listen(5678)`

Debug this thread method

A standard TANGO Device server does not use Python threads so most method calls are not debuggable unless we make them aware of the debugger.

Threads per device server? $8 + \textit{num polling} + \textit{num connected clients}$

Debug this thread method

Same situation of the pdb debugger (debugpy uses the standard pdb library).

In every method we want to debug we must add the following line of code:

```
debugpy.debug_this_thread()
```

https://github.com/microsoft/debugpy/wiki/API-Reference#debug_this_thread

Makes the debugger aware of the current thread, and start tracing it. Must be called on any background thread that is started by means other than the usual Python APIs (i.e. the threading module), in order for breakpoints to work on that thread.

Using vscode: `.vscode/launch.json`

We must configure the vscode to work with the remote debugger

```
{
  "configurations": [
    {
      "name": "Python: Remote Attach",
      "type": "python",
      "request": "attach",
      "connect": {
        "host": "127.0.0.1",
        "port": 12345,
        "justMyCode": false,
      },
      "pathMappings": [
        {
          "localRoot": "${workspaceFolder}",
          "remoteRoot": "/app"
        }
      ]
    }
  ]
}
```

PyCharm?

PyCharm debugger probably won't work for Kubernetes, as PyCharm hosts the debugger service.

It could work in environments that allow a connection to be established from the device server to the PyCharm host.

Just press the play button

The image shows a Visual Studio Code editor window with a Python project. The main editor displays the code for `central_node.py`. A red circle highlights the play button in the top toolbar. The code includes a `@command` decorator and a `@debugIt` decorator. A `def debug_this_thread()` function is also visible, with a red circle around its definition. The terminal window at the bottom shows a `kubectl` command being executed, with a red circle around the terminal area. The terminal output shows the command `skubectl port-forward centralnode-01-0 12345:5678 -n ci-debug-matt` and its output.

```
central_node.py x health_state_aggregator.py tmc-mid-config.json launch.py
VARIABLES
central_node.py x health_state_aggregator.py tmc-mid-config.json launch.py
332 return handler.check_allowed()
333
334 @command(
335     dtype_out="DevVarLongStringArray",
336     doc_out="[ResultCode, information-only string]",
337 )
338 @debugIt()
339 def StartUpTelescope(self):
340     """
341     This command invokes SetOperateMode() command on DishLeadNode,
342     SdpMasterLeafNode and SubarrayNode and sets the Central Node ir
343     """
344     debugpy.debug_this_thread()
345     handler = self.get_command_object("StartUpTelescope")
346     (result_code, message) = handler(
347         [result_code], [message])
348
349 def is_AssignResources_allowed(self):
350     """
351     Checks whether this command is allowed to be run in current dev
352     """
353     :return: True if this command is allowed to be run in current
354
355     :rtype: boolean
356
357     :raises: DevFailed if this command is not allowed to be run in current device state
358
359     """
360     debugpy.debug_this_thread()
361     handler = self.get_command_object("AssignResources")
362     return handler.check_allowed()
363
364 @command(
365     dtype_in="str",
366     doc_in="The string in JSON format. The JSON contains following values:\nsubarrayID: "
367     "Devshort\ndish: JSON object consisting of\n- receptorIDList: DevVarStringArray, "
368     "The individual string should contain dish numbers in string format with "
```

```
PROBLEMS 27 OUTPUT DEBUG CONSOLE TERMINAL
tango-test-base-test-test-0 1/1 Running 0 3m18s
tm-alarms-handler-01-0 1/1 Running 0 3m12s
tmc-mid-configuration-test-7w6w6 0/1 Completed 0 3m11s
vcc-001-vcc-001-0 1/1 Running 0 3m22s
vcc-002-vcc-002-0 1/1 Running 0 3m23s
vcc-003-vcc-003-0 1/1 Running 0 3m22s
vcc-004-vcc-004-0 1/1 Running 0 3m22s
ubuntu@LAPTOP-SLBC3H83:~/skamp1$ kubectl port-forward centralnode-01-0 12345:5678 -n ci-debug-matt
Forwarding from 127.0.0.1:12345 -> 5678
Forwarding from [::]:12345 -> 5678
```

Adding to the SKABaseDevice?

```
src > ska_tango_base > base_device.py > SKABaseDevice > DebugDevice
```

```
1725
1726 @command(
1727     dtype_out="DevUShort",
1728     doc_out="The TCP port the debugger is listening on."
1729 )
1730 def DebugDevice(self):
1731     """
1732     Enables remote debugging of this device.
1733
1734     Starts the ``debugpy`` debugger listening for remote connections
1735     (via Debugger Adaptor Protocol), and patches all methods so that
1736     they can be debugged.
1737
1738     If the debugger is already listening, additional e.
1739     command will trigger a breakpoint.
1740     """
1741     command = self.get_command_object("DebugDevice")
1742     return command()
```

1. Add new command: DebugDevice

2. Patches (almost) all the methods

[See source code on gitlab](#)

```
src > ska_tango_base > base_device.py > SKABaseDevice > DebugDeviceCommand > patch_method
```

```
1701
1702 def patch_method_for_debugger(self, name, method):
1703     """Ensure method calls trigger the debugger.
1704
1705     Most methods in a device are executed by calls from threads spawned
1706     by the cppTango layer. These threads are not known to Python, so
1707     we have to explicitly inform the debugger about them.
1708     """
1709
1710     def debug_thread_wrapper(orig_method, *args, **kwargs):
1711         debugpy.debug_this_thread()
1712         return orig_method(*args, **kwargs)
1713
1714     patched_method = partial(debug_thread_wrapper, method)
1715     setattr(self.target, name, patched_method)
```

Adding to the SKABaseDevice?

```
src > ska_tango_base > base_device.py > SKABaseDevice > DebugDevice
```

```
1725
1726 @command(
1727     dtype_out="DevUShort",
1728     doc_out="The TCP port the debugger is listening on."
1729 )
1730 def DebugDevice(self):
1731     """
1732     Enables remote debugging of this device.
1733
1734     Starts the ``debugpy`` debugger listening for remote connections
1735     (via Debugger Adaptor Protocol), and patches all methods so that
1736     they can be debugged.
1737
1738     If the debugger is already listening, additional e
1739     command will trigger a breakpoint.
1740     """
1741     command = self.get_command_object("DebugDevice")
1742     return command()
```

1. Add new command: DebugDevice

2. Patches (almost) all the methods

[See source code on gitlab](#)

```
src > ska_tango_base > base_device.py > SKABaseDevice > DebugDeviceCommand > patch_method
```

```
1701
1702 def patch_method_for_debugger(self, name, method):
1703     """Ensure method calls trigger the debugger.
1704
1705     Most methods in a device are executed by calls from threads spawned
1706     by the cppTango layer. These threads are not known to Python, so
1707     we have to explicitly inform the debugger about them.
1708     """
1709
1710     def debug_thread_wrapper(orig_method, *args, **kwargs):
1711         debugpy.debug_this_thread()
1712         return orig_method(*args, **kwargs)
1713
1714     patched_method = partial(debug_thread_wrapper, method)
1715     setattr(self.target, name, patched_method)
```

Example

1. Use DeviceProxy to enable

2. Press play

```
>>> import tango
>>> dp = tango.DeviceProxy("tango://172.17.0.3")
>>> dp.DebugDevice()
5678
>>>
```

3. Set your breakpoint

```
>>> dp.On()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

4. Cause execution
(command may timeout on
client side)

The screenshot shows a Python IDE interface with a remote connection to a device. The top toolbar includes a play button (labeled '2. Press play') and a breakpoint icon (labeled '3. Set your breakpoint'). The main editor displays the code for `base_device.py`, with a breakpoint set at line 1604. The left sidebar shows the `VARIABLES` and `CALL STACK` panels. The `CALL STACK` panel shows the current execution state: `MainThread` (PAUSED), `Thread-1` (PAUSED), and `Dummy-2` (PAUSED ON BREAKPOINT). The `CALL STACK` details for `Dummy-2` are as follows:

| Function | File | Line |
|----------------------|----------------|--------|
| do | base_device.py | 1604:1 |
| _call_do | commands.py | 207:1 |
| __call__ | commands.py | 273:1 |
| On | base_device.py | 1638:1 |
| debug_thread_wrapper | base_... | |

```
base_device.py — ska-tango-base
src > ska_tango_base > base_device.py > SKABaseDevice > OnCommand > do
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618

def do(self):
    """
    Stateless hook for On() command functionality.

    :return: A tuple containing a return code and a string
             message indicating status. The message is for
             information purpose only.
    :rtype: (ResultCode, str)
    """
    message = "On command completed OK"
    self.logger.info(message)
    return (ResultCode.OK, message)

def is_on_allowed(self):
    """
    Check if command `On` is allowed in the current device state.

    :raises `tango.DevFailed`: if the command is not allowed

    :return: `True` if the command is allowed
    :rtype: boolean
    """
    command = self.get_command_object("On")
    return command.check_allowed()
```

Example

1. Use DeviceProxy to enable

2. Press play

```
>>> import tango
>>> dp = tango.DeviceProxy("tango://172.17.0.3")
>>> dp.DebugDevice()
5678
>>>
```

3. Set your breakpoint

```
>>> dp.On()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

4. Cause execution
(command may timeout on
client side)

The screenshot shows a Python IDE interface with a remote connection to a device. The top toolbar includes a play button (labeled '2. Press play') and a breakpoint icon (labeled '3. Set your breakpoint'). The main editor displays the code for `base_device.py`, with a breakpoint set at line 1604. The left sidebar shows the `VARIABLES` and `CALL STACK` panels. The `CALL STACK` panel shows the current execution state: `MainThread` (PAUSED), `Thread-1` (PAUSED), and `Dummy-2` (PAUSED ON BREAKPOINT). The `CALL STACK` details for `Dummy-2` are as follows:

| Function | File | Line |
|----------------------|----------------|--------|
| do | base_device.py | 1604:1 |
| _call_do | commands.py | 207:1 |
| __call__ | commands.py | 273:1 |
| On | base_device.py | 1638:1 |
| debug_thread_wrapper | base_... | |

```
base_device.py — ska-tango-base
src > ska_tango_base > base_device.py > SKABaseDevice > OnCommand > do
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618

def do(self):
    """
    Stateless hook for On() command functionality.

    :return: A tuple containing a return code and a string
             message indicating status. The message is for
             information purpose only.
    :rtype: (ResultCode, str)
    """
    message = "On command completed OK"
    self.logger.info(message)
    return (ResultCode.OK, message)

def is_on_allowed(self):
    """
    Check if command `On` is allowed in the current device state.

    :raises `tango.DevFailed`: if the command is not allowed

    :return: `True` if the command is allowed
    :rtype: boolean
    """
    command = self.get_command_object("On")
    return command.check_allowed()
```

Conclusion

- pdb debugger from console (no attach option)
- debugpy and vscode
- Listening debugpy session:
 - May affect performance
 - Is a security hole
 - Cannot be turned off again

Both pdb and debugpy require a trace for non-Python threads.

[SKABaseDevice source code on gitlab](#)