Lab #3 - Introduction to MOOS

2.S998 Unmanned Marine Vehicle Autonomy, Sensing and Communications

Contents

1 Overview and Objectives

This lab will introduce MOOS to new users. It assumes nothing regarding MOOS background. The goals of this lab are to (a) understand the publish-subscribe architecture, (b) get comfortable launching and interacting with the MOOSDB, (c) understand how to generated scripted interactions with the MOOSDB, (d) understand how the logger operates and basic tools for examining log files, and (e) experiment with MOOS connections between machines.

1.1 Preliminaries

This lab does assume that you have a working MOOS-IvP tree checked out and installed on your computer. To verify this make sure that the following executables are built and findable in your shell path:

```
$ which MOOSDB
/Users/you/moos-ivp/MOOS/MOOSBin/MOOSDB
$ which uTimerScript
/Users/you/moos-ivp/bin/uTimerScript
$ which mykill
/Users/you/moos-ivp/scripts/mykill
```

1.2 MOOS vs. MOOS-IvP

What is the relationship between MOOS and MOOS-IvP? MOOS-IvP is a superset of MOOS. The additional components include another architecture, the IvP Helm behavior-based architecture, and several additional MOOS applications. This is the nested repository concept depicted in Figure 1.

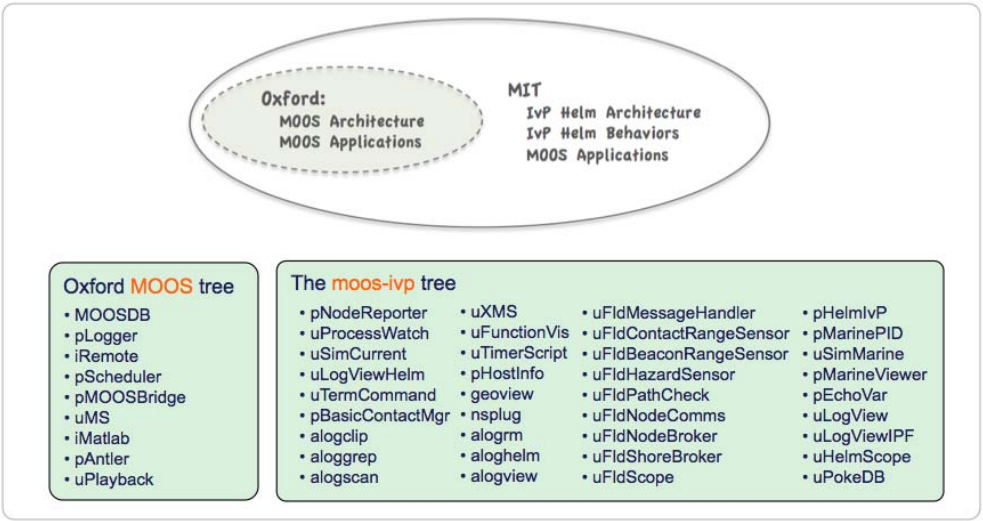

Figure 1: **Nested Repositories:** The MOOS-IvP tree contains the Oxford MOOS tree and additional modules from MIT including the Helm architecture, Helm behaviors and further MOOS applications.

1.3 More MOOS / MOOS-IvP Resources

We will only just touch the MOOS basics today. A few further resources are worth mentioning for following up this lab with your own exploration.

- See the slides from the today's class which give a bit more background into MOOS and MOOS-IvP related to marine robotics.
- The Oxford MOOS documentation.
  http://www.moos-ivp.org/oxdocs.html
- The moos-ivp.org website documentation.
  http://www.moos-ivp.org

1.4 The MOOS Architecture

The main idea explored today is that MOOS is a publish-subscribe architecture. A single MOOSDB serves multiple MOOS applications by essentially handling the mail published and subscribed for by each. A MOOS *community* is a collection of applications connected to a single MOOSDB.

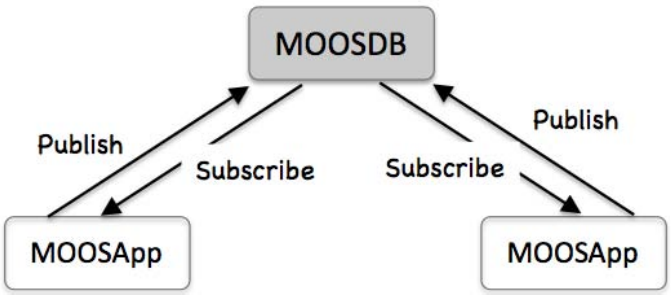

Figure 2: **The MOOS Architecture:** MOOS is a publish-subscribe architecture. The MOOSDB servers a number of clients, handling mail for each client as new information is posted.

For typical autonomous vehicle implementations, there is a MOOS community on board each vehicle. When simulating multiple vehicle on a single machine, there is also a single community associated with each vehicle. The communications discussed in today's lab concern how a single app communicates with another app via the publish-subscribe architecture. Later labs will address how vehicles communicate with each other, essentially bridging two or more MOOS communities with one another.

For today, the focus is on the MOOSDB and connected applications. The MOOSDB, unlike an actual database, does not contain a full history of information that has passed through it. At most, it stores the latest value for any given MOOS variable published to the MOOSDB. When a new app connects to the MOOSDB it must register for the mail it needs. On startup, an app can expect to get a mail message containing the latest value for any variable it registers for, even if that

mail reflects a posting to the MOOSDB long ago. Anything happening prior to that is unknown to the newly connected app.

2 Launching the MOOSDB

In this exercise we will focus on the module that is at the heart of MOOS, the MOOSDB. The MOOSDB application is a server that runs on the robot or unmanned vehicle platform's computer, or simply on your laptop during simulation. It may be launched from the command line, assuming it is in your path as decribed in Section 1.1. In this exercise we will proceed by following the example at:

> Please see "Launching the DB" in the MOOS Basics section.

Your goals in this part are:

1. Open a terminal window and launch the MOOSDB as shown on the webpage. Confirm that you do see the same/similar output.

2. Open a browser and enter the URL shown at the bottom of the MOOSDB output. It should be something like:

   ```
   serving webpages HTTP on http://fred.csail.mit.edu:9080
   ```

   The browwser should show three MOOS variables. `DB_UPTIME` reflects the number of seconds since the MOOSDB was launched. The `DB_TIME` variable reflects the Unix time, i.e., the number of seconds since midnight January 1, 1970. The `DB_CLIENTS` time shows the list of clients connected to the MOOSDB. Click on the refresh button to show how the values change.

3. Create a mission file, `moosdb_alpha.moos` with the three lines

   ```
   ServerHost = localhost
   ServerPort = 9000
   Community  = alpha
   ```

   Re-launch the MOOSDB from the command-line, passing the newly created file above as an argument. Note the difference in output produced by the MOOSDB.

3 Scoping the MOOSDB

A MOOS *scope* is a tool for examining the state of the MOOSDB. Recall that the MOOSDB does not keep a history of prior variable values, but rather just the most recent value posted for any given variable. This means that the *state* of the MOOSDB may be regarded as the set of current MOOS variables, their values, and who made the last posting to the variable and when.

In this exercise we will proceed by following the example at:

> Please see "Scoping the DB" in the MOOS Basics section.

3.1 Scoping with uXMS

The `uXMS` tool is one of two scopes we'll introduct in this section.

Your goals in this part are:

1. Open a terminal window and launch the MOOSDB as done at the end of the previous exercise:

```
% MOOSDB moosdb_alpha.moos
```

2. Open a second terminal window and launch `uXMS`, passing it the same mission file as an argument, and two other command-line switches:

```
% uXMS moosdb_alpha.moos --all
```

   Your second terminal window should look like the output on the webpage. Notice the number in parentheses on the second line is incrementing. This indicates that the report has been refreshed to your terminal. If you launched as above, the scope should come up in a mode that refreshes the report any time a scoped variable changes values. In this case, the MOOSDB is updating `DB_TIME` and `DB_UPTIME` about once per second.

   By default, `UXMS` only scopes on the variables named on the command line. In the above case, the `--all` option was used to tell `uXMS` to scope on all variables known to the MOOSDB.

   Try a few other things:

   - Hit the `'h'` key to see some keyboard interaction options that are available anytime the scope is running. Hit `'e'` any time to return to the previous mode.
   - Hit the space-bar to pause the stream of reports. This is useful if numbers are changing rapidly and you just need to take a close look at something. Return to the previous mode by hitting `'e'`.
   - Hit the `'S'` (upper-case) key to expand the `(S)ource` column. This column tells you which app made the last posting. Collapse this column by hitting the `'s'` (lower-case) key. Try the same for the `(T)ime` and `(C)ommunity` columns.

3. The whole purpose of a scope is to give you the key information you're looking for, without needing to sift through a lot of unwanted information, with as little effort as possible. In this step we'll pretend to be interested in focusing our attention on the DB_UPTIME variable. Try launching uXMS with an additional command line argument:

```
% uXMS moosdb_alpha.moos --all --colormap=DB_UPTIME,blue
```

   This may seem unnecessary when there are only three variables, but in real applications there may be hundreds of variables. In fact, the variable you're looking for may have scrolled of the window! So a similar way to accomplish the above is to only scope on the one variable we're looking for:

```
% uXMS moosdb_alpha.moos DB_UPTIME
```

4. uXMS only shows you the current snapshot of the variables in the MOOSDB. What if you would like to see how a variable is changing? In our case, we know how the DB_UPTIME variable is changing, but for the sake of showing this feature, try:

```
% uXMS moosdb_alpha.moos --history=DB_UPTIME
```

3.2 Scoping with uMS

The uMS scope is a graphical MOOS scope, often preferred by those inclined to like GUIs vs. command line tools. It has some other advantages over uXMS as well.

Your goals in this part are:

1. If you don't still have a MOOSDB running, open a terminal window and launch the MOOSDB as done previously:

```
% MOOSDB moosdb_alpha.moos
```

2. Open a second terminal window and launch uMS, passing it the same mission file as an argument:

```
% uMS moosdb_alpha.moos
```

   You should see a window open and, after clicking on the Connect button, you should see something similar to:

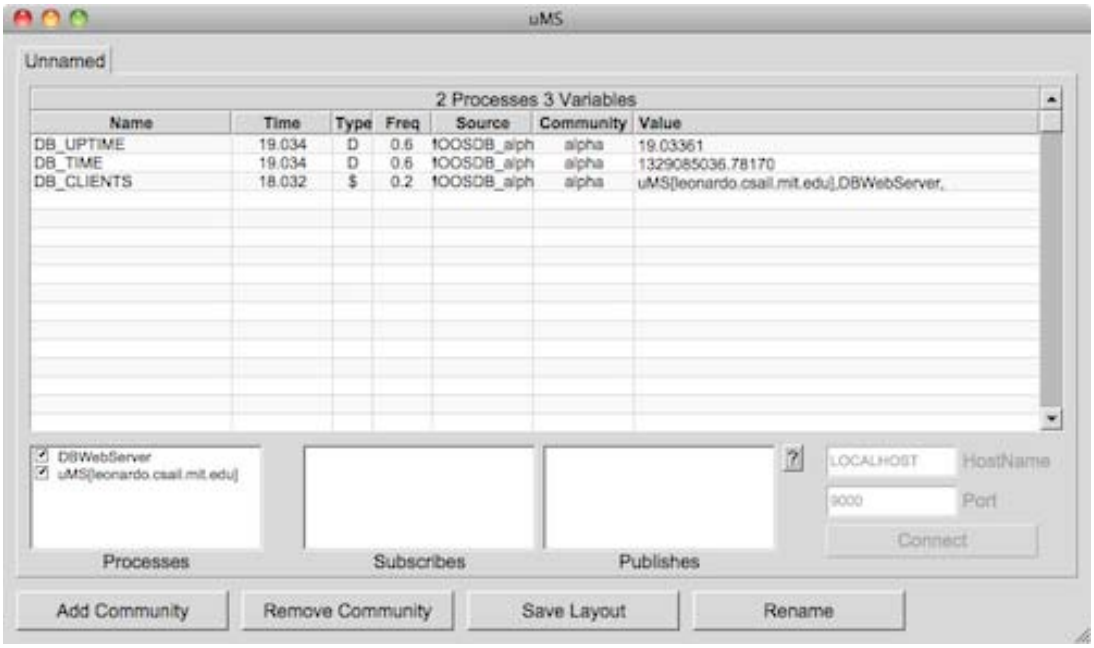

Figure 3: The uMS MOOS Scope

Pros/Cons of uXMS vs uMS

The choice of `uXMS` vs `uMS` is often just a matter of taste. A couple of differences are noteworthy however.

- uMS allows for connections to multiple MOOSDBs, on perhaps multiple vehicles, simultaneously. The user may select the vehicle with the tab at the top of the screen.
- uXMS allows the user to scope on as few as one single variable, or to name the variable scope list explicitly. uMS scopes on all variables all the time, with a few mechanisms for reducing the scope list based on process name.
- uXMS may be a better choice if one is scoping on a remote MOOSDB, perhaps on a robot with a poor connection. It is a low-bandwidth client compared to uMS. If running on a remote terminal, its bandwidth back to the user is zero in the paused mode.
- uXMS has provisions for at least limited scoping on a variable history.
- uXMS will display a variable's "auxilliary source" information. This is a secondary field associated with each posting describing the source of the posting. This is key when using the IvPHelm. Variables posted by helm behaviors will have a source of pHelmIvP and an auxilliary source showing the behavior responsible for the posting.

4 Poking the MOOSDB

Poking refers to the idea of publishing a variable-value pair to the MOOSDB. Many apps publish to the MOOSDB during the course of normal operation. Poking implies a publication that perhaps was not planned, or outside the normal mode of business. It is often very useful for debugging. Here we describe the uPokeDB tool.

In this exercise we will proceed by following the example at:

> Please see "Poking the DB" in the MOOS Basics section.

Your goals in this part are:

1. If you don't still have the MOOSDB and uXMS running, open two terminal windows and launch the MOOSDB and uXMS as done previously:

```
% MOOSDB moosdb_alpha.moos
```

```
% uXMS moosdb_alpha.moos --all
```

   Now open a third terminal window for poking the MOOSDB as shown on the webpage as follows:

```
% uPokeDB DEPLOY=true RETURN_SPEED=2 STRING_SPEED:=2.3 moosdb_alpha.moos
```

   This pokes three variables, `DEPLOY` of type string, `RETURN_SPEED` of type double, and `STRING_SPEED` of type string (note the `":="`). Verify that this is the case by examining your uXMS window. Note that in VarValue column there double-quotes around the value `"2.3"`.

2. Trying poking the MOOSDB a second time, this time with:

```
% uPokeDB DEPLOY=100 moosdb_alpha.moos
```

   Does the value of `DEPLOY` change? If not, why not?

3. Create a simple script of pokes on the command line as follows:

```
% uPokeDB APPLES=1 moosdb_alpha.moos; sleep 5; uPokeDB APPLES=2 moosdb_alpha.moos;
```

   Watch it play out in the terminal window running uXMS. Try creating a similar, longer script by creating a file `myscript`

```
uPokeDB APPLES=1 moosdb_alpha.moos;
sleep 2
uPokeDB APPLES=2 moosdb_alpha.moos
sleep 2
uPokeDB APPLES=3 moosdb_alpha.moos
sleep 2
uPokeDB APPLES=4 moosdb_alpha.moos
sleep 2
uPokeDB APPLES=5 moosdb_alpha.moos
sleep 2
uPokeDB APPLES=6 moosdb_alpha.moos
```

Launch the script by typing `"source myscript"` on the command line. Watch the script play out again in the uXMS window.

Scripting is often useful. In Section 6 we'll see a more robust method for scripted pokes to the MOOSDB.

5 Launching MOOS Missions with pAntler

In theory, a set of N applications may be launched from N terminal windows, but this is cumbersome in practice. The pAntler tool allows a block to be declared in a mission file (.moos file) listing all the apps to be launched in one go.

In this exercise we will proceed by following the example at:

> Please see "Launching with pAntler" in the MOOS Basics section.

Your goals in this part are:

1. Create a copy of the example mission file shown on the webpage and in Listing 1 below and save it locally as `db_and_uxms.moos`. (hint: the easiest way to do this is to just invoke the wget expression on the top line of this file. This will pull the file down from the server into your current directory.)

   You should see a single uXMS window pop up upon launch.

2. Modify the uXMS configuration block in the .moos file to configure uXMS to keep a history of the `DB_UPTIME` variable. To see configuration options for uXMS, type:

   ```
   % uXMS --example
   ```

   Once you have launched uXMS with the new configuration, type `'z'` to toggle in and out of history mode.

3. Modify the `db_and_uxms.moos` file to launch a new terminal window for the MOOSDB in addition the uXMS application.

Example Code 1.

```
 0  // (wget http://oceanai.mit.edu/2.S998/examples/db_and_uxms.moos)
 1  ServerHost = localhost
 2  ServerPort = 9000
 3  Community  = alpha
 4
 5  ProcessConfig = ANTLER
 6  {
 7    MSBetweenLaunches = 200
 8
 9    Run = MOOSDB     @ NewConsole = false
10    Run = uXMS       @ NewConsole = true
11  }
12
13  ProcessConfig = uXMS
14  {
15    AppTick   = 4
16    CommsTick = 4
17
18    VAR            = DB_CLIENTS, DB_UPTIME, DB_TIME
19    DISPLAY_SOURCE = true
20    DISPLAY_TIME   = true
21    COLOR_MAP      = DB_CLIENTS, red
22  }
```

6 Scripted Pokes to the MOOSDB

This exercise covers how to have a script of pre-arranged pokes to the MOOSDB. This may be useful for a number of reasons besides debugging. The primary tool described here is the uTimerScript MOOS application. It is capable of (a) scripted pokes at a pre-defined times after launch, (b) pokes times specified to fall randomly within an specified interval, (c) pokes having numerical values falling with a uniformly random interval, and several other features including conditioning the running of the script based on other MOOS variables.

In this exercise we will proceed by following the example at:

> Please see "MOOS Scripting" in the MOOS Basics section.

Your goals in this part are:

1. Create a copy of the example mission file shown on the webpage and in Listing 2 below and save it locally as `utscript.moos`. (hint: use wget!)
2. Launch the mission. It should open a uXMS window. Follow the progress of the counter script.

```
% pAntler utscript.moos
```

3. Take a look at the uTimerScript documentation linked from the web page. In particular, section 10.3.1. Configure the script such that is paused when uTimerScript is launched. Launch the same mission and confirm that the script is initially not running. Then use uPokeDB to un-pause the script, and confirm it is running.
4. This is a bit of a pAntler exercise. Configure your mission to launch two versions of the script, the second version publishing to `COUNTER_B`. Note you will need two configuration blocks, each with a unique name. Hint: see the pXRelay example on the web page, or in the next section.
5. Confirm your new mission launches and executes the two separate scripts and both counters are incrementing.
6. Configure the second script with a `CONDITION` parameter. See Section 10.3.2 of the uTimerScript documentation. Use a condition such as `"CONDITION = COUNTER_A > 5"`. Re-launch your mission. Confirm that the second script is paused periodically based on the state of the first script.
7. Add the pLogger application to your mission. You will need to add a pLogger entry to your ANTLER configuration block, and add the following pLogger configuration block at the end of your file.

```
ProcessConfig = pLogger
{
  AsyncLog = true
  WildCardLogging = true
  WildCardOmitPattern = *_STATUS
}
```

Re-run the mission. Confirm that you see the pLogger application listed in the DB_CLIENTS variable in the uXMS scope.

8. Verify that a log file has been created. Since we didn't specify a name for the log file, by default it should be in a subdirectory of where you launched the mission, looking something like MOOSLog_12_2_2012_____12_38_03. Enter the directory and confirm that you see a .alog file.

9. Take a look a the file by typing more filename.alog. Then take a look at the COUNTER variables using aloggrep:

```
% aloggrep COUNTER_A MOOSLog_12_2_2012_____12_38_03.alog
```

Example Code 2.

```
// (wget http://oceanai.mit.edu/2.S998/examples/utscript.moos)
ServerHost = localhost
ServerPort = 9000
Community  = alpha

ProcessConfig = ANTLER
{
  MSBetweenLaunches = 200

  Run = MOOSDB       @ NewConsole = false
  Run = uXMS         @ NewConsole = true
  Run = uTimerScript @ NewConsole = false
}

ProcessConfig = uXMS
{
  VAR            = COUNTER_A, DB_CLIENTS, DB_UPTIME
  COLOR_MAP      = COUNTER_A, red
  HISTORY_VAR    = COUNTER_A
}

ProcessConfig = uTimerScript
{
  paused = false

  event  = var=COUNTER_A, val=1,  time=0.5
  event  = var=COUNTER_A, val=2,  time=1.0
  event  = var=COUNTER_A, val=3,  time=1.5
  event  = var=COUNTER_A, val=4,  time=2.0
  event  = var=COUNTER_A, val=5,  time=2.5
  event  = var=COUNTER_A, val=6,  time=3.0
  event  = var=COUNTER_A, val=7,  time=3.5
  event  = var=COUNTER_A, val=8,  time=4.0
  event  = var=COUNTER_A, val=9,  time=4.5
  event  = var=COUNTER_A, val=10, time=5:10

  reset_max  = nolimit
  reset_time = all-posted
}
```

7 A Simple Example with pXRelay

pXRelay is a simple MOOS app designed solely to illustrate basic functions of a MOOS app. It registers for a single variable, and upon receipt mail for that variable, it publishes another variable incremented by 1. It provides a framework for illustrating few other introductory topics.

In this exercise we will proceed by following the example at:

> Please see "Example with pXRelay" in the MOOS Basics section.

Your goals in this part are:

1. Create a copy of the example mission file shown on the webpage and in Listing 3 below and save it locally as `pxrelay.moos`. (hint: use wget!)
2. Launch the mission. Open up uXMS in another Terminal window with the parameters of your choosing. I recommend

```
% uXMS pxrelay.moos --colorany=APPLES,PEARS --all
```

3. As described on the web page, kick off the activity by poking one of the APPLES or PEARS variables with an initial value. Confirm that things are working as they should. Perhaps see the pXRelay pdf linked on the webpage for a more detailed description of how things should look.
4. Add uTimerScript to your mission file, with a simple script to kick off the pXRelay handshaking at some point after launch (say 10 secs), as an alternative way to kicking off the active instead of uPokeDB. You'll need to add uTimerScript to your ANTLER configuration block, and add a simple script (a uTimerScript configuration block) to your .moos file.
5. Change your uTimerScript script to be the ascending counter script from Section 6, incrementing `COUNTER_A` 1 to 10. Add a condition to your script (`APPLES == $(PEARS)`). Re-launch the revised mission. Since `APPLES=PEARS` periodically, the condition will periodically be met.
6. Try changing the AppTick in the pXRelay configurations to perhaps allow more time in the `APPLES == $(PEARS)` state.

Example Code 3.

```
 0  // (wget http://oceanai.mit.edu/2.S998/examples/xrelay.moos)
 1  ServerHost = localhost
 2  ServerPort = 9000
 3  Community  = alpha
 4
 5  ProcessConfig = ANTLER
 6  {
 7    MSBetweenLaunches = 200
 8
 9    Run = MOOSDB       @ NewConsole = false
10    Run = pXRelay      @ NewConsole = false ~pXRelay_PEARS
11    Run = pXRelay      @ NewConsole = false ~pXRelay_APPLES
12  }
13
14  ProcessConfig = pXRelay_APPLES
15  {
16    AppTick       = 10
17    CommsTick     = 10
18    INCOMING_VAR  = APPLES
19    OUTGOING_VAR  = PEARS
20  }
21
22  ProcessConfig = pXRelay_PEARS
23  {
24    AppTick       = 10
25    CommsTick     = 10
26    INCOMING_VAR  = PEARS
27    OUTGOING_VAR  = APPLES
28  }
```

8 If Time Permits...

If you have time remaining in this lab, here are some suggestions:

1. Take a look at the pXRelay source code (we will be in the next lab). It can be found at `moos-ivp/ivp/src/pXRelay/`. Look inside PXR_MOOSApp.cpp. Check out the `OnNewMail()` and `Iterate()` loops.
2. Can you hack this code to make its increments by ten instead of one?
3. Can you hack this code such that it takes *two* incoming variables, and does its thing if *either* of the two variables is written to?

2.S998 Marine Autonomy, Sensing and Communications

Spring 2012