# Lab #7
# Simulation of Multi-Vehicle Operations

2.S998 Unmanned Marine Vehicle Autonomy, Sensing and Communications

## Contents


**1 Overview and Objectives 3**
- 1.1 Preliminaries . . . 3
- 1.2 More MOOS / MOOS-IvP Resources . . . 3
- 1.3 The Shoreside (Topside) / Vehicle Topology . . . 4

**2 Experimenting with pMOOSBridge 5**
- 2.1 Revisiting the pXRelay Example with pMOOSBridge . . . 5
- 2.2 Revisiting the Alpha Mission with pMOOSBridge . . . 6
- 2.3 Assignment: Converting the Alpha Mission into a Two-Vehicle Mission . . . 7

**3 Using uField Toolbox Modules and Multi-Vehicle Path Planning 9**
- 3.1 Building the Baseline Mission . . . 9
- 3.2 Assignment: Distributed Traveling Salesman . . . 10
- 3.3 Due Date . . . 12

# 1 Overview and Objectives

In today's lab we will begin shifting our focus to autonomy configurations involving multiple vehicles. Our first focus will be on communications between two independently running MOOSDBs, where typically a single MOOSDB, or "MOOS Community" is associate with a single vehicle. Ultimately the inter-MOOSDB or inter-vehicle communication may come over a acoustic modem link or a satellite link, our primary initial focus is on communications over an internet connection, even if the multiple "nodes" are all running on your one laptop.

Gaining familiarity with this mode of operation will be essential for later labs and operation of vehicles on the water.

## 1.1 Preliminaries

Standard practice now before any lab should be to perform an svn update within your moos-ivp tree and rebuild if you see any updates pulled down from the SVN server. 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
```

### Where to Build and Store Lab Missions

As with previous labs, we will use your version of the `moos-ivp-extend` tree, which by now you may have re-named something like `moos-ivp-jsmith`, where your email may be `jsmith@mit.edu`. In this tree, there is a missions folder:

```
$ cd moos-ivp-jsmith
$ ls
CMakeLists.txt  bin/         build.sh*    docs/        missions/    src/
README          build/       data/        lib/         scripts/
```

For each distinct exercise in this lab, there should be a corresponding subdirectory in the missions folder, typically with both a `.moos` and `.bhv` configuration file.

## 1.2 More MOOS / MOOS-IvP Resources

The one document that may be new to the discussion in this lab is the documentation for the uField Toolbox linked below. See also the slides from today's lecture.

- The uField Toolbox Documentation
  http://oceanai.mit.edu/moos-ivp-pdf/moosivp-ufield.pdf

- See the slides from the today's class which give a bit more background into marine autonomy and the IvP Helm.

- The pMOOSBridge documentation.

  The IvP Helm documentation.
  http://oceanai.mit.edu/moos-ivp-pdf/moosivp-helm.pdf

  The moos-ivp.org website documentation.
  http://www.moos-ivp.org

### 1.3 The Shoreside (Topside) / Vehicle Topology

The layout of interconnected MOOS communities used in this lab is depicted in the figure below. This layout will be used for the remainder of the course, including during operations on the river.

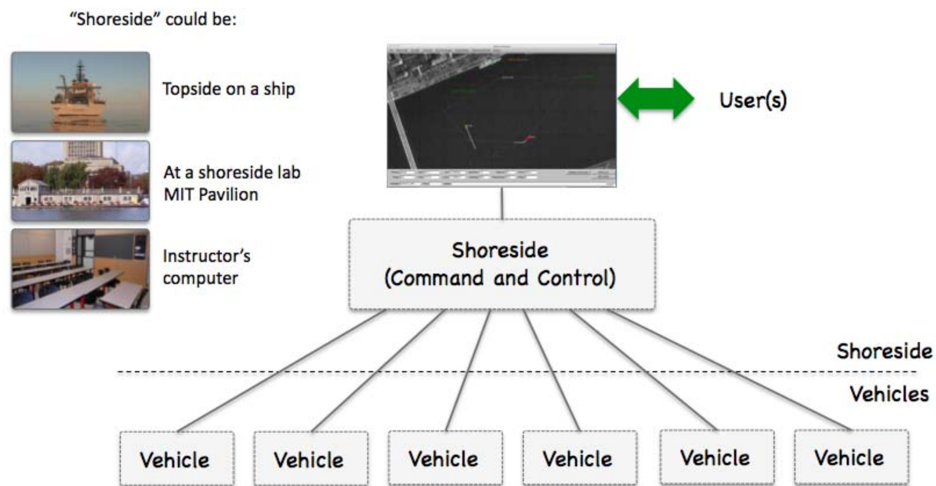


Figure 1: **Shoreside to Multi-Vehicle Topology:** A number of vehicles are deployed with each vehicle maintaining some level of connectivity to a shoreside command and control computer. Each node (vehicles and the shoreside) are comprised of a dedicated MOOS community. Modes and limits of communication may vary.

# 2 Experimenting with pMOOSBridge

In the first exercise in today's lab, the goal is to become familiar with pMOOSBridge. In these first couple of missions, we will be configuring pMOOSBridge explicitly in our .moos files. In later missions, we will be using components of the uField Toolbox to automatically configure the bridge configurations.

As the very first step, we begin by updating the moos-ivp tree and making sure we have the very latest build from the class SVN branch:

```
% cd moos-ivp
% svn update
Updated to revision 4205.
% ./build-ivp.sh
```

Assuming there were no problems with the update and build, we're ready to go.

## 2.1 Revisiting the pXRelay Example with pMOOSBridge

In this part we will:

- Prepare a copy of the xrelay mission for experimenting.
- Create two .moos files to launch two MOOSDBs.
- Launch the two communities and confirm that bridging works.

### 2.1.1 Make a copy of the xrelay mission

In this lab we will mostly be building and modifying mission files rather than writing source code. The first step is to copy the xrelay example mission from your moos-ivp-extend tree in your own moos-ivp-extend tree. This tree by now should be re-named something like `moos-ivp-jsmith` where your MIT email is `jsmith@mit.edu`. As with lab 6, this directory is what you will be submitting in this problem set.

```
% cp -rp moos-ivp/ivp/missions/xrelay  moos-ivp-jsmith/missions/lab07/xrelay
```

### 2.1.2 Split the xrelay mission up into two separate MOOS communities

The xrelay mission, as originally configured, has three essential apps launched. The MOOSDB, and two pXRelay apps. In this experiment, you will create two .moos files, each launching a MOOSDB and one pXRelay app. You will also need a pMOOSBridge app launched in each community.

*You will need to configure each .moos file with a unique community name, and port. Note each of those pieces of information since you will need it in the configuration of pMOOSBridge in both files.*

Take a look at the pMOOSBridge document linked on the course website under the reading section. Your pMOOSBridge configuration block will look something like below. *Note that the port on which pMOOSBridge is listening is not the same port on which the MOOSDB is serving.* Remember that

the same bridge configuration will not work for both sides. They must complement each other, in terms of the variable name being bridge, the UDPListen config, the destination host IP and port number.

```
ProcessConfig = pMOOSBridge_apples
{
  AppTick    = 4
  CommsTick  = 4

  UDPListen = 9201
  BridgeFrequency = 0

  UDPSHARE = [APPLES]   -> xrelay_apples @ localhost:9200 [APPLES]
}
```

#### 2.1.3 Launch the two communities and confirm that bridging works

Using perhaps two separate terminal windows, launch both MOOS communities and confirm that the modified xrelay mission works as before. You will still need to poke one of the communities to get things going. Confirm things are working by opening a scope in one of the communities. Note that "Community" field on scoped items now. You should see postings from both the local community, and the second remote community.

If things are not working, consider the following trouble-shooting points:

- Make sure the UDPListen port on side matches the destination port on the other.
- Make sure you give each pMOOSBridge app a distinct name.
- Try just poking with uPokeDB to test the bridging.

### 2.2 Revisiting the Alpha Mission with pMOOSBridge

In this part we will:

- Prepare a copy of the alpha mission for experimenting.
- Create two .moos files to launch two MOOSDBs.
- Launch the two communities and confirm that bridging works.

#### 2.2.1 Make a copy of the alpha mission

The first step is to copy the alpha example mission from the moos-ivp class tree into your own moos-ivp-extend tree.

```
  % cp -rp moos-ivp/ivp/missions/s1_alpha  moos-ivp-jsmith/missions/lab07/alpha
```

#### 2.2.2 Split the alpha mission up into two separate MOOS communities

In this step you will create two separate MOOS communities: a *shoreside* community and an *alpha* vehicle community, by creating two separate .moos files. In the shoreside community there will

be a MOOSDB and pMarineViewer. In the alpha community will be a MOOSDB and everything but pMarineViewer. In both communities you will also need to add a pMOOSBridge configuration block and add pMOOSBridge to the Antler configuration block.

The primary challenge here is to consider which variables to configure for bridging in each direction. One hint is that, from the vehicle to the shoreside you will need to bridge the `NODE_REPORT`. This is the message generated from pNodeReporter containing much of the vehicle state, and used by pMarineViewer to render the vehicle. It is generated locally on the vehicle as `NODE_REPORT_LOCAL` and should arrive in the shoreside as `NODE_REPORT`. The configuration in alpha's pMOOSBridge config block will look like:

```
UDPSHARE = [NODE_REPORT_LOCAL]   -> shoreside @ localhost:9000 [NODE_REPORT]
```

You will also want to bridge the `VIEW_SEGLIST` and `VIEW_POINT` variables to enable pMarineViewer on the shoreside to have the visual feedback of the vehicle waypoints.

### 2.2.3 Launch the two communities and confirm that bridging works

Using perhaps two separate terminal windows, launch both MOOS communities and confirm that the modified Alpha mission works as before. You still should be able to deploy and return the vehicle with the buttons in pMarineViewer. Consider what is being poked when hitting those buttons. (You can always find this out by looking at the pMarineViewer configuration block in the mission file). Make sure those variables are being properly bridged to the alpha vehicle community.

If things are not working, consider the following trouble-shooting points:

- Make sure the UDPListen port on side matches the destination port on the other.
- Make sure you give each pMOOSBridge app a distinct name.
- Try just poking with uPokeDB to test the bridging.
- You should see a vehicle in the pMarineViewer window even before deploying the vehicle. If you don't check the pMOOSBridge configuration on the vehicle side.

## 2.3 Assignment: Converting the Alpha Mission into a Two-Vehicle Mission

In this part we will:

- Prepare a copy of the previous modified alpha mission for experimenting.
- Create a third .moos file, bravo.moos, to launch another simulated vehicle.
- Launch the three communities and confirm that bridging works, and deploy and return commands work for both vehicles with a single pMarineViewer button click.

### 2.3.1 Make a copy of the previous two-vehicle alpha mission

The first step is to copy the alpha example mission from the previous exercise in Section 2.2. This mission directory will be handed in, so we will be very explicit about the naming. The file structure should be:

```
moos-ivp-jsmith/
   missions/
      lab07one/
         shoreside.moos
         alpha.moos
         bravo.moos
         launch.sh
         README.txt
```

You should create and submit a script for launching all three communities. If there is anything special the user should know to run the mission, then also include a `README.txt` file in your tree. An example shell script for launching multiple missions is posted on the class wiki page under lab updates.

#### 2.3.2 Create a new bravo.moos file for a simulating a second vehicle

In this step you will create a third mission file, `bravo.moos`, for simulating a second vehicle. In the `bravo.moos` file, you will need to configure it with a distinct community name, e.g., bravo, distinct port number, and distinct port number for UDPListen in the pMOOSBridge configuration block.

A `bravo.bhv` file will also need to be created for this vehicle. The behavior mission is not the point of focus here, so just create a waypoint survey mission similar to alpha's with the vertices shifted 50 meters to the east.

The `shoreside.moos` will also need to be altered to bridge the DEPLOY and RETURN commands out to both vehicles with a single button click.

#### 2.3.3 Launch the three communities and confirm that things work

Your final mission configuration should meet the following criteria:

1. You should be able to launch both vehicles and the shoreside community with a single shell script.
2. You should be able to deploy and return both vehicles with a single button click in pMarineViewer.
3. The vehicles and the mission waypoints for both vehicles should be viewable in pMarineViewer.
4. Your pMarineViewer should also be configured to deploy or return a single chosen vehicle in isolation. Hint: use the ACTIONS parameter in pMarineViewer to add deploy-alpha, return-bravo etc. capability in the Action pull-down menu.

# 3 Using uField Toolbox Modules and Multi-Vehicle Path Planning

In the second exercise in today's lab, the goal is to create an autonomy mission that uses a few modules in the uField Toolbox to replace some of the pMOOSBridge configuration steps used in the previous exercise. Our end goal is a two vehicle simulation that should be easily scalable to a larger number of vehicles.

Once we have the multi-vehicle simulation established, our goal will be to build a mission where two simulated vehicles are receiving points in the x-y plane, from the shoreside community, and traversing those points in a shortest-path trajectory.

## 3.1 Building the Baseline Mission

In this part we will:

- Copy the baseline mission from the moos-ivp missions folder.
- Note the structure of the launch script and the nsplug setup.
- Note the roles of uFldShoreBroker, uFldNodeBroker, and pHostInfo.
- Extend the mission such that its loiter is periodically interrupted by a return to (0,0) where it must wait 30 seconds before being free to return.

**Copy the baseline mission from the moos-ivp tree**

Start by copying a baseline version of the mission from the moos-ivp tree:

```
cp -rp moos-ivp/ivp/missions/lab7baseline moos-ivp-jsmith/missions/lab7
```

Confirm that this mission launches properly, typing `./launch.sh 10` from the command-line in this directory. You should see two vehicles appear on the screen. Deploy them by hitting the deploy button.

**Understand the launch structure**

Before moving on, take a look at how things are launched. See if you can understand what is going on inside the launch script. Note that the script is building the target .moos files by invoking an application called `nsplug`. This tool is a bit like the cpp pre-processor. You can learn a bit more about nsplug by:

```
nsplug -h
nsplug -m | less
```

Note that the `targ_*` files are generated automatically each time the launch script is invoked. If you edit these files, the changes will be lost the next time you launch!

Note that the launch script defines certain variables such as the vehicle name, MOOSDB port etc., and passes this info into nsplug for expansion.

### Understand the uField Toolbox operations

Before moving on, take a look at the relationship between pHostInfo, uFldNodeBroker and uFldShoreBroker. Note how they handle pMOOSBridge configuration for you on both ends. From this point forward, using these tools, your configuration of bridge variables should be handled in this way. Please read the sections on these three applications in the uField Toolbox documentation on the course website.

For now, beginning with this baseline mission, bridge configuration will just work. But you *will* want to augment what is being bridged for later steps in this lab, so try to understand how bridge configuration is handled in the uFldNodeBroker and uFldShoreBroker modules.

### Augment the behavior configuration

Augment the behavior configuration such that the vehicle never spends more than 10 minutes away from its starting point, and that each time it returns to its starting point it spends at least one minute there before resuming its mission. For simplicity, you can disregard the time it takes to actually return to the start position in the 10 minute rule.

## 3.2 Assignment: Distributed Traveling Salesman

In this part we will:

- Create a uTimerScript script to generate 100 random points in a specified region.
- Create a `pPointAssign` MOOS module on the shoreside to send half the points to one vehicle and half to the other.
- Create `pGenPath` MOOS module on the vehicle side. It should subscribe for the points coming from `pPointAssign` on the shoreside, generate a sequence of waypoints, and then send them to the vehicle awaiting the points.
- Configure an autonomy mission on the vehicle to traverse the points while periodically returning to home.

### Create a uTimerScript Script

Configure a uTimerScript script running on the shoreside community, to generate a random sequence of points within a region of the operation area. This region should be:

```
-25,-25
-25,-175
200,-25
200,-175
```

The script should generate 100 postings to the MOOSDB of the form:

```
VISIT_POINT = x=8,   y=9,  id=1
VISIT_POINT = x=33,  y=29, id=2
...
VISIT_POINT = x=-11, y=-9, id=100
```

The last posting in the script should be VISIT_POINT="last". This is the cue to recipients that it my regard the previous point to be the last one to expect.

### Create new shoreside pPointAssign MOOS app for distributing points

Create a new pPointAssign MOOS module on the shoreside to send half the points to one vehicle and half to the other. The module should have a flag that allows the dividing to be based on a region, or an every-other-one algorithm.

Your app should subscribe to the VISIT_POINT output of the timer script and produce output along the lines of:

```
VISIT_POINT_HENRY = x=8,   y=9,   id=1
...
VISIT_POINT_HENRY = x=33,  y=29,  id=50
VISIT_POINT_GILDA = x=19,  y=111, id=51

VISIT_POINT_GILDA = x=-11, y=-9,  id=100
```

You also need to bridge these variables out to their respective vehicles. For example you may want to bridge VISIT_POINT_HENRY out to vehicle *henry* with the arrival name VISIT_POINT. Remember that you should handle the bridge configuration with uFldShoreBroker now - not pMOOSBridge.

The last posting posted by pPointAssign, bridged to the vehicles, should be VISIT_POINT_VNAME="last". This is the cue on the vehicle that it my regard the previous point to be the last one to expect. The component VNAME above should be replace by the vehicle name and there should be one such posting per vehicle.

### Create new vehicle MOOS app, pGenPath, for traversing points

Create a new pGenPath MOOS module for running on the vehicle, that takes as input a list of VIEW_POINT messages from the shoreside, and generates a waypoint list suitable for consumption by the waypoint behavior.

In your behavior configuration, use a Waypoint behavior with an UPDATES parameter specifying the MOOS variable used for incoming dynamic parameter configuration. In this case, the primary incoming configuration will be the list of waypoints from your pGenPath module. To make things a bit easier, your shortest path algorithm may simply be a greedy shortest path with an imaginary virtual initial point being the vehicle's present position.

In your pGenPath module you can use the following pseudo code as guidance for how to build a proper list of waypoints recognized by the waypoint behavior:

```
XYSegList my_seglist;
my_seglist.add_vertex(3, 8);
my_seglist.add_vertex(43, 99);
my_seglist.add_vertex(44, -111);

string update_str = "points = ";
updates_str      += my_seglist.get_spec();
m_Comms.Notify("UPDATES_VAR", update_str);  // UPDATES_VAR depends on your config
```

The waypoint behavior will generate a series of messages similar to:

```
WPT_HIT = x=88,y=102
WPT_HIT = x=120,y=-11
...
WPT_HIT = x=10,y=31
```

In your `pGenPath` module, keep a list of which points have been visited and which have not. When the list changes, publish a status variable:

```
PATH_STATUS = total_visited=7, total_unvisited=19, vname=henry
```

Bridge this variable back to the shoreside.

When you configure this `pGenPath` module to run with your vehicle, create a new `plug_pGenPath.moos` component similar to the other plugs, and create a new `#include` for this plug in the `meta_vehicle.moos` file.

Bonus: In your greedy algorithm, you may want to consider the relative angle and position of a next waypoint given the current and previous waypoint. In some cases the vehicle will simply not be able to turn sharply and quickly enough to get to the next point. In those situations, you are welcome to regard such a point as "not reachable" from the previous point, and go on to consider the next closest point that *is* reachable.

## 3.3 Due Date

You should hand in your assignment by again uploading a single compressed file to Stellar containing your `moos-ivp-jsmith` tree. It should contain two new MOOS apps, `pPointAssign` and `pGenPath`, and two mission directories: `lab07one`, and `lab07two`.

The due date is one week from today: March 8th 2012. 6pm.

MIT OpenCourseWare
http://ocw.mit.edu

2.S998 Marine Autonomy, Sensing and Communications
Spring 2012

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.