Unit testing PyTango devices

Tango Nov 2020 status update meeting

SQUARE KILOMETRE ARRAY

Exploring the Universe with the world's largest radio telescope

Anton Joubert & Drew Devereux

Online / 18 November 2020

Presenters

Anton Joubert
South Africa

Drew Devereux
Australia

Acknowledgements

Additional Contributors:

Giorgio Brajnik
Katleho Madisa
Paul Swart
Samuel Twum
Johan Venter
Sett Wai

Overview

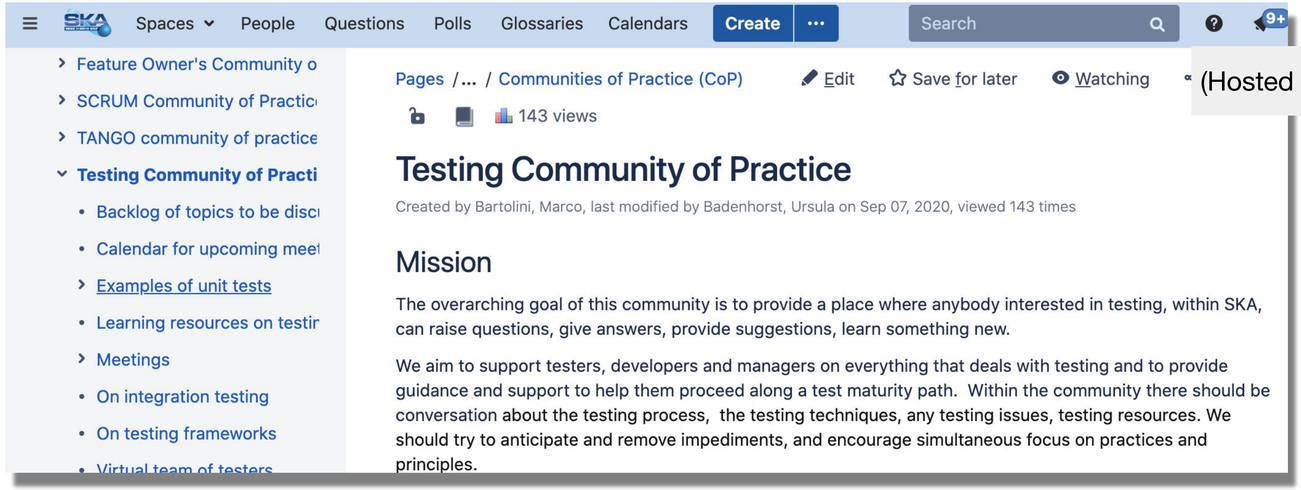
Software Testing in SKA

Testing with *DeviceTestContext*

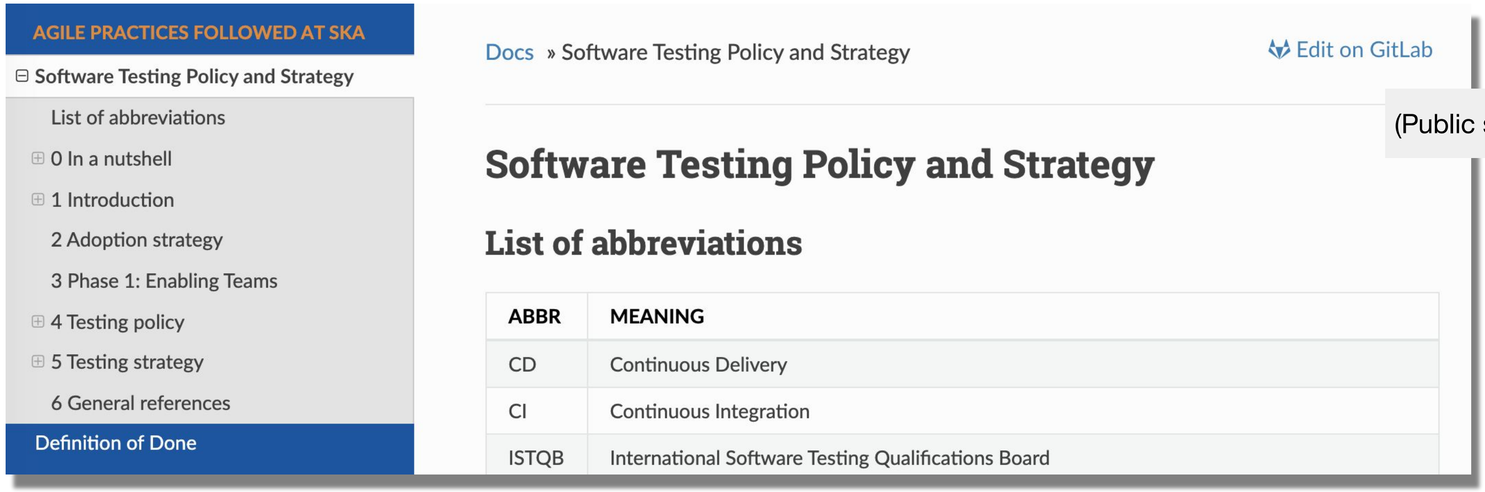
Mocking *DeviceProxy*

Pytest examples

Software Testing in SKA



(Hosted on Confluence - SKA access only)



(Public site)

https://developer.skatelescope.org/en/latest/development_practices/ska_testing_policy_and_strategy.html

Types of software testing

Unit

Integration

System (BDD and acceptance)

Static API verification: YAML spec (via tango-simlib)

Contract testing: Pact (consumer driven)

Types of software testing

Unit (testing Tango devices)

Integration

System (BDD and acceptance)

Static API verification: YAML spec (via tango-simlib)

Contract testing: Pact (consumer driven)

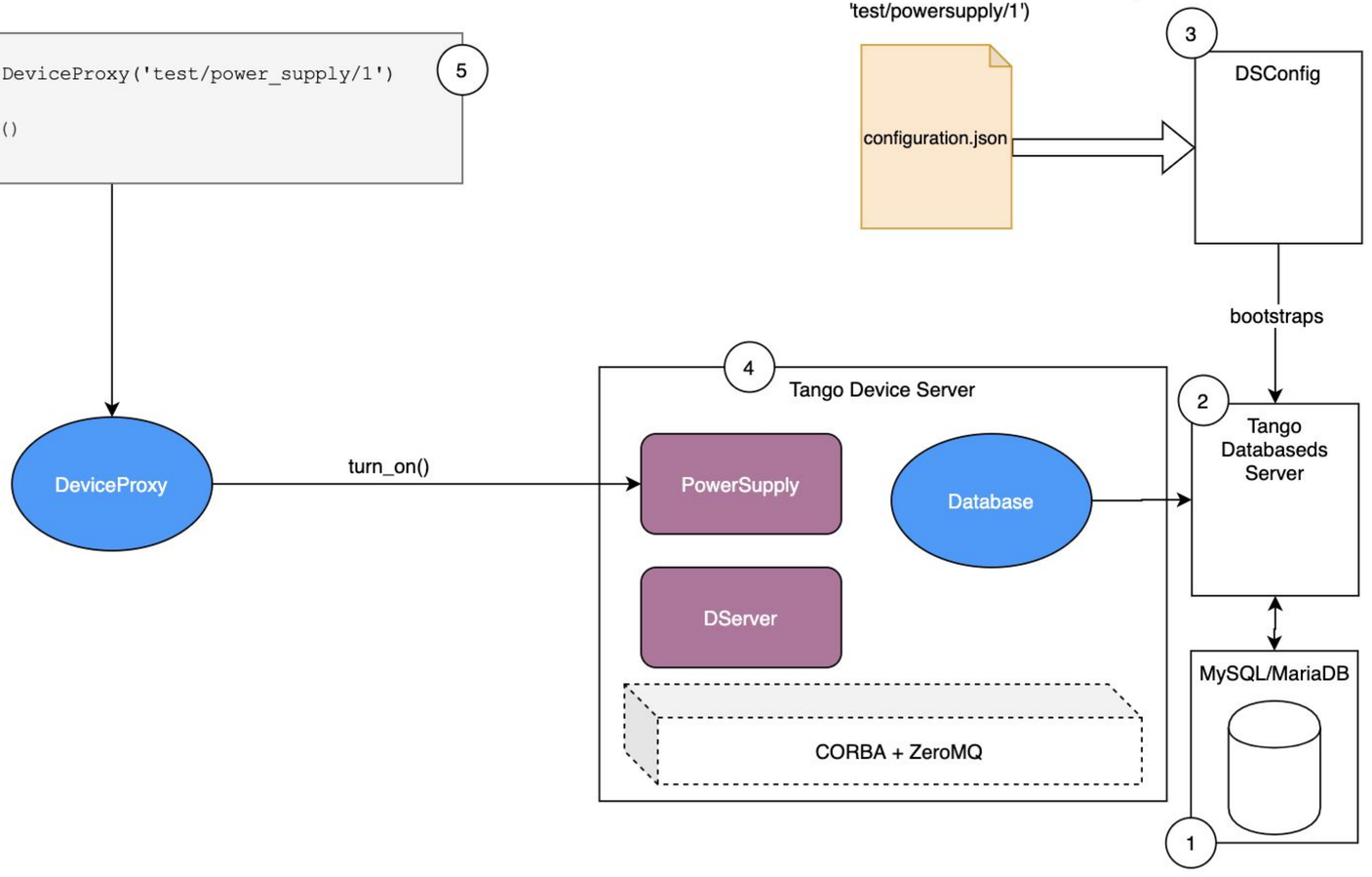
Testing with DeviceTestContext

Testing with real Tango facility

Test runner

```
powersupply_proxy = DeviceProxy('test/power_supply/1')

powersupply.turn_on()
```

5



DS Configuration

Specifies the list of Tango Device Servers, Tango Devices and their initial properties.

Configuration for the device under test should be specified in here ('test/powersupply/1')

DSConfig

A tool to configure the initial state of the Tango Database.

It reads the configuration from a file (JSON).

Tango Databaseds Server

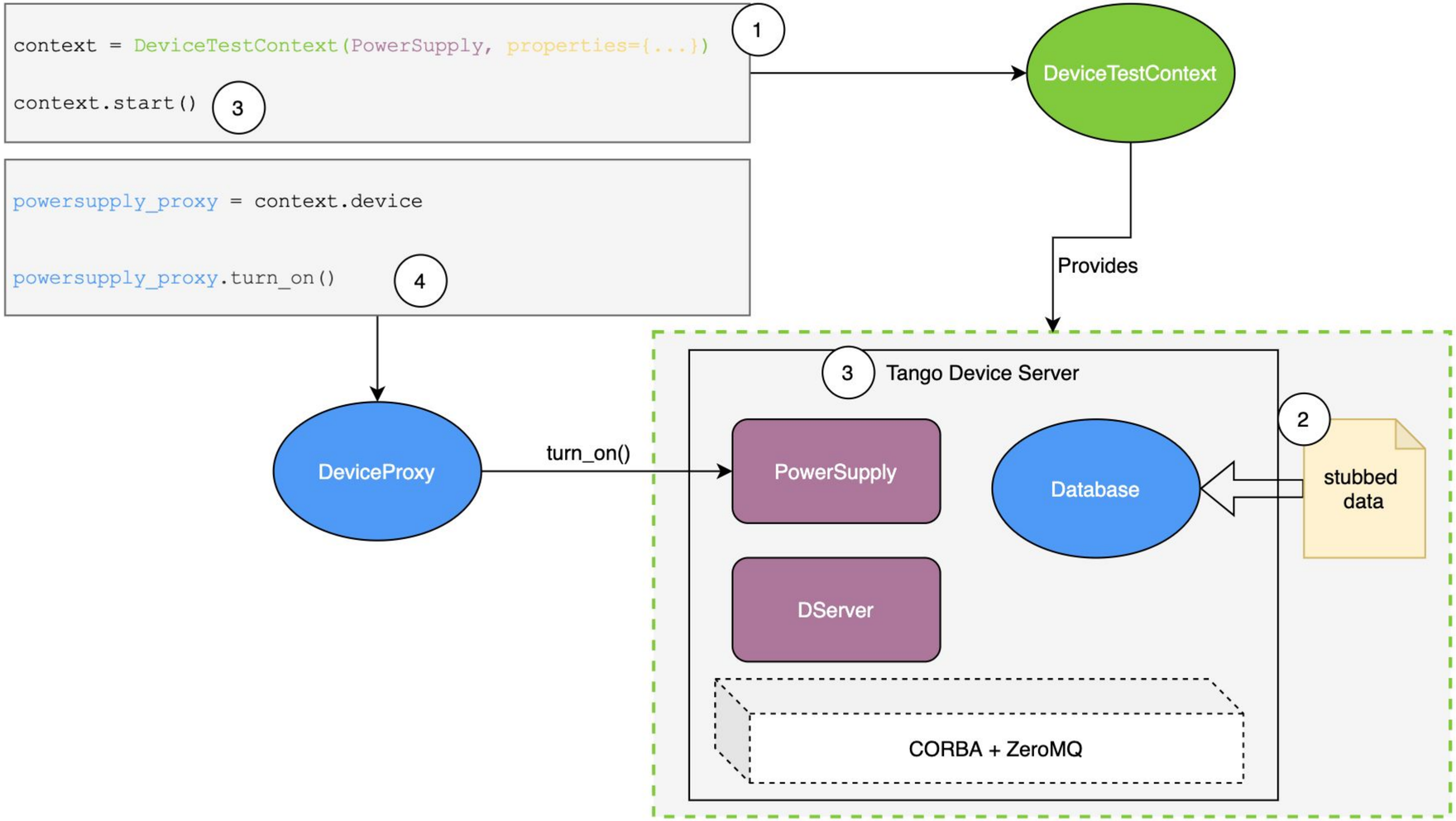
The Tango Database that acts as a registry and data store for Tango Devices.

It is backed by MySQL or MariaDB.

Testing with DeviceTestContext

```
context = DeviceTestContext(PowerSupply, properties={...})
context.start()
```

```
powersupply_proxy = context.device
powersupply_proxy.turn_on()
```

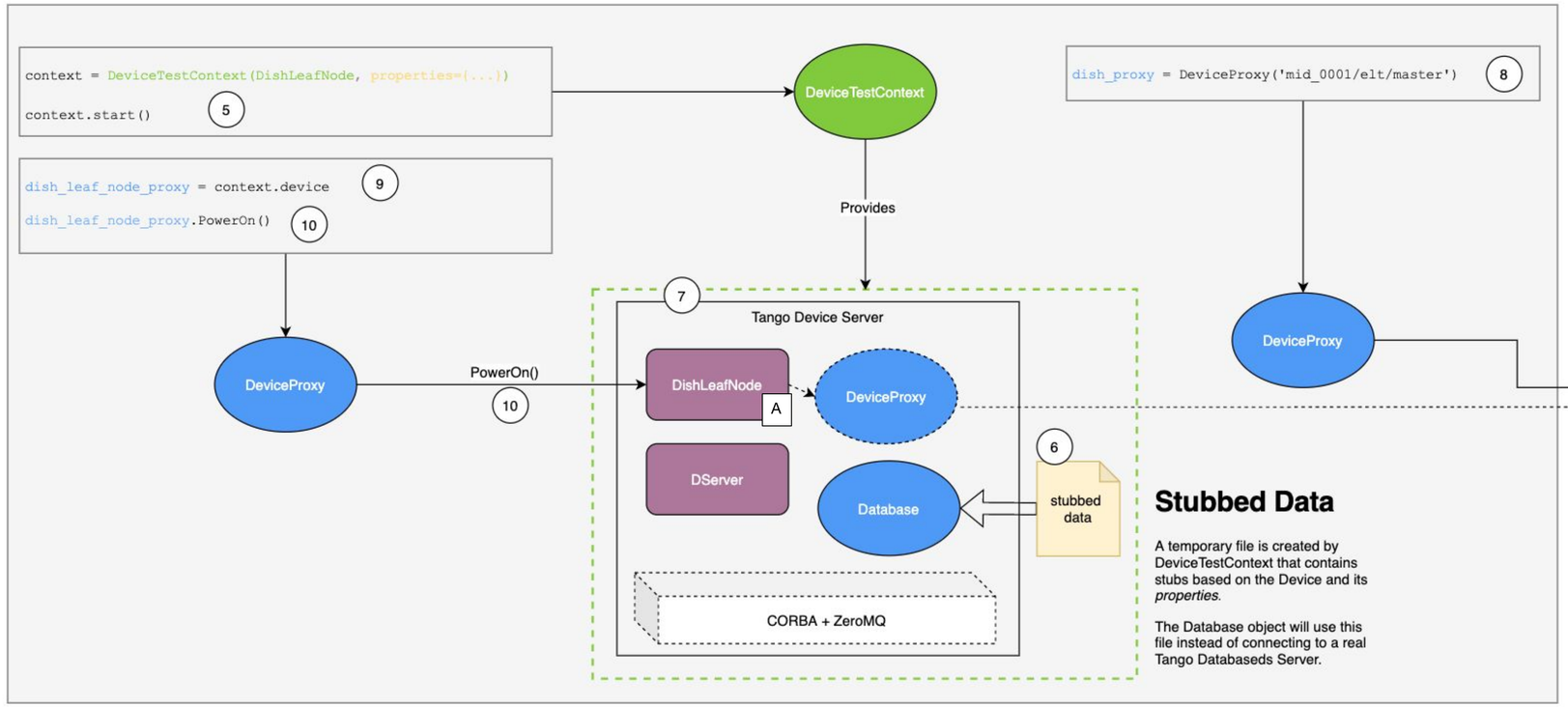


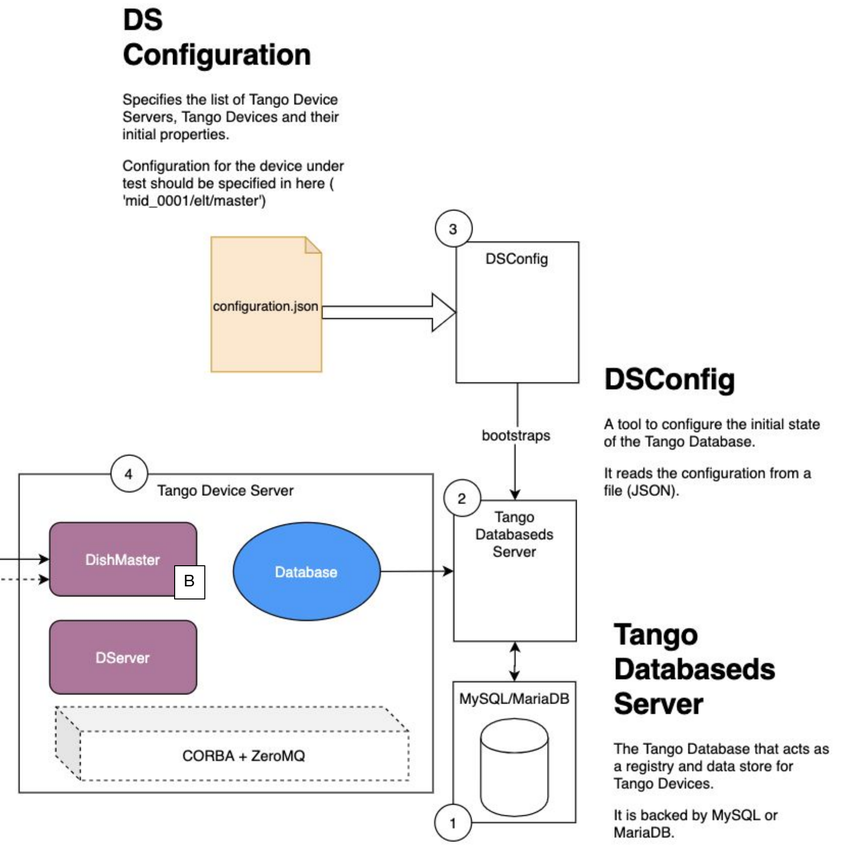
Stubbed Data

A temporary file is created by DeviceTestContext that contains stubs based on the Device and its *properties*.

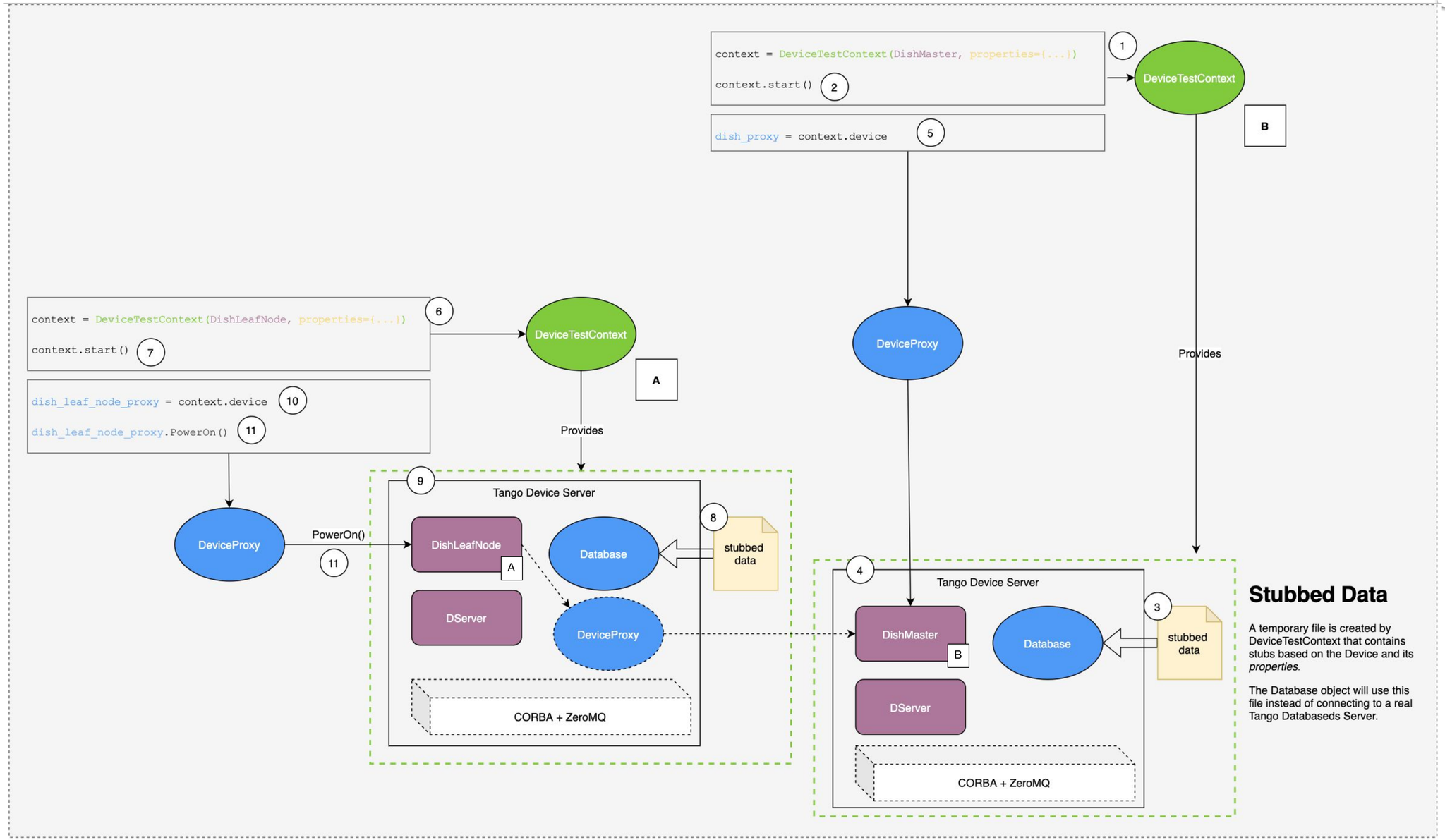
The Database object will use this file instead of connecting to a real Tango Databaseds Server.

Hybrid: DeviceTestContext + real

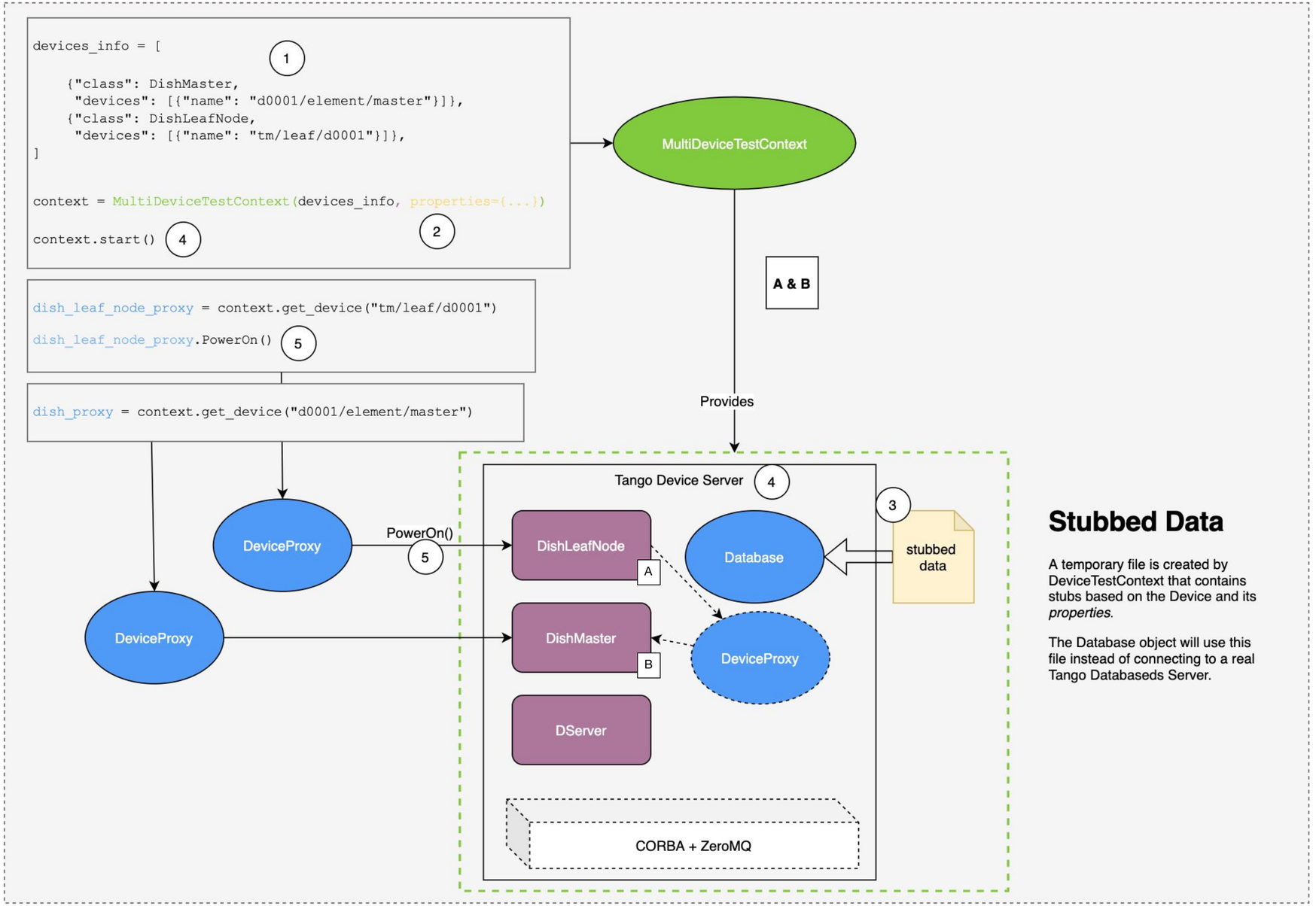




2 x DeviceTestContext (not recommended)



Testing with MultiDeviceTestContext



[Multi]DeviceTestContext thread/process

```
DeviceTestContext(..., process=False)
```

`False`: start device server in a thread (default)
`True`: start device server in a subprocess

Thread: can access device internals, but can segv
Subprocess: no internals, no segv

Best of both: use thread with multiprocess test runner (pytest --forked)

[Multi]DeviceTestContext device internals

```python
class MyDevice(Device):
    def init_device(self):
        super(MyDevice, self).init_device()
        self._attr_value = 0

    @attribute
    def attr(self):
        return self._attr_value

    @command
    def cmd(self):
        self.do_something()

    def do_something(self):
        print("something!")

```

```python

def test_single_device_no_internals():
    with DeviceTestContext(MyDevice) as proxy:
        assert proxy.attr == 0
        proxy.cmd()  # can't verify much

```

[Multi]DeviceTestContext device internals

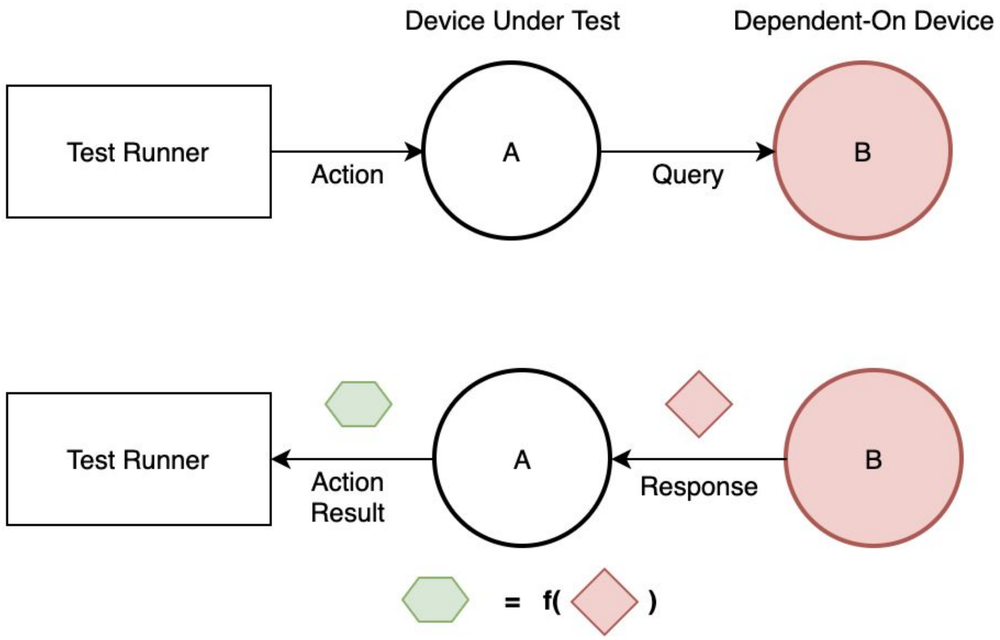
```python
class MyDevice(Device):
    def init_device(self):
        super(MyDevice, self).init_device()
        self._attr_value = 0

    @attribute
    def attr(self):
        return self._attr_value

    @command
    def cmd(self):
        self.do_something()

    def do_something(self):
        print("something!")

```

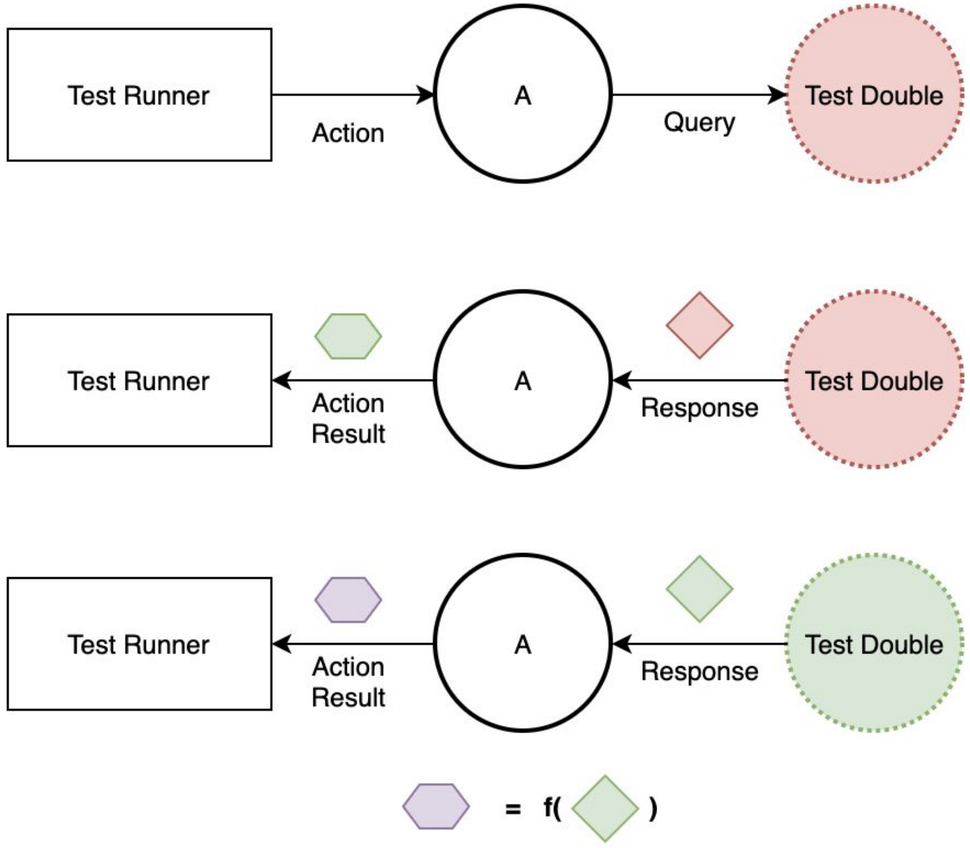
```python

def test_single_device_no_internals():
    with DeviceTestContext(MyDevice) as proxy:
        assert proxy.attr == 0
        proxy.cmd()  # can't verify much

```

```python
def test_single_device_access_internals_in_thread():

    class TestDevice(MyDevice):
        instances = weakref.WeakValueDictionary()

        def init_device(self):
            super(TestDevice, self).init_device()
            self.instances[self.get_name()] = self

    with DeviceTestContext(TestDevice) as proxy:
        instance = TestDevice.instances[proxy.name()]
        instance._attr_value = 123
        assert proxy.attr == 123
        instance.do_something = mock.Mock(side_effect=instance.do_something)
        proxy.cmd()
        instance.do_something.assert_called_once()
```

Mocking DeviceProxy

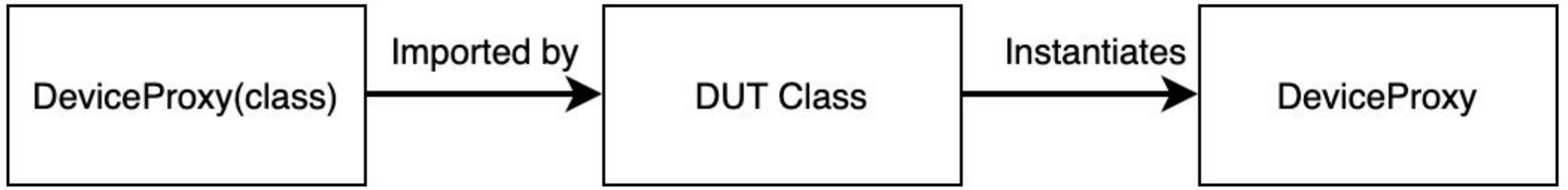
Mocking DeviceProxy - test doubles



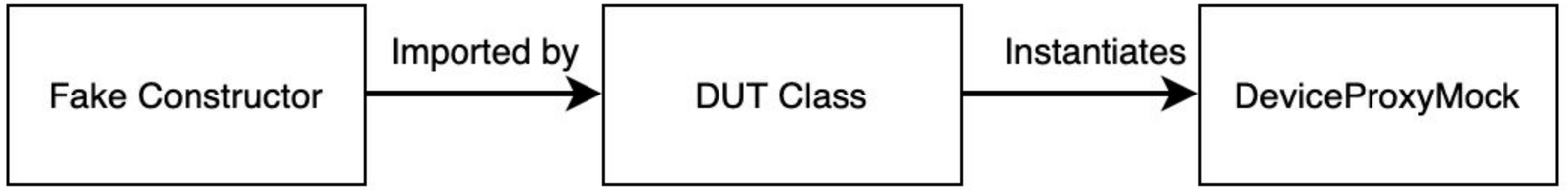


Mocking DeviceProxy

Original Implementation



Mocked Implementation



* Same approach can be used for other client access methods: *Group* and *AttributeProxy*

Pytest examples

Using Pytest fixtures with DeviceTestContext

```
def test_on():
    # TEST HARNESS SETUP
    context = DeviceTestContext(
        PowerSupply,
        properties={ ... }
    )
    context.start()
    power_supply_proxy = context.device

    # TEST
    assert power_supply_proxy.state() == DevState.OFF
    power_supply_proxy.on()
    assert power_supply_proxy.state() == DevState.ON

    # TEST HARNESS TEARDOWN
    context.stop()
```

Only three lines are actual test logic.

The other four are test setup / teardown.

These:

- obscure the test, and
- are likely to be reused in many tests.

Pytest fixtures primer

```
def test_on():
    # TEST HARNESS SETUP
    context = DeviceTestContext(
        PowerSupply,
        properties={ ... }
    )
    context.start()
    power_supply_proxy = context.device

    # TEST
    assert power_supply_proxy.state() == DevSt
    power_supply_proxy.on()
    assert power_supply_proxy.state() == DevSt

    # TEST HARNESS TEARDOWN
    context.stop()
```

Use a fixture to separate test setup / teardown from test logic

```
@pytest.fixture()
def power_supply():
    context = DeviceTestContext(
        PowerSupply,
        properties={ ... }
    )
    context.start()
    yield context.device
    context.stop()

def test_on(power_supply):
    assert power_supply.state() == DevState.OFF
    power_supply.on()
    assert power_supply.state() == DevState.ON
```

Fixtures can call fixtures

```python
# conftest.py
@pytest.fixture()
def device_under_test(device_info):
    context = DeviceTestContext(
        device_info["class"],
        properties=device_info["properties"]
    )
    context.start()
    yield context.device
    context.stop()
```

```python
@pytest.fixture()
def device_info():
    return {
        "class": PowerSupply,
        "properties": { ... },
    }
```

Fixtures can call fixtures, and override fixtures

```python
# conftest.py
@pytest.fixture()
def device_under_test(devic
    context = DeviceTestCon
        device_info["class"
        properties=device_i
    )
    context.start()
    yield context.device
    context.stop()
```

```python
# test_power_supply.py
class TestPowerSupply:

    @pytest.fixture()
    def device_info():
        return {
            "class": PowerSupply,
            "properties": { ... },
        }

    def test_on(device_under_test):
        assert device_under_test.state() == DevState.OFF
        device_under_test.on()
        assert device_under_test.state() == DevState.ON
```

Fixtures can call fixtures, and override fixtures

```python
# conftest.py
@pytest.fixture()
def device_under_test(devic
    context = DeviceTestCon
        device_info["class"
        properties=device_i
    )
    context.start()
    yield context.device
    context.stop()
```

```python
# test_power_supply.py
class TestPowerSupply:
```

```python
# test_controller.py
class TestController:

    @pytest.fixture()
    def device_info():
        return {
            "class": Controller,
            "properties": { ... },
        }

    def test_on(device_under_test):
        assert device_under_test.state() == DevState.OFF
        device_under_test.turn_on()
        assert device_under_test.state() == DevState.ON
```

Mocking tango.DeviceProxy: Why?

```python
# antenna_manager.py
class AntennaManager(Device):
    AntennaFQDN = device_property(dtype=str)

    def init_device(self):
        self.get_device_properties()
        self._antenna_proxy = tango.DeviceProxy(
            self.AntennaFQDN
        )

    @command(dtype_in=float)
    def SetAntennaGain(self, gain):
        self._antenna_proxy.gain = gain
```

If we mock out *tango.DeviceProxy* here…

Then we are writing to a mock here…

Mocking tango.DeviceProxy: a strawman

```python
@pytest.fixture()
def mock_device_proxies(mocker):
    mocker.patch("tango.DeviceProxy")

@pytest.fixture()
def device_under_test(device_info, mock_device_proxies):
    with DeviceTestContext(
        device_info["class"],
        properties=device_info["properties"]
    ) as device:
        yield device
```

We have mocked, but we can't get a handle on the mock.

Each call to *tango.DeviceProxy* will return a new mock, even for the same FQDN.

Mocking tango.DeviceProxy with a defaultdict

```python
@pytest.fixture()
def mock_device_proxies(mocker):
    mocks = defaultdict(mocker.Mock)
    mocker.patch(
        "tango.DeviceProxy",
        side_effect=lambda fqdn, *args, **kwargs: mocks[fqdn]
    )
    return mocks

@pytest.fixture()
def device_under_test(device_info, mock_device_proxies):
    with DeviceTestContext(
        device_info["class"],
        properties=device_info["properties"]
    ) as device:
        yield device
```

We get the same mock each time we call *DeviceProxy* with the same FQDN.

We have handles on all of our mocks.

Mocking tango.DeviceProxy with a defaultdict

```python
@pytest.fixture()
def mock_device_proxies(mocker):
    mocks = defaultdict(mocker.Mock)
    mocker.patch(
        "tango.Dev
        side_effe
    )
    return mocks

@pytest.fixture()
def device_under_
    with DeviceTe
        device_in
        propertie
    ) as device:
        yield dev
```

```python
# test_antenna_manager.py
class TestAntennaManager:
    @pytest.fixture()
    def device_info():
        return {
            "class": AntennaManager,
            "properties": {"AntennaFQDN": "test/antenna/1"},
        }

    def test_SetAntennaGain(device_under_test, mock_device_proxies):
        device_under_test.SetAntennaGain(1.1)
        assert mock_device_proxies["test/antenna/1"].gain == 1.1
```

We get the same mock each time we call *DeviceProxy* with

Mocking tango.DeviceProxy with a defaultdict

```python
@pytest.fixture()
def mock_device_proxies(mocker):
    mocks = defaultdict(mocker.Mock)
    mocker.patch(
        "tango.Devi
        side_effect
    )
    return mocks

@pytest.fixture()
def device_under_te
    with DeviceTest
        device_info
        properties=
    ) as device:
        yield devic
```

```python
# test_antenna_manager.py
class TestAntennaManager:
    @pytest.fixture()
    def device_info():
        return {
            "class": AntennaManager,
            "properties": {"AntennaFQDN": "test/antenna/1"},
        }

    def test_SetAntennaGain(device_under_test):
        device_under_test.SetAntennaGain(1.1)
        assert tango.DeviceProxy("test/antenna/1").gain == 1.1
```

We get the same mock each time we

Testing with a mocked tango.DeviceProxy

```python
@command(dtype_in=float)
def SetAntennaGain(self, gain):
    # self._antenna_proxy.gain = gain
    self._antenna_proxy.write_attribute("gain", gain)
```

```python
def test_SetAntennaGain(device_under_test):
    device_under_test.SetAntennaGain(1.1)
    # assert tango.DeviceProxy("test/antenna/1").gain == 1.1
    mock_antenna = tango.DeviceProxy("test/antenna/1")
    mock_antenna.write_attribute.assert_called_once_with("gain", 1.1)
```

Customizing mock behaviour: the need

```
@command(dtype_in=float, dtype_out=bool)
def SetAntennaGain(self, gain):
    result = self.antenna_proxy.SetGain(gain):
    return result==ReturnCode.SUCCESS
```

Antenna.SetGain can now return *SUCCESS or FAILURE*.

Problem: *self.antenna_proxy* is a mock. If you call *SetGain()* on a mock, it returns another mock. So *SetAntennaGain()* will always return False. How can we test it?

Solution: We need to set the expected behaviour of the mock in advance of the test.

Customizing mock behaviour: the solution

```python
# conftest.py
@pytest.fixture()
def initial_mocks():
    return {}

@pytest.fixture()
def mock_factory(mocker):
    return mocker.Mock

@pytest.fixture()
def mock_device_proxies(mocker, mock_factory, initial_mocks):
    device_proxy_mocks = defaultdict(mock_factory, initial_mocks)
    mocker.patch(
        "tango.DeviceProxy",
        side_effect=lambda fqdn: device_proxy_mocks[fqdn]
    )
    return device_proxy_mocks
```

Customizing mock behaviour: an example

```python
# conftest.py
@pytest.fixture()
def initial_mocks():
    return {}

@pytest.fixture()
def mock_factory(mocker):
    return mocker.Mock

@pytest.fixture()
def mock_device_proxies(mocke
    device_proxy_mocks = defa
    mocker.patch(
        "tango.DeviceProxy",
        side_effect=lambda fq
    )
    return device_proxy_mocks
```

```python
class TestAntennaManager:
    @pytest.fixture()
    def device_info():
        return {
            "class": AntennaManager,
            "properties": {"antenna_fqdn": "test/antenna/1"},
        }

    @pytest.fixture()
    def initial_mocks(mocker):
        mock_antenna = mocker.Mock()
        mock_antenna.SetGain.return_value = ResultCode.SUCCESS
        return {"test/antenna/1": mock_antenna}

    def test_SetAntennaGain(device_under_test):
        assert device_under_test.SetAntennaGain(1.1)
        mock_antenna = tango.DeviceProxy("test/antenna/1")
        mock_antenna.SetGain.assert_called_once_with(1.1)
```

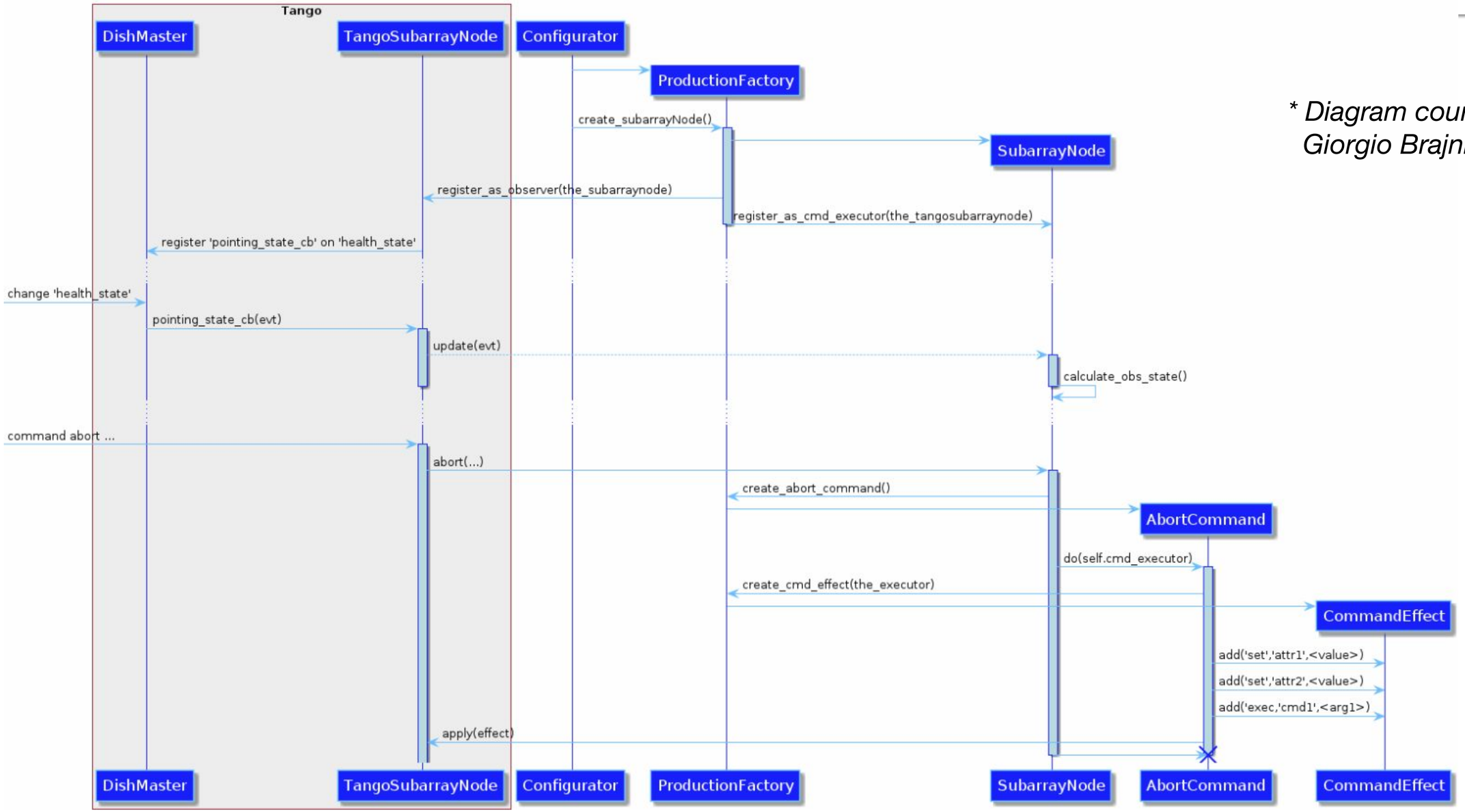
MultiDeviceTestContext workaround fixture

```python
@pytest.fixture()
def device_context(mocker, devices_info):
    def _get_open_port():
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.bind(("", 0))
        s.listen(1)
        port = s.getsockname()[1]
        s.close()
        return port

    HOST = get_host_ip()
    PORT = _get_open_port()

    _DeviceProxy = tango.DeviceProxy
    mocker.patch(
        "tango.DeviceProxy",
        wraps=lambda fqdn, *args, **kwargs: _DeviceProxy(
            f"tango://{HOST}:{PORT}/{fqdn}#dbase=no", *args, **kwargs
        ),
    )

    with MultiDeviceTestContext(
        devices_info, process=True, host=HOST, port=PORT
    ) as context:
        yield context
```

- Fixture for *MultiDeviceTestContext*
- Uses *devices_info* fixture
- Workaround for short-address resolution issue
  - Needed in PyTango 9.3.2
  - Fix coming

Wrapping up

Future? Move logic outside Tango domain



** Diagram courtesy of Giorgio Brajnik*

SQUARE KILOMETRE ARRAY

Exploring the Universe with the world's largest radio telescope

Thanks!

Questions?

https://pytango.readthedocs.io/en/latest/testing.html