python-moos: Python for MOOS:V10 Communications

Paul Newman, University of Oxford

....ten years on

10.0.2

Contents

1 Getting Started - Acquiring and Building MOOS

If you already have MOOS::V10 installed and built you can skip this section.

1.1 Before you start you will need...

- a working compiler like `gcc` or `clang`
- `CMake` installed
- `git` installed (well actually this is optional as you can download the source code as .zip file and we won't make much use of git in this tutorial)
- boost (http://www.boost.org) installed. Why you might ask? Well python-moos uses the boost python suite to facilitate binding MOOS to python. It is dead good.

1.2 Downloading and Building MOOS V10

We shall begin where we should and check out a version of MOOS-V10 from a git repos. We will follow good practice and do an out of place build - the source code will go in "src" and we will build in "build". We will also, after fetching the source switch to the "devel" branch because here we are living on the edge [1].

```
pmn@mac:~$ mkdir core-moos-v10
pmn@mac:~$ cd core-moos-v10
pmn@mac:~$ git clone https://github.com/themoos/core-moos.git src
pmn@mac:~$ mkdir build
pmn@mac:~$ cd build
pmn@mac:~$ ccmake ../src
```

At this point you should, after hitting 'c' a couple of times be presented with a CMake screen that looks like that shown in Figure 1.1 (note some of the entries are platform dependent so don't worry if what you see is not identical to this).

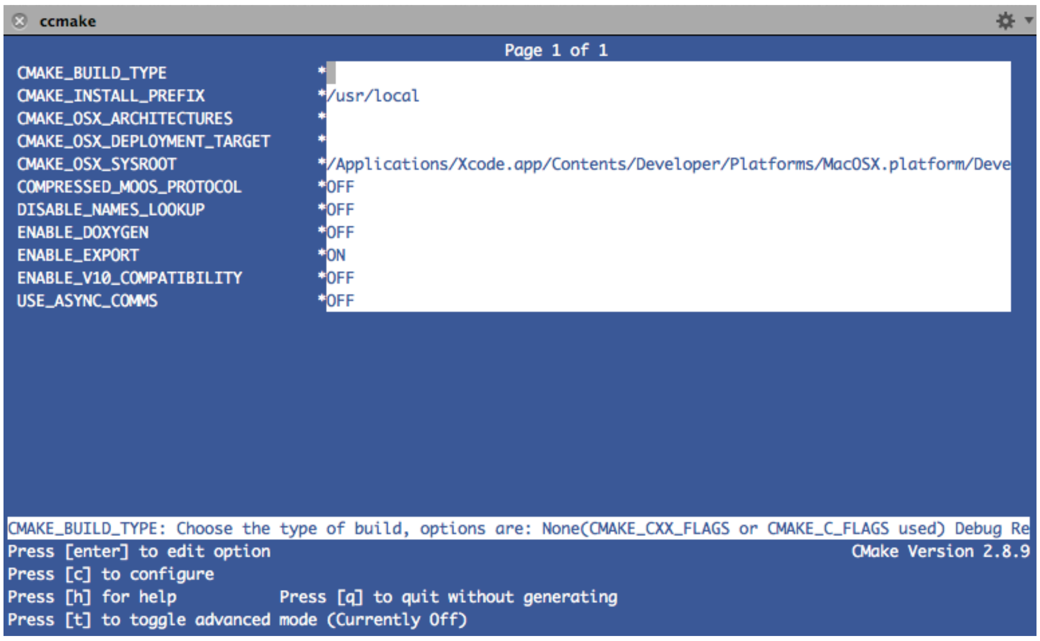
You are are now in a position to build the MOOS. So press 'c' until 'g' appears, then press 'g' and you are good to go. Then at the terminal prompt type 'make' to build the project. Two directories should have been created **bin** and **lib.** In lib you will see `libMOOS.a` and in `bin` you will find the newly created `MOOSDB` and some other fabulous tools like UMM, MTM and MQOS. Nice job.

2 Acquiring python-moos

PYTHON-MOOS is hosted on github (at time of writing) but where ever you get it from the build path looks somethign like this.

```
pmn@mac:~$ mkdir python-moos
pmn@mac:~$ cd python-moos
pmn@mac:~$ git clone https://github.com/themoos/python-moos.git src
pmn@mac:~$ mkdir build
pmn@mac:~$ cd build
pmn@mac:~$ ccmake ../src
```

[1] if you want to know what branches are available type `git branch`



```
ccmake
                                     Page 1 of 1
CMAKE_BUILD_TYPE                *
CMAKE_INSTALL_PREFIX            */usr/local
CMAKE_OSX_ARCHITECTURES         *
CMAKE_OSX_DEPLOYMENT_TARGET     *
CMAKE_OSX_SYSROOT               */Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Deve
COMPRESSED_MOOS_PROTOCOL        *OFF
DISABLE_NAMES_LOOKUP            *OFF
ENABLE_DOXYGEN                  *OFF
ENABLE_EXPORT                   *ON
ENABLE_V10_COMPATIBILITY        *OFF
USE_ASYNC_COMMS                 *OFF

CMAKE_BUILD_TYPE: Choose the type of build, options are: None(CMAKE_CXX_FLAGS or CMAKE_C_FLAGS used) Debug Re
Press [enter] to edit option                                              CMake Version 2.8.9
Press [c] to configure
Press [h] for help           Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)
```

Fig. 1.1: The default build screen for MOOS V10.

At the end of this process you should see in the build director some example python scripts and the python-moos library (most likely called pymoos.so). This object is importable into python. The next section will show how.

3 Getting help

You can see quickly seewhat goodies are offered by a class by calling for example `dir(pymoos.comms)`, `dir(pymoos.comms)` or dir(pymoos.moos_msg). For example:

```
pmn@mac:~$ python
pmn@mac:~$>> import pymoos
pmn@mac:~$>> dir(pymoos.comms)
pmn@mac:~$>>....'add_active_queue', '↩
    add_message_route_to_active_queue', 'close', '↩
    do_local_time_correction', 'enable_comms_status_monitoring', '↩
    fetch', 'get_client_comms_status', 'get_client_comms_statuses', '↩
    get_comms_control_timewarp_scale_factor', 'get_community_name', '↩
    get_description', 'get_moos_name',...
```

Alternatively you can access help via python's 'help' interface which will show you all the methods supported and the c++ signatures they wrap.

```
pmn@mac:~$ python
pmn@mac:~$>> import pymoos
pmn@mac:~$>> help(pymoos.moos_msg)
|
|   binary_data_size(...)
|      binary_data_size( (moos_msg)arg1) -> int :
|
|          C++ signature :
|              unsigned int binary_data_size(CMOOSMsg {lvalue})
|
|  double(...)
|      double( (moos_msg)arg1) -> float :
|
|          C++ signature :
|              double double(CMOOSMsg {lvalue})
|
|  double_aux(...)
|      double_aux( (moos_msg)arg1) -> float :
|
|          C++ signature :
|              double double_aux(CMOOSMsg {lvalue})
|
|  is_binary(...)
|      is_binary( (moos_msg)arg1) -> bool :
```

4 A simple python-moos example

Lets start with the simplest possible MOOS interface - a simple program that connects to a MOOS community occasionally sends a message and also subscribes to a message - in this case the same one).

Listing 1: A simple example using python-moos

```
import pymoos
import time

# A super simple example:
# a) we make a comms object, and run it
# b) we enter a forever loop in which we both send notifications
#    and check and print mail (with fetch)
# c) the map(lambda,,,) bit simply applies trace() to every message to print
#    a summary of the message.

comms = pymoos.comms()

def on_connect():
    return comms.register('simple_var',0)

def main():
    # my code here
    comms.set_on_connect_callback(on_connect);
    comms.run('localhost',9000,'pymoos')

    while True:
        time.sleep(1)
        comms.notify('simple_var','hello world',pymoos.time());
        map(lambda msg: msg.trace(), comms.fetch())

if __name__ == "__main__":
    main()
```

pymoos::comms

Looking at this example, first we import PYMOOS and make ourselves a `pymoos.comms` object. This is the object that provides MOOS comms functionality.

Then we define a function, `on_connect` which will be called when we connect to a MOOSDB and in which, as all good moos citizens do, we regsiter our interest in messages. In this case we want to know about any changes made to `simple_var`.

Then we define a main which begins with installing the connection callback, running the the comms and entering a simple a `while`(1) which sleeps for a second, sends a message and then polls the comms object to see if any messages have arrived (with `comms.fetch()`) and, if they have, prints them all out using some fancy python “map” functionality and the `pymoos.moos_msg.trace()` method. Note that `fetch()` returns a python list of `moos_msgs`.

So we have now seen a trivial example using polling to retrieve any incoming messages. Before we up our game, it is worth quickly looking at `pymoos.moos_msg.`

pymoos::moos_msg

Where in c++ land we have `CMOOSMsg`, in python land we have `pymoos.moos_msg.` Many of the methods of the former are exposed in the latter. Again you can see them by calling `dir(pymoos.moos_msg).`

5 A python-moos example with a message callback

In the next example, we install a messaging callback(we call it `m()` here). Everytime mail arrives we are given the opportunity to grab it using fetch() and print it.

Listing 2: A msg callback example using python-moos

```
import pymoos
import time

#another simple example - here we install a callback which
#fires every time mail arrives. We need to call fetch to
#retrieve it though. We send mail in simple forever loop
#We also subscribe to the same message..

comms = pymoos.comms()

def c():
    return comms.register('simple_var',0)

def m():
    map(lambda msg: msg.trace(), comms.fetch() )

def main():

    comms.set_on_connect_callback(c);
    comms.set_on_mail_callback(m);
    comms.run('localhost',9000,'pymoos')

    while True:
        time.sleep(1)
        comms.notify('simple_var','a string',pymoos.time());

if __name__=="__main__":
    main()
```

But do be careful....`m()` is being called in a different thread to the `while()` loop in main. The python interpreter is appropriately locked though so don't panic too much.

6 A python-moos example with active queues

Active queues are a mechanism to route messages to dedicated handling threads or 'queues'. As messages are received they can be distributed according to preferences configured by a simple API, to activie queues. Each named queue processes messages sequentially and applies a user specified callback to each message.

Listing 3: A msg callback example using python-moos

```
import pymoos
import time


#simple example which uses an active queue to handle received messages
#you can send any number of messages to any number of active queues
comms = pymoos.comms()

def c():
    comms.register('simple_var',0)
    return True

def queue_callback(msg):
    msg.trace();
    return True;

def main():

    comms.set_on_connect_callback(c);
    comms.add_active_queue('the_queue',queue_callback)
    comms.add_message_route_to_active_queue('the_queue','simple_var')
    comms.run('localhost',9000,'pymoos')

    while True:
        time.sleep(1)
        comms.notify('simple_var','hello world',pymoos.time());

if __name__=="__main__":
        main()
```

The important lines are

```
comms.add_active_queue('the_queue' , queue_callback )
comms.add_message_route_to_active_queue ('the_queue' , 'binary_var' )
```

The first line creates an active queue called 'the_queue' and attaches the function queue_callback to it. The second line instructs the comms object to place a copy of any message called 'binary_var' into this queue so it will end up being processed by queue_callback.

7 Dealing with binary data

MOOS is at home with binary data of course. The next example makes this clear by building on the previous active queue example. You can see if someone sent you binary data via the moos_msg.is_binary() method. And you can send binary data with the comms.binary_notify() method

Listing 4: sending and receiving binary data using python-moos

```
import pymoos
import time


#simple example which uses an active queue to handle received messages
#you can send any number of messages to any number of active queues
#here we send binary data just to show we can
comms = pymoos.comms()

def c():
    comms.register('binary_var',0)
    return True

def queue_callback(msg):
    if msg.is_binary():
        b = bytearray(msg.binary_data())
        print 'received ', len(b), 'bytes'
    return True;

def main():
    comms.set_on_connect_callback(c);
    comms.add_active_queue('the_queue',queue_callback)
    comms.add_message_route_to_active_queue('the_queue','binary_var')
    comms.run('localhost',9000,'pymoos')

    while True:
        time.sleep(1)
        x = bytearray( [0, 3, 0x15, 2, 6, 0xAA] )
        comms.notify_binary('binary_var',bytes(x),pymoos.time());


if __name__=="__main__":
    main()
```