Launching Processes and Running Mission Scripts with `Antler`

Paul Newman

May 5, 2008

Abstract

This document tells you how to use the application `pAntler` to launch multiple MOOS processes. This is usefultool for starting up a whole bunch of processes all of which share a single configuration file.

Contents

1 Synopsis

The process pAntler is used to launch/create a MOOS community. It is simple to use and Post V7.0.2 very extensible.

One of the ideas underlying MOOS is the one mission file one mission paradigm. A single mission file contains all the information required to configure all the processes needed to undertake the task (mission) in hand [1]. Note a collection of MOOS processes is commonly referred to as a "community".

1.1 Basic Syntax

Antler provides a simple and compact way to start a MOOS mission. For example if the desired mission file is *Mission.moos* then executing

```
pAntler Mission.moos
```

will launch the required processes/community for the mission

It reads from its configuration block **(which is declared as** ProcessConfig=ANTLER **)** a list of process names that will constitute the MOOS community. Each process to be launched is specfied with a line with the general syntax

Run = *procname* [@ LaunchConfiguration] [∼ MOOSName]

where LaunchConfiguration is an optional comma separated list of "parameter=value" pairs which collectively control how the process "procname" (for example iGPS, or iRemote or MOOSDB) is launched. Exactly what parameters can be specified is detailed later in the document.

Antler looks through its entire configuration block an launches one process for every line which begins with RUN= . When all processes have been launched Antler waits for all of them to exit and then quits itself.

2 Controlling Process Launch

Immediately after the "@" symbol in a RUN directive the user can supply a list of "parameter=value" pairs (comma separated) which control how the process in question should be launched. The following subsections will explain the action of available parameters.

2.0.1 Launching Processes in new console windows (or not)

```
Run = MOOSDB @NewConsole = true
```

The optional NewConsole parameter specifies whether the named process should be launched in a new window (an xterm in Unix or cmd-prompt in Win32 derived platforms). By default a new console is launched.

[1] And the pLogger application backs up each mission file so you know exactly what mission file was run at the time data was recorded — see pLogger documentation

2.1 Controlling Console Appearance

Post V7.0.2 releases allow a good deal of control over the appearance of the windows in which processes will be launched. Especially so on the 'nix side of life [2]

By specifying `XConfig=<Name>` or `Win32Config=<Name>` (depending on OS) the user can have `Antler` apply customisations to the new console in a process is launched. For example:

```
Run = MOOSDB @NewConsole = true, XConfig=DBXConsoleSettings, Win32Config=
    DBW32ConsoleSettings
```

will cause `Antler` to search through its configuration block to find a line which begins with `DBXConsoleSettings =` . or `DBW32ConsoleSettings =` – depending on OS). What is is the left of the equality determines the appearance of the new console and is a function of the host operating system.

2.1.1 Console Appearance in Unix like OS's

In unix derived operating systems the appearance string (referenced by "XConfig") is a comma separated list of parameters that would be used to configure an xterm. So continuing by way of the DBConsoleSettings example. If the DBConsoleSettings was specified (on its own line) as follows

```
DBXConsoleSettings = -bg, \#FF0000,-fg,\#FFFFFF,-geometry,80x12+2+00,+sb,-
    T,TheMOOSDB
```

then the MOOSDB would be launched in 12 rows by 80 columns window, white text on red at the top left of teh screen. The string "TheMOOSDB" would appear in the title. Note any xterm configuration parameters can specified in this way. See the manual page for xterm for information on the options allowed.

2.1.2 Console Appearance in Win32 OS's

The only native WIN32 console options supported control the background color of the terminal (text is always white). The LHS of the configuration line (referenced by "Win32Config") can contain a comma separated list of `BACKGROUND_RED` `BACKGROUND_BLUE` and `BACKGROUND_GREEN` . In this way

```
DBW32ConsoleSettings = BACKGROUND_RED
```

would produce a white on red win32 console.

2.1.3 Appearance Example

```
ProcessConfig = Antler
{
        //look on system path
        ExecutablePath = system
```

[2] the native win32 console has less flexibility than the xterm. Deep apologies for Win32 users who may feel hard done by by the asymmetry here

```
        //launch a DB
        Run = MOOSDB  @NewConsole = true,XConfig=DBXConsoleSettings,
            Win32Config=DBW32ConsoleSettings

          //xterm configuration for DB
          DBXConsoleSettings = -bg, \#FF0000,-fg,\#FFFFFF,-geometry,80x12
              +2+00,+sb,-T,TheMOOSDB

          //Win32 Configuration for DB
          DBW32ConsoleSettings = BACKGROUND_RED
}
```

2.2 Controlling Search Paths

Post V7.0.2 `Antler` offers extended functionality regarding specifying how executables are located on the host file system. The paths which you wish the OS to use when searching for executable to launch can be specified globally (a common path for all processes) or on a process by process basis.

2.2.1 Specifying Global Executable Paths

Adding line of the form

`ExecutablePath` = *path*

to Antler's configuration block where *path* is a suitable path string, will make `Antler` search in that place for the exectables to launch. Not specifying this variable or setting path to "SYSTEM" will cause Antler to relie on the host OS being able to locate the executable in its own executable paths.

2.2.2 Specifying Paths for an Individual Process

The global executable path (default "system") can be overridden for a particular process by providing your prefered path in the "RUN" directive line. For example

```
Run = pP1          @  NewConsole = true,path=/usr/strangeplace
```

will try to launch a process called "pP1" from a the directory "/usr/strangeplace". Such process specific path directives overide any path set with `ExecutablePath=...` (Section 2.2.1

3 Passing Parameters to Launched Processes

3.1 The Two Default Parameters

Unless told otherwise (see **??** each process launched is passed the mission file name as a command line argument and also the name it should use to register with the MOOSDB. This means that by default `argv[1]` of `main` is the name of the mission file currently in play (the one which `pAntler` is itself

reading) and argv[2] is the name of the process in to be launched (for example `iGPS` or `pLogger`). By default pAntler assumes the name which a process will be registering with the MOOSDB with is the name of the process itself. For example `pLogger` will register with the MOOSDB with the name "pLogger". However this can be changed using the ∼ `MOOSName` syntax:

```
Run = iGPS @NewConsole = true ~ GPS_A
```

will cause the executable called "iGPS" to be launched in a new console but (because iGPS handles command line parameters appropriately) it will register with the MOOSDB under the name of "GPS_A".

3.1.1 Handling default parameters

Of course just passing the MOOSName to a process doesn't mean automagically that all MOOS connections within that process will use this name. Supporting code must be provided. Listing 1 illustrates just one way in which this can be done.

Listing 1: Handling default command line parameters. Note how the MOOSName and Mission file are passed to the CMOOSApp derived object.

```
#include "SimpleApp.h"

//simple "main" file which serves to build and run a CMOOSApp-derived
    application


int main(int argc ,char * argv[])
{
    //set up some default application parameters

    //whats the name of the configuarion file that the application
    //should look in if it needs to read parameters?
    const char * sMissionFile = "Mission.moos";

    //under what name shoud the application register with the MOOSDB?
    const char * sMOOSName = "MyMOOSApp";

    switch(argc)
    {
    case 3:
        //command line says don't register with default name
        sMOOSName = argv[2];
    case 2:
        //command line says don't use default "mission.moos" config file
        sMissionFile = argv[1];
    }

    //make an application
    CSimpleApp TheApp;

    //run forever pasing registration name and mission file parameters
```

```
    TheApp.Run(sMOOSName, sMissionFile);

    //probably will never get here..
    return 0;
}
```

4 Running Multiple Instances of a Particular Process

As already described in Section 3, the optional `MOOSName` parameter allows MOOSProcesses to connect to the `MOOSDB` under a specified name. Why is this useful? Well for example a vehicle may have two GPS instruments onboard. Now by default `iGPS` may register it existence with the `MOOSDB` under the name "iGPS". This name is now taken and no other MOOSClient can use the name "iGPS"[3]. By using the ~ syntax multiple instances of the executable `iGPS` can be run but with each connecting to a the `MOOSDB` using a different name. For example

```
Run = iGPS @ NewConsole = true ~iGPSA
Run = iGPS @ NewConsole = true ~iGPSB
```

would launch two instances of `iGPS` registering under "iGPSA" and "iGPSB" respectively. **Note there would need to be *two* GPS configuration blocks in the mission file – one for each and the process names (RHS of `ProcessConfig=`) would be "iGPSA" and "iGPSB"**

4.1 Customising the Command Line Parameters Passed to a Launched Process

But what if your beloved new process which you desire `Antler` to launch requires extra command line configuration? Or what if you don't want `Antler` to pass the Mission file name and the MOOS name in a parameters? Fear not, just read on.

4.1.1 Specifying Additional Process Command Line Parameters

You can specify additional parameters which should be passed to a launched process using a syntax similar to that used to specify console appearance (see Section 2.1) The trick is to specify the name of a parameter string (R.H.S of a `ExtraProcessParams=...` in the process's `RUN` directive line. `Antler` the rescans its configuration block looking for this named string which must be a comma separated list of parameters. An example will make this blindingly obvious.

```
ProcessConfig = Antler
{
        Run = iProcA @ NewConsole = true,path=/usr/local/bin,
            ExtraProcessParams=ProcAParams
```

[3] if they try the MOOSDB will not accept them into the fold

```
        ProcAParams =-o,--verbose,--clever
}
```

The above would launch a process called ïProcA" in a new console, (with default appearance as no appearance string is specified see Section 2.1), and the process will be passed **six** parameters at launch time:

argv[0] the executable image name.

argv[1] the mission file name

argv[2] the process's MOOS name

argv[3] -o

argv[4] –verbose

argv[5] –clever

4.1.2 Inhibiting default parameters and Launching Arbitrary (non-MOOS) Processes

If you want to launch a process with `Antler` that has not been designed to handle the mission file name an MOOS name as the first two parameters passed in the command line then it is possible to tell `Antler` not to pass these parameters. This is done using the `InhibitMOOSParams` key word. For example if you wanted to launch the executable `top` in its own window you would use configuration similar to that in Listing 2

Listing 2: Launching a non moos process -like top (here on a 'nix system). Note the use of `InhibitMOOSParams`

```
ProcessConfig = Antler
{
Run = top @ NewConsole = true,path=system, InhibitMOOSParams=true, XConfig
    = TopX, ExtraProcessParams=TopParams

 //xterm configuration
 TopX = -bg, #F00000,-fg,#FFFFFF,-geometry,80x40+200+300,+sb,-T,top_top

  //list of parameters
  TopParams =-o,cpu
}
```

5 Distributing a Community over Multiple Machines

Up until now we have implicitly assumed that all processes launched by a single instance of `Antler` reside on the same physical computer. Surely this is conflicts with the idea that any MOOS process can run on any machine under any (common) OS? You're right it does and this issue has been addressed in post V7.0.2 versions.

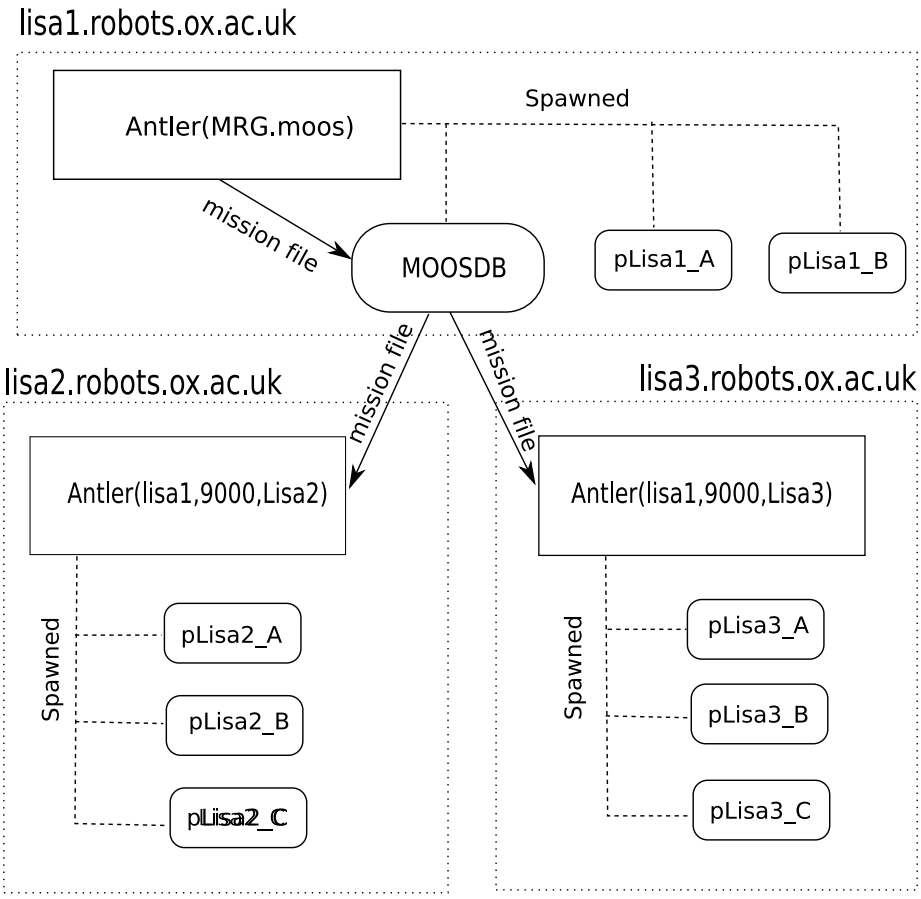

Figure 1: Antler can be started in one of two ways. Here on the machine lisa1.robots.ox.ac.uk it is started in "Top MOOS" mode with the name of a mission file on the command line. On the two other machines (lisa2 and lisa3) Anterl is started in headless mode receiving three command line parameters - the machine name on which a `MOOSDB` can be found, the port that `MOOSDB` is serving on and an Antler name which in this case is simply set to the machine name. When the "topMOOS" has spawned its processes it pushes the mission file to the DB. The headless Antlers pick up this notification and run themselves from the newly received Mission file. They only launch processes which are specified to run on Antlers with their own Antler name.

6 Application note : I/O Redirection - Deployment

Frequently `iRemote` , displayed on a remote machine, will be the only interface a user has to the MOOS community. We must ask the question - "where does all the IO from other processes go to prevent I/O blocking?". One answer to this is I/O redirection and backgrounding MOOS processes - a simple task in unix derived systems [4]/

Running `Antler` in the following fashion followed by a manual start up of `iRemote` is the recommended way of running MOOS in the field on a 'nix platform.

1. `./pAntler` *mission.moos* > ptyZ0 > /dev/null &
2. `./iRemote` *mission.moos*

This redirection of `iRemote` is encapsulated in the `moosbg` script included with the MOOS installations. In the case of an AUV the interface can only be reached through in-air wireless communications, which will clearly disappear when the vehicle submerges but will gracefully re-connect when surfacing(not so easy to do with a PPP or similar link).

[4] some OS are good for development others for running...