MONK Testframework Documentation

Release 0.17.6

DResearch Fahrzeugelektronik GmbH

December 10, 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 monk_tf.fixture module

Instead of creating `Device` and `ConnectionBase` 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 enables you to reuse data that you already have described.

A hello world test with a fixture looks like this:

```
import nose
from monk_tf import fixture

def test_hello():
    ''' say hello
    '''
    with fixture.Fixture(__file__) as (fix, dev):
        # set up
        expected_out = "hello"
        # execute
        retcode, out = dev.cmd('echo "hello"')
        # assert
        nose.tools.eq_(expected_out, out)
        # tear down - automatically done by Fixture
```

Everything is handled in a context that manages the fixture and your *target device*. The Fixture is automatically looking for `fixture.cfg` in the current directory or its parents. The `fixture.cfg` contains the data that is necessary to build your test fixture. This includes connection data like IP, user name, and password. MONK separates this data from the code, that the tests can be executed on different *target devices* without changing the tests themselves. The format of these files is quite close to ini files, just with an added layer of depth, enabling sections to contain other sections if the inner section is surrounded by an additional set of square brackets (`[]`).

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.

2.1.1 Classes

exception monk_tf.fixture.**AFixtureException**
Bases: monk_tf.general_purpose.MonkException

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

is raised when a Fixture cannot parse a given file.

class monk_tf.fixture.**FileHandler**(*name*, *sink*, *target*, *format*, *level*)
Bases: monk_tf.fixture.LogHandler

pre_register()

class monk_tf.fixture.**Fixture**(*call_location*, *name=None*, *fixture_locations=None*, *parsers=None*)
Bases: monk_tf.general_purpose.MonkObject

Creates *MONK* objects based on dictionary like objects.

Use this class if you want to seperate the details of your MONK objects from your code. Also use it if you want to write tests with it, as described above.

default_fixturelocations()
this is preferred over a list/dict

because some paths need to be set dynamically!

default_parsers()

firstdev

parse_conns(*name*, *sectype*, *section*)

parse_device(*name*, *sectype*, *section*)

parse_filehandler(*name*, *sectype*, *section*)

parse_logging(*name*, *sectype*, *section*)

parse_serialconn(*name*, *sectype*, *section*)

parse_sshconn(*name*, *sectype*, *section*)

parse_streamhandler(*name*, *sectype*, *section*)

parsers

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

Parameters **sources** – a iterable of data sources; each is either a file name or a `AParser` child class instance.

Returns self

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

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

update(***kwargs*)
update the externally manageable data of this fixture object

class monk_tf.fixture.**LogHandler**(*name*, *sink*, *target*, *format*, *level*)
Bases: monk_tf.general_purpose.MonkObject

config_subs(*txt*, *subs=None*)
replace the strings in the config that we have reasonable values for

post_register()

pre_register()

register()

class monk_tf.fixture.**LogManager**(***config*)
Bases: monk_tf.general_purpose.MonkObject

managing configuration and setup of logging mechanics

Might strongly interact with your nose config or similar.

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.**NoDevicesDefinedException**
Bases: monk_tf.fixture.AFixtureException

is raised when we found out that there are no devices.

Currently it makes no sense to use a fixture without devices.

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

If the use_devs attribute is not set this is raised

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

is raised when

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

If no name can be derived from parsing a section

class monk_tf.fixture.**StreamHandler**(*name*, *sink*, *target*, *format*, *level*)
Bases: monk_tf.fixture.LogHandler

pre_register()

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

Handler Type was not recognized

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

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

2.2 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)
return_code, output = d.cmd('ls -al')
print output
[...]
```

exception monk_tf.dev.**ADeviceException**
Bases: monk_tf.general_purpose.MonkException

Base class for exceptions of the device layer.

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

is the API abstraction of a *target device*.

close_all()
loop through all connections calling close().

cmd(*msg*, *expect=None*, *timeout=30*, *login_timeout=None*, *do_retcode=True*, *fallback_conn=None*, *conn=None*)
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
- **fallback_conn** – use this connection to reboot command.
- **conn** – the name of the connection that should be used for this command.

Returns *returncode*, *standard output* of the shell command

cp(*src_path*, *trgt_path*)
send files via scp to target device

Parameters

- **src_path** – the path to the file on the host machine
- **trgt_path** – the path of the file on the target machine

eval_cmd(*msg*, *timeout=None*, *expect=None*, *do_retcode=True*)
apply the same method from the first connection

firstconn

wait_for(*msg*, *retries=3*, *sleep=5*, *timeout=10*)
apply the same method from the first connection

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.

2.3 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: monk_tf.general_purpose.MonkException

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.**CantCreateConnException**
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()

tab()

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

is raised in an eval_cmd() request if the returncode was != 0. The returncode can be parsed from the Exception's message.

class monk_tf.conn.**ConnectionBase**(*name*, *target*, *user*, *pw*, *default_timeout=None*, *first_prompt_timeout=None*)
Bases: monk_tf.general_purpose.MonkObject

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.

eval_cmd(*msg*, *timeout=None*, *expect=None*, *do_retcode=True*)
evaluate cmd's returncode and therefore don't return it

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

wait_for(*msg*, *retries=3*, *sleep=5*, *timeout=20*)
repeatedly send shell command until output is found

Parameters

- **msg** – the shell command that should be executed
- **retries(3)** – how often should we try it. should be at least 1, otherwise the loop is not executed.
- **sleep(5)** – the time to wait between requests
- **timeout(20)** – the timeout used for every cmd() request

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

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

when trying something repeatedly didn't succeed but a more specific reason is not available

class monk_tf.conn.**SerialConn**(*name*, *port*, *user*, *pw*, *prompt='r?n?[^n]*#'*, *default_timeout=None*, *first_prompt_timeout=None*, *speed=115200*)
Bases: monk_tf.conn.ConnectionBase

implements a serial connection.

port

prompt

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()

cp(*src_path*, *trgt_path*, *retry=5*, *sleep=5*, *timeout=10*)
send files via scp to target device

Parameters

- **src_path** – the path to the file on the host machine
- **trgt_path** – the path of the file on the target machine

expect_prompt(*timeout=None*)

host

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

m

R

S

T

U

W