MONK Testframework Documentation

Release 0.8.1

DResearch Fahrzeugelektronik GmbH

October 22, 2015

Contents

Contents:

CHAPTER 1

Intro

Using MONK you can write tests like you would write unit tests, just that they are able to interact with your embedded system.

Let's look at an example. In the following example we have an embedded system with a serial terminal and a network interface. We want to write a test, which checks whether the network interface receives correct information via dhcp.

The test case written with nosetests:

```
import nose.tools as nt

import monk_tf.conn as mc
import monk_tf.dev as md

def test_dhcp():
    """ check whether dhcp is implemented correctly
    """
    # setup
    device = md.Device(mc.SerialConn('/dev/ttyUSB1','root','sosecure'))
    # exercise
    device.cmd('dhcpc -i eth0')
    # verify
    ifconfig_out = device.cmd('ifconfig eth0')
    nt.ok_('192.168.2.100' in ifconfig_out)
```

Even for non python programmers it should be not hard to guess, that this test will connect to a serial interface on `/dev/ttyUSB1`, send the shell command `dhcpc` to get a new IP adress for the `eth0` interface, and in the end it checks whether the received IP address that the tester would expect. No need to worry about connection handling, login and session handling.

For more information see the API Docs.

CHAPTER 2

monk_tf package

2.1 Submodules

2.2 monk_tf.conn module

This module implements connection handling. Using the classes from this module you can connect directly to a *target device* via serial or ssh. Example:

```
import monk_tf.conn as mc
# create a serial connection
serial=mc.SerialConn(name="ser1", port="/dev/ttyUSB3", user="tester", pw="test")
# create a ssh connection
ssh=mc.SshConn(name="ssh1", host="192.168.2.123", user="tester", pw="test")
# send a command
print serial.cmd("ls -al")
[...]
# send a command
ssh.cmd("ls -al")
[...]
```

exception monk_tf.conn.**AConnectionException**
Bases: exceptions.Exception

Base class for Exceptions from this module

exception monk_tf.conn.**BccException**
Bases: monk_tf.conn.AConnectionException

is raised to explain some BCC behaviour

exception monk_tf.conn.**CantCreateConn**
Bases: monk_tf.conn.AConnectionException

is raised when even several attempt were not able to create a connection.

class monk_tf.conn.**Capture**(*handle=None*)
Bases: object

a helper class

that supports ConnectionBase in handling Terminal special chars.

draw(*ch*, ***flags*)

linefeed()

class monk_tf.conn.**ConnectionBase**(*name*, *default_timeout=None*, *first_prompt_timeout=None*)
Bases: object

is the base class for all connections.

Don't instantiate this class directly.

This class implements the behaviour of cmd() interactions, makes sure you get logged in etc.

Extending this class requires to implement _get_exp() and _login().

close()
close the connection and get rid of the inner objects

cmd(*msg*, *timeout=None*, *expect=None*, *do_retcode=True*)
send a shell command and retreive its output.

Parameters

- **msg** – the shell command
- **timeout** – how long we wait for expect; if None is set to self.default_timeout
- **expect** – a list of things to expect, e.g. output strings
- **do_retcode** – boolean which says whether or not a returncode should be retreived.

exp
the pexpect object - Don't bother with this if you don't know what it means already. Really!

expect_prompt(*timeout=None*)
enter + look in the output for what is currently set as self.prompt

log(*msg*)
wrapper for simpler debug logging

name
the name of this connection and its corresponding logger

wait_for_prompt(*timeout=-1*)
this method continuously retries to get a working connection

(by means of self.expect_prompt()) and raises an exception otherwise

Parameters **timeout** – how long we retry

exception monk_tf.conn.**NoBCCException**
Bases: monk_tf.conn.BccException

is raised when the BCC class does not find the drbcc tool needed for execution.

exception monk_tf.conn.**NoRetcodeException**
Bases: monk_tf.conn.AConnectionException

is raised when the output doesn't contain a retcode for unknown reasons.

exception monk_tf.conn.**OutputParseException**
Bases: monk_tf.conn.AConnectionException

is raised when cmd output cannot be parsed to utf8 for further processing

class monk_tf.conn.**SerialConn**(*name*, *port*, *user*, *pw*, *prompt='r?n?[^n]*#'*, *default_timeout=None*, *first_prompt_timeout=None*)
Bases: monk_tf.conn.ConnectionBase

implements a serial connection.

class monk_tf.conn.**SshConn**(*name*, *host*, *user*, *pw*, *prompt=None*, *default_timeout=None*, *force_password=True*, *first_prompt_timeout=None*, *login_timeout=10*)

Bases: monk_tf.conn.ConnectionBase

implements an ssh connection.

close()

expect_prompt(*timeout=None*)

prompt

exception monk_tf.conn.**TimeoutException**

Bases: monk_tf.conn.AConnectionException

is raised if retrying something was not successful until its timeout

class monk_tf.conn.**pxsshWorkaround**(*timeout=30*, *maxread=2000*, *searchwindowsize=None*, *logfile=None*, *cwd=None*, *env=None*, *echo=True*)

Bases: pexpect.pxssh.pxssh

just to add that echo=False

2.3 monk_tf.dev module

This module implements device handling. Using the classes from this module you can abstract a complete *target device* in a single object. On instantiation you give it some connections and then (theoretically) let the device handle the rest.

Example:

```
import monk_tf.dev as md
import monk_tf.conn as mc
# create a device with a ssh connection and a serial connection
d=md.Device(
    mc.SshConn('192.168.2.100', 'tester', 'secret'),
    mc.SerialConn('/dev/ttyUSB2', 'root', 'muchmoresecret'),
)
# send a command (the same way as with connections)
print d.cmd('ls -al')
[...]
```

exception monk_tf.dev.**ADeviceException**

Bases: exceptions.Exception

Base class for exceptions of the device layer.

exception monk_tf.dev.**CantHandleException**

Bases: monk_tf.dev.ADeviceException

is raised when a request cannot be handled by the connections of a Device.

class monk_tf.dev.**Device**(**args*, ***kwargs*)

Bases: object

is the API abstraction of a *target device*.

close_all()

cmd(*msg*, *expect=None*, *timeout=30*, *login_timeout=None*, *do_retcode=True*)

Send a *shell command* to the *target device*.

Parameters

- **msg** – the *shell command*.
- **expect** – if you don't expect a prompt in the end but something else, you can add a regex here.
- **timeout** – when command should return without finding what it's looking for in the output. Will raise a **:py:exception:'pexpect.Timeout'** Exception.
- **do_retcode** – should this command retreive a returncode

Returns the standard output of the *shell command*.

get_conn(*which*)

log(*msg*)
sends a debug-level message to the logger

This method is used so often, that a smaller version of it is quite comfortable.

name

class monk_tf.dev.**Hydra**(**args*, ***kwargs*)
Bases: monk_tf.dev.Device

is the device type of DResearch Fahrzeugelektronik GmbH.

current_fw_version
the current version of the installed firmware

has_newest_firmware
check whether the installed firmware is the newest on jenkins

is_updated
check whether the device is already updated.

Currently it is implementd with `dev.Hydra.has_newest_firmware()`.

latest_build
get the latest build ID from jenkins

reset_config()
reset the HydraIP configuration on the device

update(*link=None*, *force=None*)
update the device to current build from Jenkins.

class monk_tf.dev.**PromptReplacement**
Bases: object

should be replaced by each connection's own prompt.

classmethod replace(*c*, *expect*)
this is an awful workaround...

exception monk_tf.dev.**UpdateFailedException**
Bases: monk_tf.dev.ADeviceException

is raised if an update didn't get finished or was rolled back.

exception monk_tf.dev.**WrongNameException**
Bases: monk_tf.dev.ADeviceException

is raised when no connection with a given name could be found.

2.4 monk_tf.fixture module

Instead of creating `Device` and `AConnection` objects by yourself, you can also choose to put corresponding data in a separate file and let this layer handle the object concstruction and destruction for you. Doing this will probably make your test code look more clean, keep the number of places where you need to change something as small as possible, and lets you reuse data that you already have described.

A hello world test with it looks like this:

```
import nose
from monk_tf import fixture

def test_hello():
    ''' say hello
    '''
    # set up
    h = fixture.Fixture('target_device.cfg')
    expected_out = "hello"
    # execute
    out = h.devs[0].cmd('echo "hello"')
    # assert
    nose.tools.eq_(expected_out, out)
    # tear down
    h.tear_down()
```

When using this layer setting up a device only takes one line of code. The rest of the information is in the `target_device.cfg` file. *MONK* currently comes with one text format parser predefined, which is the `XiniParser`. `Xini` is short for *extended INI*. You may, however, use any data format you want, if you extend the `AParser` class accordingly.

An example `Xini` data file might look like this:

```
[device1]
    type=Device
    [[serial1]]
        type=SerialConnection
        port=/dev/ttyUSB1
        user=example
        password=secret
```

As you can see it looks like an *INI* file. There are sections, consisting of a title enclosed in squared brackets (`[]`) and lists of properties, consisting of key-value pairs separated by equality signs (=). The unusual part is that the section *serial1* is surrounded by two pairs of squared brackets (`[]`). This is the specialty of this format indicating that *serial1* is a subsection of *device1* and therefore is a nested section. This nesting can be done unlimited, by surrounding a section with more and more pairs of squared brackets (`[]`) according to the level of nesting intended. In this example *serial1* belongs to *device1* and the types indicate the corresponding *MONK* object to be created.

2.4.1 Classes

exception `monk_tf.fixture.`**`AFixtureException`**

Bases: `exceptions.Exception`

Base class for exceptions of the fixture layer.

If you want to make sure that you catch all exceptions that are related to this layer, you should catch *AFixture-Exceptions*. This also means that if you extend this list of exceptions you should inherit from this exception and not from `Exception`.

exception monk_tf.fixture.**AParseException**
Bases: monk_tf.fixture.AFixtureException

Base class for exceptions concerning parsing errors.

exception monk_tf.fixture.**CantHandleException**
Bases: monk_tf.fixture.AFixtureException

if none of the devices is able to handle a cmd_any() call

exception monk_tf.fixture.**CantParseException**
Bases: monk_tf.fixture.AFixtureException

is raised when a Fixture cannot parse a given file.

class monk_tf.fixture.**Fixture**(*call_location*, *name=None*, *classes=None*, *lookfordbgsrc=True*, *filename='fixture.cfg'*, *auto_search=True*)
Bases: object

Creates *MONK* objects based on dictionary like objects.

This is the class that provides the fundamental feature of this layer. It reads data files by trying to parse them via its list of known parsers and if it succeeds, it creates *MONK* objects based on the configuration given by the data file. Most likely these objects are one or more Device objects that have at least one AConnection object each. If more than one *fixture file* is read containing the same name on the highest level, then the latest data gets used. This does not work on lower levels of nesting, though. If you attempt to overwrite lower levels of nesting, what actually happens is that the highest layer gets overwritten and you lose the data that was stored in the older objects. This is simply how set.update() works.

One source of data (either a file name or a child class of AParser) can be given to an object of this class by its constructer, others can be added afterwards with the read() method. An example looks like this:

```
import monk_tf.fixture as mf

fixture = mf.Fixture('/etc/monk_tf/default_devices.cfg')
        .read('~/.monk/default_devices.cfg')
        # can also be a parser object
        .read(XiniParser('~/testsuite12345/suite_devices.cfg'))
```

cmd_all(*msg*, *expect=None*, *timeout=30*, *login_timeout=None*)

cmd_any(*msg*, *expect=None*, *timeout=30*, *login_timeout=None*)

cmd_first(*msg*, *expect=None*, *timeout=30*, *login_timeout=None*)
call cmd() from first Device

get_dev(*which*)

log(*msg*)

name

read(*source*)
Read more data, either as a file name or as a parser.

Parameters **source** – the data source; either a file name or a AParser child class instance.

Returns self

reset_config_all()

tear_down()
Can be used for explicit destruction of managed objects.

This should be called in every *test case* as the last step.

exception monk_tf.fixture.**NoDeviceException**
Bases: monk_tf.fixture.AFixtureException

is raised when a **:py:clas:‘~monk_tf.fixture.Fixture‘** requires a device but has none.

exception monk_tf.fixture.**NoPropsException**
Bases: monk_tf.fixture.AFixtureException

is raised when

exception monk_tf.fixture.**WrongNameException**
Bases: monk_tf.fixture.AFixtureException

is raised when no devs with a given name could be found.

2.5 Module contents

This is the package overview of *MONK*. If anything is unclear, you might have a look into *chap-intro*.

The following texts describe the three layers that were explained in *intro-layers*:

m

R

S

T

U

W