Lab 2 - Introduction to MOOS

Spring 2019

Michael Benjamin, mikerb@mit.edu
Henrik Schmidt, henrik@mit.edu
Department of Mechanical Engineering
MIT, Cambridge MA 02139

1 Overview and Objectives **3**
1.1 Preliminaries . . . 3
1.2 MOOS vs. MOOS-IvP . . . 3
1.3 More MOOS / MOOS-IvP Resources . . . 4
1.4 The MOOS Architecture . . . 4
1.5 Launching the MOOSDB . . . 6
1.6 Scoping the MOOSDB . . . 8
1.7 Poking the MOOSDB . . . 12
1.8 Launching a Mission with pAntler . . . 15
1.9 Scripted Pokes to the MOOSDB . . . 17

2 A Simple Example with pXRelay **21**
2.1 Basic pXRelay Usage . . . 21
2.2 A Simple Example with pXRelay . . . 21

3 Modify the pXRelay Code **23**

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 generate scripted interactions with the MOOSDB, (d) understand how the logger operates and basic tools for examining log files

- MOOS Preliminaries: MOOS vs. MOOS-IvP, the MOOS Architecture
- Launching, Scoping, and Poking the MOOSDB
- Launching a Mission with pAntler
- Scripted Pokes the to the MOOSDB
- A Simple Example with pXRelayTest
- Modify the pXRelayTest Code

1.1 Preliminaries

This lab assumes you have a working MOOS-IvP tree checked out and built 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/bin/MOOSDB
$ which pHelmIvP
/Users/you/moos-ivp/bin/pHelmIvP
```

If unsuccessful with the above, return to the steps in Lab 1:

http://oceanai.mit.edu/pavlab/pmwiki/pmwiki.php?n=MiniCourse.LabIntro

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.

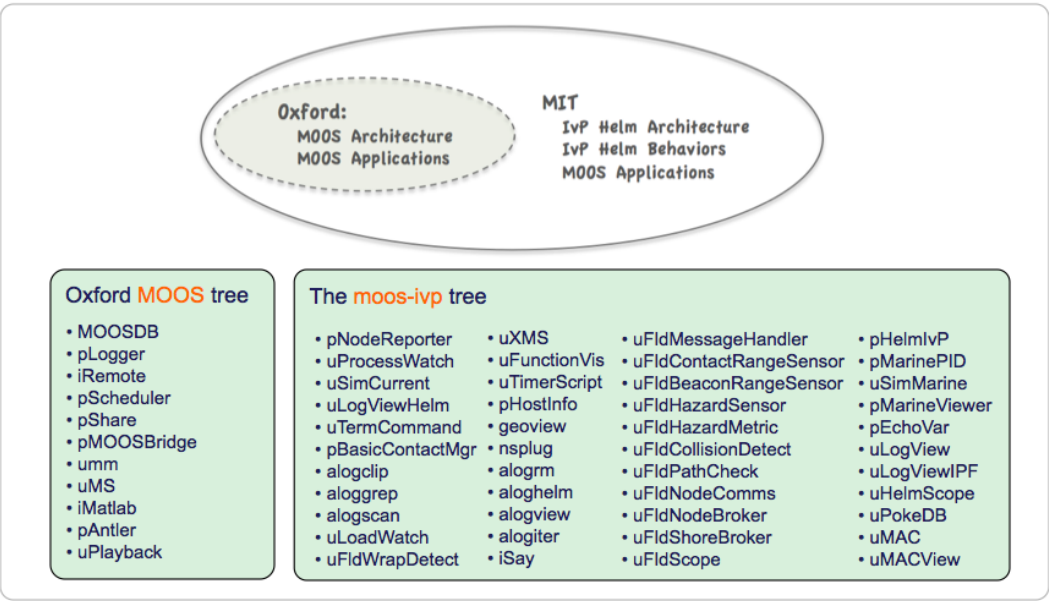

Figure 1: **Nested Repositories:** The MOOS-IvP tree contains the Oxford MOOS tree and additional modules from MIT including the Helm architecture, Helm behaviors and further MOOS applications.

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.

- The "Very Brief Overview of MOOS" page on the course documentation page:
  http://oceanai.mit.edu/ivpman/pmwiki/pmwiki.php?n=Helm.MOOSOverview
- Follow the links to the documentation on the Oxford MOOS website.
  http://themoos.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 app. A MOOS *community* is a collection of applications connected to a single MOOSDB.

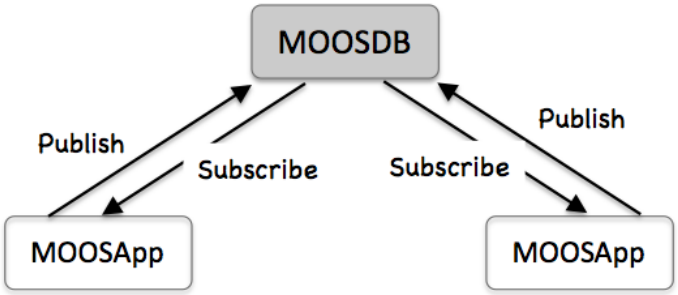

Figure 2: **The MOOS Architecture:** MOOS is a publish-subscribe architecture. The MOOSDB serves a number of clients, handling mail for each client as new information is posted. A MOOSDB with connected clients constitutes a *MOOS community*. There may be multiple MOOS communities on a single machine, and a single MOOS community may be distributed over more than one machine.

For typical autonomous vehicle implementations, there is a MOOS community on board each vehicle. When simulating multiple vehicles on a single machine, there is also a single community associated with each vehicle. A *MOOS community* consists of a single MOOSDB with one or more connected clients. The communications discussed in today's lab concern how a single app communicates with another app via the MOOSDB in 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 will be unknown to the newly connected app.

1.5 Launching the MOOSDB

Here we describe how to launch the MOOSDB from the perspective of the first-time user. The MOOSDB application is a server that runs on the robot or unmanned vehicle computer, or simply on your laptop during simulation. It may be launched from the command line, assuming it is in your path. Two minimalist methods are described here, starting with the most bare-bones.

1.5.1 A Bare-Bones Launching of the MOOSDB

In a bare-bones manner, the MOOSDB may be launched from the command line without any arguments. Normally the MOOSDB needs to know at least two pieces of configuration information, (a) the machine (IP address) on which to run, and (b) the port number on which to serve clients. It will default to running on the localhost and port 9000:

```
$ MOOSDB
------------------- MOOSDB V10 -------------------
Hosting  community                "#1"
Name look up is                   off
Asynchronous support is           on
Connect to this server on port    9000
--------------------------------------------------
network performance data published on localhost:9020
listen with "nc -u -lk 9020"
```

At this point the MOOSDB is running on the local machine, serving clients on port 9000. A few variables are already being published, by the MOOSDB itself. You can open a scope with uXMS in another terminal window:

```
 $ uXMS DB_CLIENTS DB_TIME DB_UPTIME --serverhost=localhost --serverport=9000

 ================================================================
 uXMS_581                                              0/0(154)
 ================================================================
 VarName     (S)ource   (T)ime  (C)  VarValue (SCOPING:EVENTS)
 ----------  ---------  ------  ---  -------------------------
 DB_CLIENTS  MOOSDB_#1  38.14        "uXMS_581,"
 DB_TIME     MOOSDB_#1  38.14        1387380731.414707
 DB_UPTIME   MOOSDB_#1  38.14        39.00859
```

The Source and Time columns may be expanded by hitting the `"s"` and `"t"` keys respectively after it launches, or you can add `--show=source,time` as a command line argument to uXMS, to launch with these two columns expanded.

Alternatively you can scope with the umm tool:

```
$ umm --spy
```

1.5.2 A More Civilized Launching of the MOOSDB

Virtually all MOOS applications are launched with a "mission configuration" file, a.k.a. a "dot-moos" file. The below mission file, `moosdb_alpha.moos`, provides the minimal configuration parameters the MOOSDB likes to see upon starting.

```
// (wget http://oceanai.mit.edu/2.680/examples/moosdb_alpha.moos)
ServerHost = localhost
ServerPort = 9000
Community  = alpha
```

Passing the `moosdb_alpha.moos` file as a command line argument produces the below output.

```
$ MOOSDB moosdb_alpha.moos
------------------- MOOSDB V10 -------------------
  Hosting  community                "alpha"
  Name look up is                   off
  Asynchronous support is           on
  Connect to this server on port    9000
--------------------------------------------------
network performance data published on localhost:9020
listen with "nc -u -lk 9020"
```

Try this. Either copy and paste the above `.moos` file, or use `wget` as shown on the first line above in the example file. If you haven't installed `wget` or similar (e.g., `curl`), you should pause here and do that.

1.6 Scoping the MOOSDB

A MOOS *scope* is a tool for examining the state of the MOOSDB. The MOOSDB does not keep a history of prior values for a given variable, but rather just the most recent value posted. 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. Scoping allows a view into the current state (or even recent history) of the MOOSDB. There are multiple tools for scoping the DB, each providing conveniences of one kind or another. Here we describe the uXMS and uMS tools, the two favorites of our own lab.

More info on these two tools can be found at:

- uXMS: http://oceanai.mit.edu/ivpman/apps/uXMS
- uMS: http://www.themoos.org

1.6.1 Scoping the MOOSDB with uXMS

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 V10 --------------------
 Hosting  community                "#1"
 Name look up is                   off
 Asynchronous support is           on
 Connect to this server on port    9000
---------------------------------------------------
network performance data published on localhost:9020
listen with "nc -u -lk 9020"
```

   The MOOSDB is normally launched with a mission file specifying the ServerHost and ServerPort parameters. When launched from the command line as above with no command line arguments, these two parameters default to `localhost` and `9000`.

2. Open a second terminal window and launch uXMS, passing it the `--all` command line switch. Just hit `ENTER` when prompted for the IP address and Port number, accepting the defaults of `localhost` and `9000`. It should look something like:

```
$ uXMS --all
Enter IP address:  [localhost]
Enter Port number: [9000]
***************************************************
*  uXMS_632 starting ...
***************************************************
uXMS_632 is Running:
 |-Baseline AppTick   @ 5.0 Hz
 |--Comms is Full Duplex and Asynchronous
 -Iterate Mode 0 :
   |-Regular iterate and message delivery at 5 Hz
```

After the above initial standard MOOSApp output, you should see a sequence of uXMS reports similar to:

```
============================================================
uXMS_443                                            0/0(31)
============================================================
VarName            (S)  (T)  (C)  VarValue (SCOPING:EVENTS)
-----------------  ---  ---  ---  -------------------------
DB_CLIENTS                       "uXMS_443,"
DB_EVENT                         "connected=uXMS_443"
DB_QOS                           "uXMS_443=0.448942:0.573874:0.364065:0.490189,"
DB_TIME                          1424028179.877422
DB_UPTIME                        16.741557
UXMS_443_ITER_GAP                1.019045
UXMS_443_ITER_LEN                0.00024
  -- displaying all variables --
```

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.

The three variables shown beginning with `DB_` are all published by the MOOSDB. The user may choose whether or not to show the variable (S)ource, (T)ime of post, or (C)ommunity from where the post was made. A key feature of uXMS vs. uMS is the ability to specify on the command-line exactly which subset of variables to scope, possibly with color-coding. This is helpful when there are hundreds of variables in the DB.

1.6.2 More Suggested Tinkering with uXMS

Try a few other things:

1. Hit the `'h'` key to see some keyboard interaction options that are available anytime the scope is running. Hit `'h'` again any time to return to the previous mode.
2. 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'`.
3. Hit the `'s'` key to expand the `(S)ource` column. This column tells you which app made the last posting. Try the same for the `(T)ime` and `(C)ommunity` columns.
4. 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 --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!

5. A similar way to focus on a single variable is to only scope on the one variable we're looking for:

```
$ uXMS DB_UPTIME
```

6. 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 --history=DB_UPTIME
```

1.6.3 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
```

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

```
$ uMS
```

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

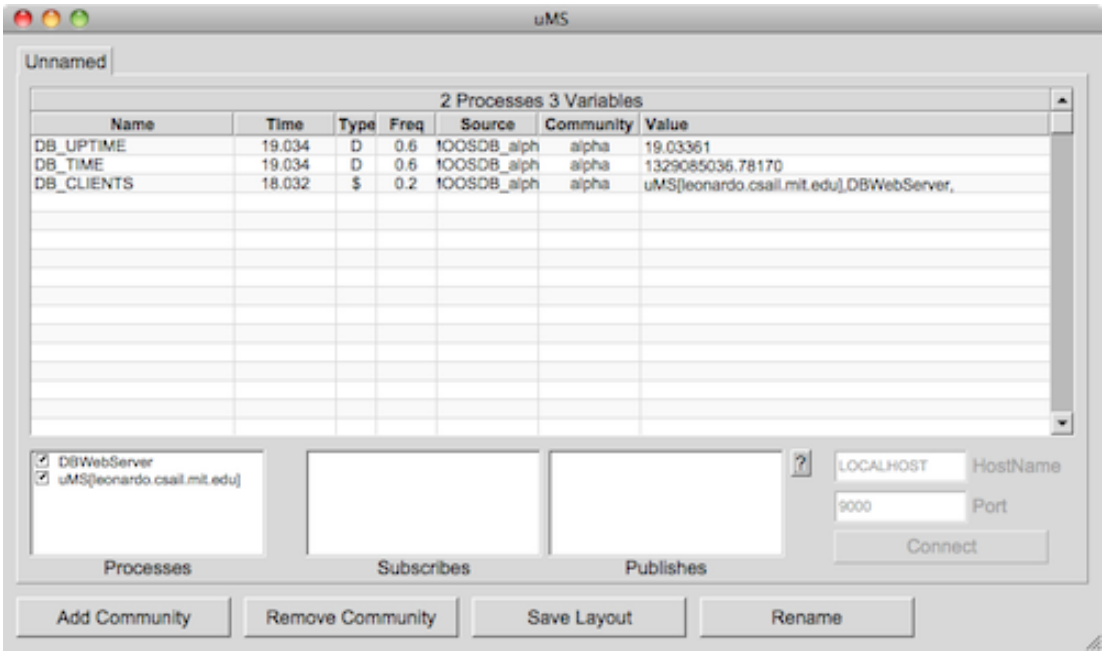

Figure 3: The uMS MOOS Scope

1.6.4 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 "auxiliary 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 auxiliary source showing the behavior responsible for the posting.

1.7 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.

Where to get more information:

- uPokeDB: http://oceanai.mit.edu/ivpman/apps/uPokeDB

1.7.1 Poking the MOOSDB with uPokeDB

uPokeDB is a command-line tool for poking the MOOSDB with one or more variable-value pairs. Poking the MOOSDB requires knowing where the MOOSDB is running in terms of its IP address, ServerHost, and port number, ServerPort. These may be specified on the command line to uPokeDB, but for our purposes here we assume the existence of a mission file, `alpha.moos` with this information:

```
// A simple mission file, alpha.moos
ServerHost = localhost
ServerPort = 9000
Community  = alpha
```

Your goals in this part are:

1. Open two terminal windows and launch the MOOSDB and uXMS as done previously:

   ```
   $ MOOSDB alpha.moos
   $ uXMS alpha.moos --all
   ```

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

   ```
   $ uPokeDB DEPLOY=true SPEED=2 alpha.moos
   ```

   Note the two new variables, DEPLOY and SPEED, appearing in the uXMS window. It should look something like:

   ```
   ================================================================
   uXMS_655                                              0/0(204)
   ================================================================
   VarName     (S)  (T)  (C)  VarValue (SCOPING:EVENTS)
   ----------  ---  ---  ---  ------------------------
   DB_CLIENTS                "uXMS_655,"
   DB_TIME                   1386249435.276804
   DB_UPTIME                 46.213629
   DEPLOY                    "true"
   SPEED                     2
   ```

2. Note the variable values in uXMS. DEPLOY has the value `"true"` with double quotes, indicating that it is a string. The variable SPEED is of type double, indicated by the lack of quotes. The

types were inferred by uPokeDB by heuristically checking whether the arguments are numerical or not. But sometimes you *do* want to publish a string with a numerical value. Try posting the variable HEIGHT with the string value of "192", noting the colon-equals instead of equals:

```
$ uPokeDB HEIGHT:=192 alpha.moos
```

Note the new variable, HEIGHT, appearing in the uXMS window. It should look something like the below output, where in this case, the (S)ource column is expanded to show the source of the postings.

```
================================================================
uXMS_655                                              0/0(347)
================================================================
VarName     (S)ource     (T)  (C)  VarValue (SCOPING:EVENTS)
----------  ---------    ---  ---  -------------------------
DB_CLIENTS  MOOSDB_alpha          "uXMS_655,"
DB_TIME     MOOSDB_alpha          1386250092.847527
DB_UPTIME   MOOSDB_alpha          703.784353
DEPLOY      uPokeDB               "true"
HEIGHT      uPokeDB               "192"
SPEED       uPokeDB               2
```

1.7.2 Further things to try Using uPokeDB

Here's some other things to consider and try:

1. Trying poking the DEPLOY variable to the MOOSDB a second time, this time with:

   ```
   $ uPokeDB DEPLOY=100 alpha.moos
   ```

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

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

   ```
   $ uPokeDB APPLES=1 alpha.moos; sleep 5; uPokeDB APPLES=2 alpha.moos;
   ```

   If you're new to the command line environment, the semicolon above separates successful command line invocations. The sleep command is a common shell utilitity that will simply pause a given number of seconds before completing.

3. Another way to execute the same simple script as above is to store the above three commands in a file named, for example, myscript:

   ```
   uPokeDB APPLES=1 alpha.moos
   sleep 5
   uPokeDB APPLES=2 alpha.moos
   ```

   With the above file you can make the two successive pokes to the MOOSDB, with five seconds in between, with:

   ```
   $ source myscript
   ```

There are many other ways of poking the MOOSDB. All MOOS apps that publish anything are examples. Of course many MOOS applications publish a fixed set of variables that are not easily changeable without re-coding. But certain apps like uTimerScript and pMarineViewer have built-in configuration file parameters for poking the MOOSDB in user configurable ways.

1.8 Launching a Mission with pAntler

In theory a set of N MOOS applications may be launched from N terminal windows, but this is cumbersome in practice. The pAntler tool allows this to be done from a single mission file. In this file, a block of lines declares all the apps to be launched with one invocation of pAntler.

Where to get more information:

- pAntler: http://oceanai.mit.edu/ivpman/apps/pAntler

1.8.1 Basic pAntler Usage

The Antler block is typically the first configuration block in a .moos file, declared with ProcessConfig = ANTLER as below. The MSBetweenLaunches parameter specifies the number of milliseconds between launching processes. Each line thereafter specifies an app to be launched and whether a dedicated console window should be opened for the application.

```
ProcessConfig = ANTLER
{
  MSBetweenLaunches = 200

  Run = MOOSDB       @ NewConsole = true/false
  Run = AnotherApp   @ NewConsole = true/false
  ...
  Run = AnotherApp   @ NewConsole = true/false
}
```

Further options exist beyond the vanilla launch configuration described above, including (a) the ability to launch a given app under an aliased name, (b) specifying command-line arguments to an app at launch time and more. See the documentation.

1.8.2 An Example: Launching the MOOSDB along with uXMS

In the example below we use pAntler to launch the MOOSDB and the uXMS scope from a single mission file. The user preferences for uXMS are provided in its configuration block. Type uXMS --example on the command line for further options.

Your goals in this part are:

1. Create a copy of the example mission file shown 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.) The mission may be launched from the command-line with:

   ```
   $ pAntler db_and_uxms.moos
   ```

   This should open a new console window for uXMS displaying the variables posted by the DB, with the (S)ource and (T)ime columns expanded, but not the (C)ommunity column.

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 to the uXMS application.

Listing 1.1: A simple mission file.

```
// (wget http://oceanai.mit.edu/2.680/examples/db_and_uxms.moos)
ServerHost = localhost
ServerPort = 9000
Community  = alpha

ProcessConfig = ANTLER
{
  MSBetweenLaunches = 200

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

ProcessConfig = uXMS
{
  AppTick   = 4
  CommsTick = 4

  VAR            = DB_CLIENTS, DB_UPTIME, DB_TIME
  DISPLAY_SOURCE = true
  DISPLAY_TIME   = true
  COLOR_MAP      = DB_CLIENTS, red
}
```

1.9 Scripted Pokes to the MOOSDB

Here we cover 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 having a poke-time 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.

Where to get more information:

- uTimerScript: http://oceanai.mit.edu/ivpman/apps/uTimerScript

1.9.1 Basic uTimerScript Usage

uTimerScript is configured with its own block in the MOOS configuration file. The general format is below. The primary entries are the events themselves, defined by a MOOS variable, value, and time or time-range when the event is to occur. There are many options for configuring the script. These options are described in the documentation, but a quick look at the options can be seen by typing uTimerScript --example on the command line.

```
ProcessConfig = uTimerScript
{
  event = var=<MOOSVar>, val=<value>, time=<value>
  event = var=<MOOSVar>, val=<value>, time=<value>
  ...
  event = var=<MOOSVar>, val=<value>, time=<value>

  [OPTIONS]
}
```

1.9.2 A Simple Example with uTimerScript

The below mission file contains a uTimerScript script for repeatedly posting the variable COUNTER_A with values 1-10 in ascending order roughly once every half second. The last event in the script is posted at time chosen from a random five second interval.

Listing 1.2: A simple counter example with uTimerScript.

```
// (wget http://oceanai.mit.edu/2.680/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
}
```

The mission may be launched from the command-line with:

```
 $ pAntler utscript.moos
```

This should open a new console window for uXMS displaying the variables COUNTER_A variable repeatedly

incrementing from 1 to 10. Note that reaching 10 happens somewhere between 0.5 and 5.5 seconds after reaching 9.

1.9.3 Exercises

Your goals in this part are:

1. Create a copy of the example mission file in Listing 2 above and save it locally.
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 6.3 Script Flow Control. 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. Hint: to un-pause the script with uPokeDB, you'll need to know which variable to poke, and this also can be found in the uTimerScript documentation or by typing uTimerScript -i on the command line.
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. And you will need to launch uTimerScript twice within the Antler block, each with an alias. Hint: see the pXRelay at the end of Lab 3.
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_11_23_2016_____11_31_13/. Enter the directory and confirm that you see an .alog file.
9. Take a look a the file by typing more filename.alog. Then take at look at the COUNTER variables using aloggrep (substituting of course the name of your .alog file:

```
$ aloggrep COUNTER_A COUNTER_B   MOOSLog_11_23_2016_____11_31_13.alog
```

2 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 receiving mail for that variable, it publishes another variable incremented by 1. It provides a framework for illustrating a few other introductory topics.

2.1 Basic pXRelay Usage

pXRelay is configured with its own block in the MOOS configuration file. It is configured with (a) an incoming variable, the variable it will register for incoming mail, and (b) an outgoing variable, a variable it will post an incremented integer each time it receives mail on the incoming variable. The basic form is:

```
ProcessConfig = pXRelay
{
  outgoing_var = <MOOSVar>
  incoming_var = <MOOSVar>
}
```

2.2 A Simple Example with pXRelay

The below mission file contains a configuration for two instances of the pXRelay application. All MOOS apps must have a unique name to connect to the MOOSDB, so we launch them with an alias with pAntler using the pXRelay_PEARS alias for example. The two apps each register for what the other produces, and each produces what the other registers for.

Listing 2.3: Example Code.

```
 0   // (wget http://oceanai.mit.edu/2.680/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   }
```

Upon launch, the two pXRelay apps are in a stalemate, each waiting for the other to make the first posting. We can break this stalemate with uPokeDB:

```
$ uPokeDB xrelay.moos PEARS=1
```

This should get things going. Now it would be good to see it all running by launching a scope:

```
$ uXMS xrelay.moos --all --show=time
```

Your goals in this part are:

1. Create a copy of the example mission file shown in Listing 3 above 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. 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.
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 1.9, incrementing COUNTER_A 1 to 10. Configure it with paused=false, but add a condition to your script (condition = APPLES == $(PEARS). Re-launch the revised mission. Since APPLES is equal to PEARS periodically, the condition will periodically be met.
6. Try changing the AppTick in one of the pXRelay configurations to 0.1. You should note that the script is now mostly in the state where its conditions are not met. Can you explain why?

3 Modify the pXRelay Code

[Note - this step is great to try if you have time. It's not essential in terms of continuity to the next lab. The most useful part of this step is to pull down the `moos-ivp-extend` tree and verify that you can build the tree with the `pXRelayTest` in your shell path.]

Next we will take the first steps toward extending the MOOS-IvP code. This is done by creating your own "third-party" tree, with your own MOOS apps. A template for this is available on the moos-ivp.org website: Go to www.moos-ivp.org and click on the `moos-ivp-extend` link in the lefthand menu. Download the `moos-ivp-extend` tree, in the same directory where the `moos-ivp` tree resides:

```
$ svn co https://oceanai.mit.edu/svn/moos-ivp-extend/trunk moos-ivp-extend
```

Verify that you can build this tree by:

```
 $ cd moos-ivp-extend
 $ ./build.sh
```

It should add the executable, `pXRelayTest`, in `moos-ivp-extend/bin/`.

Add the above directory to your shell path.

The moos-ivp-extend tree contains an app very similar to `pXRelay` from the moos-ivp tree. This app is call `pXRelayTest`. We will attempt to modify some of the behavior of the `pXRelay` application while putting our modifications in the new moos-ivp-extend tree, in `pXRelayTest`.

1. Take a look at the `pXRelayTest` source code. Look inside `Relayer.cpp`. Check out the `OnNewMail()` and `Iterate()` loops.
2. Can you modify this code to make it increment by ten instead of one?
3. Can you modify this code such that it takes *two* incoming variables, and does its thing if *either* of the two variables is written to?