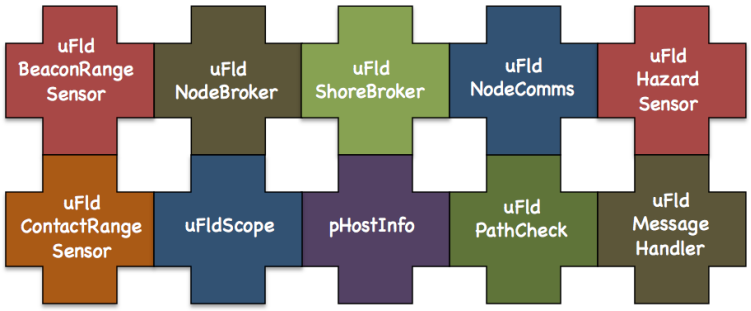
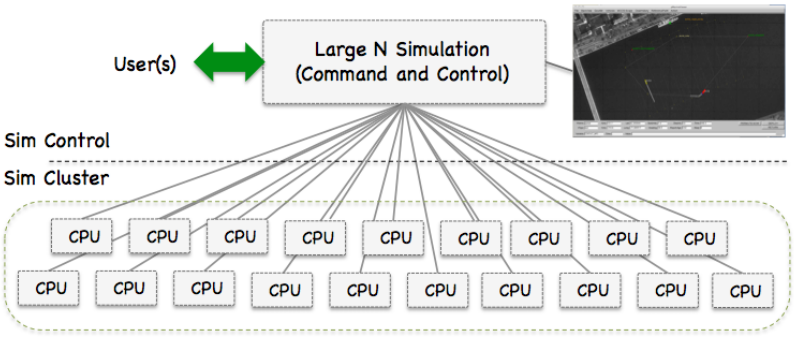
The MOOS-IvP uField Toolbox for Multi-Vehicle Operations and Simulation

Release 12.2





Michael R. Benjamin
Department Mechanical Engineering
Massachusetts Institute of Technology, Cambridge MA

March 13, 2012 - Release 12.2

Abstract

The uField Toolbox is a set of MOOS applications to facilitate the operation and simulation of multiple unmanned marine vehicles. It includes tools for automating the communication between vehicles and a shoreside command and control MOOS community (`uFldNodeBroker`, `uFldShoreBroker` and `pHostInfo`). A tool for handling inter-vehicle communications while simulating range dependencny and bandwidth limitations (`uFldNodeComms`). A tool for handling incoming messages on a vehicle (`uFldMessageHandler`). A tool for shoreside scoping on information gathered about multiple fielded vehicles (`uFldScope`). A tool for simulating a shallow-water hazard sensor (`uFldHazardSensor`). A tool for simulating a range only sensor to fixed beacon locations (`uFldBeaconRangeSensor`). And a tool for simulating a range only sensor to other moving contacts (`uFldContactRangeSensor`).

This work is graciously supported under a multi-year Battelle subcontract #251192 entitled "Multi-Vehicle Cooperative and Adaptive Autonomy Algorithms for Unmanned Marine Vehicles". Battelle Program Manager, Robert Carnes, MD.

Contents

1 Overview **7**
- 1.1 Purpose and Scope of this Document . . . 7
- 1.2 Motivations for the uField Toolbox . . . 8
- 1.3 Brief Background of MOOS-IvP . . . 9
- 1.4 Sponsors of MOOS-IvP . . . 10
- 1.5 The Software . . . 10
  - 1.5.1 Building and Running the Software . . . 10
  - 1.5.2 Operating Systems Supported by MOOS and IvP . . . 11
- 1.6 Where to Get Further Information . . . 11
  - 1.6.1 Websites and Email Lists . . . 11
  - 1.6.2 Documentation . . . 12

2 The uFldNodeBroker Utility: Brokering Node Connections **14**
- 2.1 Overview of the uFldNodeBroker Interface and Configuration Options . . . 15
  - 2.1.1 Configuration Parameters of uFldNodeBroker . . . 15
  - 2.1.2 MOOS Variables Published by uFldNodeBroker . . . 15
  - 2.1.3 MOOS Variables Subscribed for by uFldNodeBroker . . . 15
  - 2.1.4 Command Line Usage of uFldNodeBroker . . . 16
  - 2.1.5 An Example MOOS Configuration Block . . . 16
- 2.2 Terminal Output - A Detailed Snapshot of the Broker Status . . . 17

3 The uFldShoreBroker Utility: Brokering Shore Connections **19**
- 3.1 Overview of the uFldShoreBroker Interface and Configuration Options . . . 20
  - 3.1.1 Configuration Parameters of uFldShoreBroker . . . 20
  - 3.1.2 MOOS Variables Published by uFldShoreBroker . . . 20
  - 3.1.3 MOOS Variables Subscribed for by uFldShoreBroker . . . 20
  - 3.1.4 Command Line Usage of uFldShoreBroker . . . 21
  - 3.1.5 An Example MOOS Configuration Block . . . 21
- 3.2 Bridging Variables Upon Connection to Nodes . . . 22
  - 3.2.1 Bridging with pMOOSBridge . . . 22
  - 3.2.2 Handling a Valid Incoming Ping from a Remote Node . . . 23
- 3.3 Usage Scenarios the uFldShoreBroker Utility . . . 25
- 3.4 Terminal Output . . . 25

4 The pHostInfo Utility: Detecting and Sharing Host Info **27**
- 4.1 Overview of the pHostInfo Interface and Configuration Options . . . 28
  - 4.1.1 Configuration Parameters of pHostInfo . . . 28
  - 4.1.2 MOOS Variables Published by pHostInfo . . . 28
  - 4.1.3 MOOS Variables Subscribed for by pHostInfo . . . 28
  - 4.1.4 Command Line Usage of pHostInfo . . . 28
  - 4.1.5 An Example MOOS Configuration Block . . . 29
- 4.2 Usage Scenarios the pHostInfo Utility . . . 29
- 4.3 Handling Multiple IP Addresses . . . 29
- 4.4 A Peek Under the Hood . . . 30

4.4.1 Temporary Files . . . 30
4.4.2 Possible Gotchas . . . 30

5 The uFldNodeComms Utility: Simulating Intervehicle Comms 31
5.1 Overview of the uFldNodeComms Interface and Configuration Options . . . 32
5.1.1 Configuration Parameters of uFldNodeComms . . . 32
5.1.2 MOOS Variables Published by uFldNodeComms . . . 32
5.1.3 MOOS Variables Subscribed for by uFldNodeComms . . . 33
5.1.4 Command Line Usage of uFldNodeComms . . . 33
5.1.5 An Example MOOS Configuration Block . . . 33
5.2 Handling Node Reports . . . 34
5.2.1 The Criteria for Routing Node Reports . . . 34
5.2.2 Node Report Transmissions and pMOOSBridge . . . 36
5.3 Handling Node Messages . . . 37
5.3.1 The Criteria for Routing Node Messages . . . 37
5.4 Visual Artifacts for Rendering Inter-Vehicle Communications . . . 38
5.5 Status Updates to the Terminal . . . 39

6 The uFldMessageHandler Utility: Handling Node Messages 41
6.1 Overview of the uFldMessageHandler Interface and Configuration Options . . . 42
6.1.1 Configuration Parameters of uFldMessageHandler . . . 42
6.1.2 MOOS Variables Published by uFldMessageHandler . . . 42
6.1.3 MOOS Variables Subscribed for by uFldMessageHandler . . . 42
6.1.4 Command Line Usage of uFldMessageHandler . . . 42
6.1.5 An Example MOOS Configuration Block . . . 43
6.2 Terminal Output - A Detailed Snapshot of the Broker Status . . . 43

7 The uFldScope Utility: Multi-Vehicle Status Summary 45
7.1 Overview of the uFldScope Interface and Configuration Options . . . 46
7.1.1 Configuration Parameters of uFldScope . . . 46
7.1.2 MOOS Variables Published by uFldScope . . . 46
7.1.3 MOOS Variables Subscribed for by uFldScope . . . 46
7.1.4 Command Line Usage of uFldScope . . . 46
7.1.5 An Example MOOS Configuration Block . . . 46
7.2 Configuring the uFldScope Utility . . . 47
7.2.1 Configuring Scope Elements . . . 47
7.2.2 Configuring Scope Layouts . . . 48
7.2.3 Further Control of the Terminal Output . . . 49

8 The uFldPathCheck Utility: Checking Vehicle Path Properties 50
8.1 Overview of the uFldPathCheck Interface and Configuration Options . . . 50
8.1.1 Configuration Parameters of uFldPathCheck . . . 51
8.1.2 MOOS Variables Published by uFldPathCheck . . . 51
8.1.3 MOOS Variables Subscribed for by uFldPathCheck . . . 51
8.1.4 Command Line Usage of uFldPathCheck . . . 51
8.1.5 An Example MOOS Configuration Block . . . 52

8.2 Usage Scenarios the uFldPathCheck Utility . . . 52

9 The uFldHazardSensor: Simulating an Simple Hazard Sensor 53
9.1 The uFldHazardSensor Interface and Configuration Options . . . 54
9.1.1 Configuration Parameters of uFldHazardSensor . . . 55
9.1.2 MOOS Variables Published by uFldHazardSensor . . . 55
9.1.3 MOOS Variables Subscribed for by uFldHazardSensor . . . 56
9.1.4 Command Line Usage of uFldHazardSensor . . . 56
9.1.5 An Example MOOS Configuration Block . . . 57
9.2 Configuring the Hazard Field . . . 57
9.2.1 An Example Hazard Field . . . 58
9.2.2 Automatically Generating a Hazard Field . . . 59
9.3 Configuring the Possible Sensor Settings . . . 59
9.3.1 Sensor Swath Width Options . . . 59
9.3.2 Sensor ROC Curve Configuration Options . . . 60
9.3.3 Classification Configuration Options . . . 61
9.3.4 Dynamic Resetting of the Sensor . . . 62
9.3.5 Posting of Sensor Configuration Options . . . 62
9.4 Configuring the Simulator Visual Preferences . . . 62
9.4.1 Configuring the Sensor Field Swath Rendering . . . 63
9.4.2 Configuring the Hazard Field Renderings . . . 63
9.4.3 Configuring the Sensor Report Renderings . . . 63
9.5 Sensor Processing Algorithm . . . 64
9.6 Console Output Generated by uFldHazardSensor . . . 64
9.6.1 Console Output in the Non-Verbose Mode . . . 64
9.6.2 Console Output in the Verbose Mode . . . 65
9.7 The Jake Example Mission Using uFldHazardSensor . . . 66
9.7.1 What is Happening in the Jake Mission . . . 66

10 The uFldBeaconRangeSensor Utility: Simulating Vehicle to Beacon Ranges 69
10.1 The uFldBeaconRangeSensor Interface and Configuration Options . . . 70
10.1.1 Configuration Parameters of uFldBeaconRangeSensor . . . 70
10.1.2 MOOS Variables Published by uFldBeaconRangeSensor . . . 71
10.1.3 MOOS Variables Subscribed for by uFldBeaconRangeSensor . . . 71
10.1.4 Command Line Usage of uFldBeaconRangeSensor . . . 72
10.1.5 An Example MOOS Configuration Block . . . 72
10.2 Using and Configuring the uFldBeaconRangeSensor Utility . . . 73
10.2.1 Configuring the Beacon Locations and Properties . . . 74
10.2.2 Unsolicited Beacon Range Reports . . . 75
10.2.3 Solicited Beacon Range Reports . . . 76
10.2.4 Limiting the Frequency of Vehicle Range Requests . . . 76
10.2.5 Producing Range Measurements with Noise . . . 77
10.2.6 Console Output Generated by uFldBeaconRangeSensor . . . 78
10.3 Interaction between uFldBeaconRangeSensor and pMarineViewer . . . 80
10.4 The Indigo Example Mission Using uFldBeaconRangeSensor . . . 82

10.4.1 Generating Range Report Data for Matlab 83

11 The uFldContactRangeSensor Utility: Detecting Contact Ranges 84
11.1 The uFldContactRangeSensor Interface and Configuration Options 85
11.1.1 Configuration Parameters of uFldContactRangeSensor 85
11.1.2 MOOS Variables Published by uFldContactRangeSensor 86
11.1.3 MOOS Variables Subscribed for by uFldContactRangeSensor 86
11.1.4 Command Line Usage of uFldContactRangeSensor 86
11.2 Configuring the uFldContactRangeSensor Parameters 87
11.3 Limiting the Frequency of Vehicle Range Requests 88
11.4 Producing Range Measurements with Noise 88
11.5 An Example MOOS Configuration Block 89
11.5.1 Console Output Generated by uFldContactRangeSensor 89
11.6 Interaction between uFldContactRangeSensor and pMarineViewer 92
11.7 The Hugo Example Mission Using uFldContactRangeSensor 92

A Colors 96

1 Overview

1.1 Purpose and Scope of this Document

The uField Toolbox is a set of tools to facilitate the deployment of and simulation of multiple fielded marine vehicles. It pre-supposes the arrangment depicted in Figure 1 below. A number of vehicles are deployed and are connected to a single shoreside command and control computer. Each vehicle, as well as the shoreside computer, contain a dedicated MOOS community. A community is comprised of a MOOSDB process and a number of connected MOOS applications.

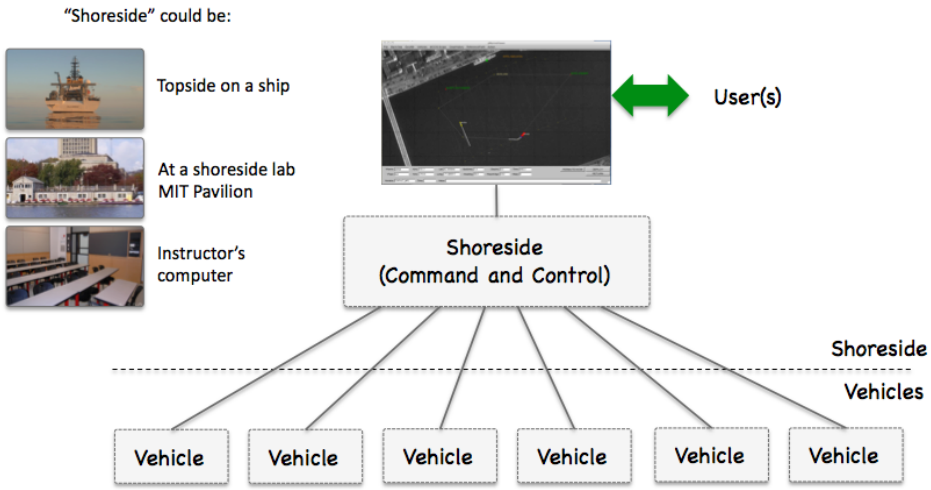


Figure 1: **Shoreside to Multi-Vehicle Topology:** A number of vehicles are deployed with each vehicle maintaining some level of connectivity to a shoreside command and control computer. Each node (vehicles and the shoreside) are comprised of a dedicated MOOS community. Modes and limits of communication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

The uField Toolbox is designed to support both laboratory simulations as well as fielded experiments with multiple marine vehicles.

The uField Toolbox in Laboratory Simulations

In laboratory or classroom simulations, the nodes in Figure 1 may be either (a) all running on a single user's machine during initial development and testing, or (b) each running on a separate laptop with the shoreside community rendered on a central laboratory screen.

The uField Toolbox in Fielded Experimentation

During field experiments, the vehicle nodes in Figure 1 are actual fielded vehicles. The shoreside node is a machine on the shore with connectivity to each of the vehicles. For example, at the MIT Sailing Pavilion lab, the vehicles are either kayaks, kingfishers or a man-portable UUV. Communications is typically achieved with WiFi, and occasionally over acoustic link, or cellular network link.

1.2 Motivations for the uField Toolbox

The uField Toolbox is meant to facilitate and simulate. It facilitates aspects of configuration for inter-vehicle communications with pMOOSBridge by letting nodes auto-configure with one another. It facilitates the monitoring of properties of several fielded or simulated vehicles by collecting and displaying user configurable information. It simulates properties inter-vehicle communication such as range dependency and bandwidth limitations. It simulates inter-vehicle contact detection as would otherwise occur through AIS or vehicle mounted sensors. And it simulates a range-only sensor to fixed beacon locations and moving contacts.

Facilitation of Inter-Vehicle Connections

When vehicles are on the internet, or common local subnet, their inter-vehicle communications are handled by `pMOOSBridge`. This is the case during multi-vehicle simulations on a single machine, multi-vehicle simulations between a room full of laptops, and field exercises at a facility like the MIT Sailing Pavilion when all vehicles are connected by a local WiFi router. It is even the case when vehicles are connected using the cellular telephone network. When or if a vehicle is connected via an acoustic modem, or via a satellite link, `pMOOSBridge` is replaced with a dedicated device interface.

A hurdle to using `pMOOSBridge` has been the configuration of `pMOOSBridge` on each machine to properly point the IP address and community name of the vehicles to which it wishes to communicate. If the IP address and vehicle/community names of all machines never change, this may be tractible but cumbersome. When the IP addresses are not known at run time, the configuration problem is often a deal-breaker. In conjunction with the development of the uField Toolbox, a key modification to `pMOOSBridge` was made to allow it to accept incoming mail specifying new bridge configurations. The `uFldNodeBroker` app may be run on each vehicle to communicate in a generic manner to the shoreside MOOS community running `uFldShoreBroker`. These two apps communicate to share IP and host information and request modifications to their own local `pMOOSBridge` instances, automatically configuring them for all further communications. All that is needed on the vehicles is a list of possible shoreside IP addresses to try for initial connections. Collectively these tools greatly simplify the `pMOOSBridge` comms configurations in both simulation and fielded exercises.

Facilitation of Multi-Vehicle Status Fields

Deploying multiple vehicles makes it increasingly important to have access to key information across all vehicles in a concise, user-configurable format. For example, it may be good to know which vehicles have had start-up problems, which vehicles are running low on power, or the total trip distance or maximum noted speed for all vehicles. The `uFldScope` tool is built to address this. It exploits a convention that many MOOS status messages are in the format of comma-separated parmeter=value pairs. The `uFldScope` tool may be configured to register for any such status message, provided a field to scope, and provided a field indicating the vehicle name. It produces a series of terminal-output reports in a simple tabular format with a row for each vehicle and column for each scoped piece of information. The tool may be configured to produce reports on multiple different sets of scoped data, allowing the user to toggle through the reports.

Simulating Inter-Vehicle Contact Detection

Detecting the position of another vehicle may be necessary (a) for performing collision avoidance, (b) for coordination with a collaborating vehicle, or (c) for stategizing against a competitor vehicle. Knowning another vehicle's position may come from either communication of vehicle position through a general system such as AIS, through direct vehicle to vehicle communication, or through on-board sensors uses to detect and track another vehicle.

The goal of the `uFldNodeComms` utility is simulate the above scenarios. It accepts node reports from *all* fielded or simulated vehicles, and passes on this information to other vehicles. The user may choose a configuration where every vehicle knows the position of every other vehicle all the time. Or a configuration may be chosen where vehicle report sharing is based on the vehicle range. The user may also choose a configuration where vehicle group or team membership determines, in part, the sharing of node reports.

Simulating Range-Only Sensors

A set of common sensing problems for marine vehicles include (a) determining the location of a fixed object given only a sequence of range-only measurements, (b) determining the location of one's own vehicle given range-only measurements to an object with a known location, and (c) determining the location and trajectory of a moving contact given a sequence of range-only measurements to the contact.

The uField Toolbox includes two tools for simulating these range-only sensors, the `uFldBeaconRangeSensor` and `uFldContactRangeSensor` applications. Each tool is situated on the shoreside computer and awaits the arrival of a "range request" from a deployed or simulated vehicle. The shoreside computer is also receiving node reports from all vehicles and is thus able to factor in relative vehicle position in determining whether or not to reply. Range replies contain the information otherwise perhaps derived from a range-only sensor residing on the vehicle.

Simulating Inter-Vehicle Message Transmission

Message transmission between vehicles is never a given. There is almost always some degree of range dependency, and some degree of uncertainty at any range as to whether the message will get through. The message length itself may also be limited depending on how it is sent. Underwater messaging is particularly subject to all three considerations.

The `uFldNodeComms` utility is designed to simulate message passing between vehicles. This utility resides on a shoreside computer and operates by requiring inter-vehicle message passing to be routed through this utility back out to the receiving vehicles. Since `uFldNodeComms` also is receiving node reports from all fielded vehicles, it may apply the relative position of vehicles in its determination of whether or not to send the message. It may also arbitrarily restrict the size of the message being sent.

1.3 Brief Background of MOOS-IvP

MOOS was written by Paul Newman in 2001 to support operations with autonomous marine vehicles in the MIT Ocean Engineering and the MIT Sea Grant programs. At the time Newman was a post-doc working with John Leonard and has since joined the faculty of the Mobile Robotics

Group at Oxford University. MOOS continues to be developed and maintained by Newman at Oxford and the most current version can be found at his web site. The MOOS software available in the MOOS-IvP project includes a snapshot of the MOOS code distributed from Oxford. The IvP Helm was developed in 2004 for autonomous control on unmanned marine surface craft, and later underwater platforms. It was written by Mike Benjamin as a post-doc working with John Leonard, and as a research scientist for the Naval Undersea Warfare Center in Newport Rhode Island. The IvP Helm is a single MOOS process that uses multi-objective optimization to implement behavior coordination.

Acronyms

MOOS stands for "Mission Oriented Operating Suite" and its original use was for the Bluefin Odyssey III vehicle owned by MIT. IvP stands for "Interval Programming" which is a mathematical programming model for multi-objective optimization. In the IvP model each objective function is a piecewise linear construct where each piece is an *interval* in N-Space. The IvP model and algorithms are included in the IvP Helm software as the method for representing and reconciling the output of helm behaviors. The term interval programming was inspired by the mathematical programming models of linear programming (LP) and integer programming (IP). The pseudo-acronym IvP was chosen simply in this spirit and to avoid acronym clashing.

1.4 Sponsors of MOOS-IvP

Original development of MOOS and IvP were more or less infrastructure by-products of other sponsored research in (mostly marine) robotics. Those sponsors were primarily The Office of Naval Research (ONR), as well as the National Oceanic and Atmospheric Administration (NOAA). MOOS and IvP are currently funded by Code 31 at ONR, Dr. Don Wagner and Dr. Behzad Kamgar-Parsi. Testing and development of course work at MIT is further supported by Battelle, Dr. Robert Carnes. MOOS is additionally supported in the U.K. by EPSRC. Early development of IvP benefited from the support of the In-house Laboratory Independent Research (ILIR) program at the Naval Undersea Warfare Center in Newport RI. The ILIR program is funded by ONR.

1.5 The Software

The MOOS-IvP autonomy software is available at the following URL:

```
http://www.moos-ivp.org
```

Follow the links to *Software*. Instructions are provided for downloading the software from an SVN server with anonymous read-only access.

1.5.1 Building and Running the Software

This document is written to Release 12.2. After checking out the tree from the SVN server as prescribed at this link, the top level directory should have the following structure:

```
moos-ivp/
   MOOS@                        README-WINDOWS.txt          build-moos.sh*
   MOOS-2375-Oct0611-Modified/  README.txt                  configure-ivp.sh*
```

```
bin/                    ivp/                    README-LINUX.txt
build/                  lib/                    README-OS-X.txt
build-ivp.sh*           scripts/
```

Note there is a `MOOS` directory and an `IvP` sub-directory. The `MOOS` directory is a symbolic link to a particular MOOS revision checked out from the Oxford server. In the example above this is Revision 2374 on the Oxford SVN server. This directory is left completely untouched other than giving it the local name `MOOS-2374-Apr0611`. The use of a symbolic link is done to simplify the process of bringing in a new snapshot from the Oxford server.

The build instructions are maintained in the README files and are probably more up to date than this document can hope to remain. In short building the software amounts to two steps - building MOOS and building IvP. Building MOOS is done by executing the build-moos.sh script:

```
> cd moos-ivp
> ./build-moos.sh
```

Alternatively one can go directly into the `MOOS` directory and configure options with `ccmake` and build with `cmake`. The script is included to facilitate configuration of options to suit local use. Likewise the IvP directory can be built by executing the `build-ivp.sh` script. The `MOOS` tree must be built before building IvP. Once both trees have been built, the user's shell executable path must be augmented to include the two directories containing the new executables:

```
moos-ivp/MOOS/MOOSBin
moos-ivp/bin
```

At this point the software should be ready to run and a good way to confirm this is to run the example simulated mission in the missions directory:

```
> cd moos-ivp/ivp/missions/alpha/
> pAntler alpha.moos
```

Running the above should bring up a GUI with a simulated vehicle rendered. Clicking the `DEPLOY` button should start the vehicle on its mission. If this is not the case, some help and email contact links can be found at `www.moos-ivp.org/support/`, or emailing `issues@moos-ivp.org`.

1.5.2 Operating Systems Supported by MOOS and IvP

The MOOS software distributed by Oxford is well supported on Linux, Windows and Mac OS X. The software distributed by MIT includes additional MOOS utility applications and the IvP Helm and related behaviors. These modules are support on Linux and Mac OS X and the software compiles and runs on Windows but Windows support is limited.

1.6 Where to Get Further Information

1.6.1 Websites and Email Lists

There are two web sites - the MOOS web site maintained by Oxford University, and the MOOS-IvP web site maintained by MIT. At the time of this writing they are at the following URLs:

```
http://www.robots.ox.ac.uk/~pnewman/TheMOOS/

http://www.moos-ivp.org
```

What is the difference in content between the two web sites? As discussed previously, MOOS-IvP, as a set of software, refers to the software maintained and distributed from Oxford *plus* additional MOOS applications including the IvP Helm and library of behaviors. The software bundle released at moos-ivp.org does include the MOOS software from Oxford - usually a particular released version. For the absolute latest in the core MOOS software and documentation on Oxford MOOS modules, the Oxford web site is your source. For the latest on the core IvP Helm, behaviors, and MOOS tools distributed by MIT, the moos-ivp.org web site is the source.

There are two mailing lists open to the public. The first list is for MOOS users, and the second is for MOOS-IvP users. If the topic is related to one of the MOOS modules distributed from the Oxford web site, the proper email list is the "moosusers" mailing list. You can join the "moosusers" mailing list at the following URL:

```
https://lists.csail.mit.edu/mailman/listinfo/moosusers,
```

For topics related to the IvP Helm or modules distributed on the moos-ivp.org web site that are not part of the Oxford MOOS distribution (see the software page on moos-ivp.org for help in drawing the distinction), the "moosivp" mailing list is appropriate. You can join the "moosivp" mailing list at the following URL:

```
https://lists.csail.mit.edu/mailman/listinfo/moosivp,
```

1.6.2 Documentation

Documentation on MOOS can be found on the Oxford University web site:

```
http://www.robots.ox.ac.uk/~pnewman/MOOSDocumentation/index.htm
```

This includes documentation on the MOOS architecture, programming new MOOS applications as well as documentation on several bread-and-butter applications such as `pAntler`, `pLogger`, `uMS`, `pMOOSBridge`, `iRemote`, `iMatlab`, `pScheduler` and more. Documentation on the IvP Helm, behaviors and autonomy related MOOS applications not from Oxford can be found on the www.moos-ivp.org web site under the Documentation link. Below is a summary of documents:

List of available MOOS-IvP related documentation

- *An Overview of MOOS-IvP and a Brief Users Guide to the IvP Helm Autonomy Software* - This is the primary document describing the IvP Helm regarding how it works, the motivation for its design, how it is used and configured, and example configurations and results from simulation.
- *MOOS-IvP Autonomy Tools Users Manual* - A users manual for several MOOS applications, and off-line tools for post-mission analysis collectively referred to as the Alog Toolbox. The MOOS applications include: `pNodeReporter`, `uTimerScript`, `uHelmScope`, `uPokeDB`,

`uSimMarine`, `pSearchGrid`, `pBasicContactMgr`, `uXMS`, `uTermCommand`, `pMarineViewer`, `pEchoVar`, and `uProcessWatch`. These applications are common supplementary tools for running an autonomy system in simulation and on the water.

- *Extending a MOOS-IvP Autonomy System and Users Guide to the IvPBuild Toolbox* - This document is a users manual for those wishing to write their own IvP Helm behaviors and MOOS modules. It describes the IvPBehavior and CMOOSApp superclass. It also describes the IvPBuild Toolbox containing a number of tools for building IvP Functions, the primary output of behaviors. It provides an example template directory with example IvP Helm behavior and an example MOOS application along with an example CMake build structure for linking against the standard software MOOS-IvP software bundle. MIT CSAIL Technical Report TR-2009-037.

2 The uFldNodeBroker Utility: Brokering Node Connections

The `uFldNodeBroker` application is a tool for brokering connections between a node (a simulated or real vehicle) and a shoreside community. It is used primarily in coordination with `uFldShoreBroker` to discover and share host IP and port information to automate the dynamic configurations of `pMOOSBridge`. Inter-vehicle communications over the network are handled by `pMOOSBridge` in both simulation with single or multiple machines as well as on fielded vehicles using Wi-Fi or cellphone connections. The `pMOOSBridge` application simply needs to know the IP address and port number of connected machines. Often these aren't known at run-time and even if they were, maintaining that information in configuration files may be unduly cumbersome, especially for large sets of vehicles. This tool meant to automate the configuration by letting the nodes and shoreside community discover each other by giving the node (`uFldNodeBroker`) some initial hints on where to find the shoreside community on the network. The typical layout is shown in Figure 2.

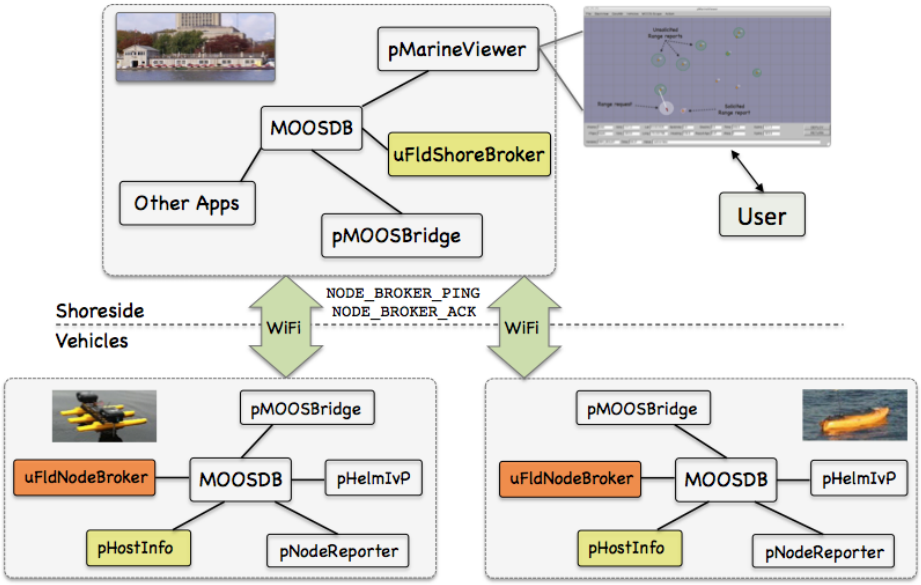


Figure 2: **Typical uFldNodeBroker Topology:** A vehicle (node) sends information about itself (IP address and port number) to possible shoreside locations. Once a connection is made, further bridging is established between the node and shoreside communities.

The functionality of `uFldNodeBroker` paraphrased:

- Discover the node's host information (typically from `pHostInfo`).
- For candidate shoreside hosts, request a new bridge in `pMOOSBridge` to each candidate for the variable `NODE_BROKER_PING`.
- Publish `NODE_BROKER_PING` with the node's host information.
- Await a reply in the form of incoming `NODE_BROKER_ACK` mail, presumably from the shoreside community running `uFldShoreBroker`.
- Now that a shoreside community is known, request new bridges from the node's local `pMOOSBridge` for all the info we otherwise want bridged to the shoreside.

- Keep sending pings periodically in case the shoreside community is re-started and needs to re-establish connections to nodes.

2.1 Overview of the uFldNodeBroker Interface and Configuration Options

The `uFldNodeBroker` application may be configured with a configuration block within a `.moos` file. Its interface is defined by its publications and subscriptions for MOOS variables consumed and generated by other MOOS applications. An overview of the set of configuration options and interface is provided in this section. If one has access to to a command line where the `uFldNodeBroker` has been built, interface information may also be seen by typing `"uFldNodeBroker --interface"`, and configuration information by typing `"uFldNodeBroker --example"`.

2.1.1 Configuration Parameters of uFldNodeBroker

The following parameters are defined for `uFldNodeBroker`.

`KEYWORD:`	An optional unique identifier to be read by the shoreside broker to conditionlly respond to pings.
`BRIDGE:`	A variable to register with `pMOOSBridge` for bridging to a shoreside MOOS community once a shoreside connection has been established.
`TRY_SHORE_HOST:`	A candidate IP address and port number to send initial pings in hopes of establishing a connection.

Examples:

```
KEYWORD = lemon
TRY_SHORE_HOST = host_ip=localhost, port_udp=9200, community=topside
BRIDGE = src=NODE_REPORT_LOCAL, alias=NODE_REPORT
```

2.1.2 MOOS Variables Published by uFldNodeBroker

The primary output of `uFldNodeBroker` to the MOOSDB are the requests to `pMOOSBridge` for registrations, and the outgoing pings to candidate shoreside communities.

`PMB_REGISTER:`	A message to pMOOSBridge to add a new bridge for a given variable and given target MOOS community at a specified IP address and port number.
`NODE_BROKER_PING:`	A message written locally but bridged to a candidate shoreside MOOS community, containing IP address and port information about the local community.

2.1.3 MOOS Variables Subscribed for by uFldNodeBroker

The `uFldNodeBroker` application subscribes to the following MOOS variables:

PHI_HOST_INFO:	Information about the local host IP address, the MOOS community name, the port on which the DB is running, and the port on which the local pMOOS-Bridge is listening for UDP messages.
NODE_BROKER_ACK:	Information published presumably by uFldShoreBroker running in a separate shoreside community. Message has information about the shoreside host including the community name, IP address and port numbers for the MOOSDB and its local pMOOSBridge process.

2.1.4 Command Line Usage of uFldNodeBroker

The uFldNodeBroker application is typically launched with pAntler, along with a group of other shoreside modules. However, it may be launched separately from the command line. The command line options may be shown by typing "uFldNodeBroker --help":

Listing 1 - Command line usage for the uFldNodeBroker tool.

```
 0  ============================================================
 1  Usage: uFldNodeBroker file.moos [OPTIONS]
 2  ============================================================
 3
 4  Options:
 5    --alias=<ProcessName>
 6        Launch uFldNodeBroker with the given
 7        process name rather than uFldNodeBroker.
 8    --example, -e
 9        Display example MOOS configuration block.
10    --help, -h
11        Display this help message.
12    --interface, -i
13        Display MOOS publications and subscriptions.
14    --version,-v
15        Display release version of uFldNodeBroker.
```

2.1.5 An Example MOOS Configuration Block

To see an example MOOS configuration block, enter the following from the command-line: "uFldNodeBroker -e". This will show the output shown in Listing 2 below.

Listing 2 - Example configuration of the uFldNodeBroker application.

```
 0  ============================================================
 1  uFldNodeBroker Example MOOS Configuration
 2  ============================================================
 3
 4  ProcessConfig = uFldNodeBroker
 5  {
 6    AppTick   = 4
 7    CommsTick = 4
 8
 9    KEYWORD        = lemon
10
```

```
11    TRY_SHORE_HOST = host_ip=localhost, port_udp=9200, community=topside
12    TRY_SHORE_HOST = host_ip=128.30.24.232, port_udp=9200
13
14    BRIDGE = src=VIEW_POLYGON
15    BRIDGE = src=VIEW_POINT
16    BRIDGE = src=VIEW_SEGLIST
17
18    BRIDGE = src=NODE_REPORT_LOCAL, alias=NODE_REPORT
19  }
```

2.2 Terminal Output - A Detailed Snapshot of the Broker Status

The `uFldNodeBroker` application produces some useful information to the terminal on every iteration of the application. An example is shown in Listing 3 below. The counter on the end of line 0 is incremented on each iteration of `uFldNodeBroker`. On line 1, the number of incoming mail messages for `PHI_HOST_INFO` is tallied where the host information bundle is deemed complete. It is complete if it contains the host community, IP address and UDP port number. If an incomplete host information packet is received, it is tallied in line 2. The number on line 2 should always be zero. If the number on line 2 is not zero, or the number on line 1 never increments, one should check the operation of the `pHostInfo`. The number on line 3 indicates the number of time the detected host information changes. This should always be zero. The number on line 4 indicates the number of dynamic bridge requests posted to `pMOOSBridge`. These are in the form of postings to the variable `PMB_REGISTER`, which occur first for outgoing pings to candidate shoreside hosts, and then for the user `BRIDGE` configuration variables after a shoreside host has been connected. The number on line 5 represents the number of pings generated. This number should always continue to grow since the node broker continues to emit pings even after a normal shoreside connection has been established. The number on line 6 indicates the number of valid `NODE_BROKER_ACK` messages received from the shoreside community. Invalid acks are tallied on line 7. An invalid ack may be due to a mismatch in time warp between node and shoreside, or a mismatch in keywords if keywords are being used.

Listing 3 - Example terminal output of the uFldNodeBroker tool.

```
 0  NodeBroker Status ----------- (32)
 1    Total OK  PHI_HOST_INFO    received: 4
 2    Total BAD PHI_HOST_INFO    received: 0
 3    Total HOST_INFO changes    received: 0
 4    Total PMB_REGISTER           posted: 5
 5    Total NODE_BROKER_PING       posted: 31
 6    Total OK  NODE_BROKER_ACK  received: 26
 7    Total BAD NODE_BROKER_ACK  received: 0
 8
 9  Configuration Info:
10    Total Valid TryHosts:   1
11    [1]: hostip=localhost,port_udp=9200
12    Total InValid TryHosts: 0
13
14  Node Information: james
15        HostIP:  10.0.0.8
16     Port (DB):  9204
17    Port (UDP):  9304
18     Time Warp:  1
19
```

```
20  Shoreside Information: shoreside
21         HostIP:  10.0.0.8
22      Port (DB):  9000
23     Port (UDP):  9200
24      Time Warp:  1
```

The information group in lines 9-12 indicates the validity of the `TRY_SHORE_HOST` configurations provided by the user. The number of valid configurations are shown on line 10, with the actual valid configurations following. The number of invalid configurations is tallied on line 12 with the invalid configuration lines following. An invalid configuration merely reflects a syntax error and doesn't imply anything about the status of the candidate host.

The information group in lines 14-18 reflect information about the node running `uFldNodeBroker`. It reflects the content received in incoming `PHI_HOST_INFO` messages (and for that matter, the info contained in outgoing `NODE_BROKER_PING` messages).

The information group in lines 20-24 is perhaps the most telling. The value in the fields after each colon will be empty until a shoreside connection has been established. If the nodes and the shoreside communities are all running in simulation on a single machine, the IP addresses between lines 15 and 21 will match as they do in this example.

3 The uFldShoreBroker Utility: Brokering Shore Connections

The `uFldShoreBroker` application is a tool for brokering connections between a shoreside community and one or more nodes (simulated or real vehicles). A shoreside community is collection of MOOS processes typically running a GUI providing a situational display and managing messages to and from fielded vehicles. This is depicted in the notional rendering in Figure 3 below. The shoreside community in practice is often situated on a ship with UUVs below, and is more aptly refered to as the topside community. The user interacts with the GUI or perhaps other communication modules, to send high-level messages to the vehicles.

The `uFldShoreBroker` application is used primarily in coordination with `uFldNodeBroker`, running on the vehicles, to discover and share host IP and port information to automate the dynamic configurations of `pMOOSBridge`. Inter-vehicle communications over the network are handled by `pMOOSBridge` in both simulation with single or multiple machines as well as on fielded vehicles using Wi-Fi or cellphone connections. The `pMOOSBridge` application simply needs to know the IP address and port number of connected machines. Often these aren't known at run-time and even if they were, maintaining that information in configuration files may be unduly cumbersome, especially for large sets of vehicles. This tool is meant to automate the configuration by letting the nodes and shoreside community discover each other by letting (`uFldShoreBroker`) respond to incoming pings, i.e., initialization messages, from nodes on the network. The typical layout is shown in Figure 2.

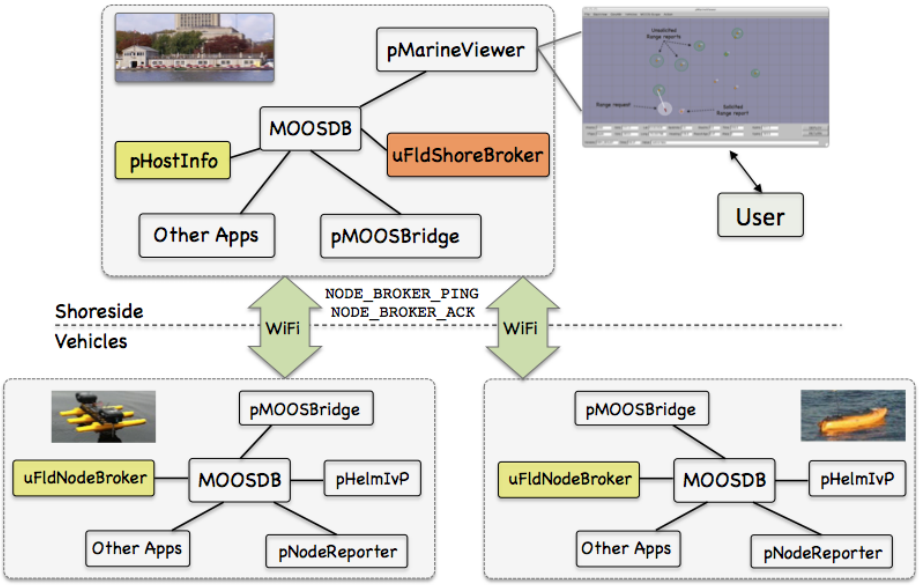


Figure 3: **Typical uFldShoreBroker Topology:** A vehicle (node) sends information about itself (IP address and port number) to the shoreside, received by uFldShoreBroker. It responds by (a) acknowledging the connection to the node, and (b) establishing user configured bridges of particular MOOS variables to the node.

The functionality of `uFldShoreBroker` paraphrased:

- Discover the shoreside's own host information (typically from `pHostInfo`).

- Await incoming NODE_BROKER_PING messages from non-local vehicles.
- Upon an incomqing ping, respond to the nodes with a NODE_BROKER_ACK message to the location specified in the ping message.
- Establish new bridges to the nodes for variables specified previously by the user in the uFldShoreBroker configuration.
- Keep sending acknowledgements periodically to confirm to vehicles that they are still connected to the shoreside community.

3.1 Overview of the uFldShoreBroker Interface and Configuration Options

The uFldShoreBroker application may be configured with a configuration block within a .moos file. Its interface is defined by its publications and subscriptions for MOOS variables consumed and generated by other MOOS applications. An overview of the set of configuration options and interface is provided in this section. If one has access to to a command line where the uFldShoreBroker has been built, interface information may also be seen by typing "uFldShoreBroker --interface", and configuration information by typing "uFldShoreBroker --example".

3.1.1 Configuration Parameters of uFldShoreBroker

The following parameters are defined for uFldShoreBroker.

BRIDGE:	Names a MOOS variable to be bridged to a node community.
QBRIDGE:	Shorthand notation for a common bridging pattern.

As an example, BRIDGE = src=DEPLOY_ALL, alias=DEPLOY, will result in the bridging of variable DEPLOY_ALL from the local MOOSDB, to the variable DEPLOY in a remote MOOS community. Further examples are given in Section 3.2.

3.1.2 MOOS Variables Published by uFldShoreBroker

The primary output of uFldShoreBroker to the MOOSDB are the requests to pMOOSBridge for registrations, and the outgoing acknowledgement replies to remote node/vehicle communities.

PMB_REGISTER:	A message to pMOOSBridge to add a new bridge for a given variable and given target MOOS community at a specified IP address and port number.
NODE_BROKER_ACK:	A message written locally but bridged to a remote vehicle MOOS community, containing IP address and port information about the local shoreside community.

3.1.3 MOOS Variables Subscribed for by uFldShoreBroker

The uFldShoreBroker application subscribes to the following MOOS variables:

PHI_HOST_INFO: Information about the local host IP address, the MOOS community name, the port on which the DB is running, and the port on which the local pMOOSBridge is listening for UDP messages.

NODE_BROKER_PING: Information published presumably by uFldNodeBroker running in a remote vehicle community. Message has information about the node host including the community name, IP address and port numbers for the MOOSDB and its local pMOOSBridge process.

3.1.4 Command Line Usage of uFldShoreBroker

The uFldShoreBroker application is typically launched with pAntler, along with a group of other shoreside modules. However, it may be launched separately from the command line. The command line options may be shown by typing "uFldShoreBroker --help":

Listing 4 - Command line usage for the uFldShoreBroker tool.

```
0  ============================================================
1  Usage: uFldShoreBroker file.moos [OPTIONS]
2  ============================================================
3
4  Options:
5    --alias=<ProcessName>
6        Launch uFldShoreBroker with the given
7        process name rather than uFldShoreBroker.
8    --example, -e
9        Display example MOOS configuration block.
10   --help, -h
11       Display this help message.
12   --interface, -i
13       Display MOOS publications and subscriptions.
14   --version,-v
15       Display release version of uFldShoreBroker.
```

3.1.5 An Example MOOS Configuration Block

As of MOOS-IvP Release 4.2, most if not all MOOS apps are implemented to support the -e or --example command-line switches. To see an example MOOS configuration block, enter the following from the command-line:

```
$ uFldShoreBroker -e
```

This will show the output shown in Listing 5 below.

Listing 5 - Example configuration of the uFldShoreBroker application.

```
0  ============================================================
1  uFldShoreBroker Example MOOS Configuration
2  ============================================================
3
4  ProcessConfig = uFldShoreBroker
```

```
 5  {
 6    AppTick   = 4
 7    CommsTick = 4
 8
 9    BRIDGE =  src=DEPLOY_ALL, alias=DEPLOY
10    BRIDGE =  src=DEPLOY_$V,  alias=DEPLOY
11
12    QBRIDGE = RETURN
13
14    BRIDGE  = src=UP_LOITER_$N, alias=UP_LOITER
15
16    // NOTE: The following line is shorthand for the next two.
17    // QBRIDGE = FOOBAR
18    // BRIDGE  = src=FOOBAR_ALL, alias=FOOBAR
19    // BRIDGE  = src=FOOBAR_$V,  alias=FOOBAR
20
21  }
```

3.2 Bridging Variables Upon Connection to Nodes

A primary function of `uFldShoreBroker` is to establish bridging relationships to a remote node community after it has received a ping from that community. These variables are specified in the configuration block with lines like `BRIDGE = "src=DEPLOY_ALL, alias=DEPLOY"` as in Listing 5. This step is described next.

3.2.1 Bridging with pMOOSBridge

Static bridging with `pMOOSBridge` is done by specifying the desired bridge in the the `pMOOSBridge` configuration block with a line of the form:

```
UDPSHARE= [VAR]-> community_name @ HostIP : Port [ALIAS]
```

For example:

```
UDPSHARE= [DEPLOY_ALL]-> henry @ 12.56.111.1 : 9200 [DEPLOY]
```

The problem with the static bridging is that the details, such as the community name, host IP address and port number of the vehicles that ultimately will connect to the shoreside community are not known until run time. In this case, dynamic bridge registration needs to be used by sending `pMOOSBridge` a message after it has been launched. For example, the above bridging relationship could be established by sending the following message:

```
PMB_REGISTER = "src_var=DEPLOY_ALL, dest_community=henry, dest_host=2.56.111.1
               dest_port=9200, dest_alias=DEPLOY"
```

It is the job of `uFldShoreBroker` to post the above style dynamic requests once the node information becomes known to the shoreside community.

3.2.2 Handling a Valid Incoming Ping from a Remote Node

The basic job of `uFldShoreBroker` is to await incoming pings, and use the information in a ping message to (a) decide if the ping should be accepted, and (b) send the appropriate response back to the sender, and (c) set up new outgoing bridge relationships if the ping is indeed accepted. The contents of a ping may look something like:

```
NODE_BROKER_PING = "community=henry,host=192.168.1.22,port=9000,port_udp=9200,
                    keyword=lemon,time_warp=10,time=1325178800.81"
```

The last `keyword` and `time_warp` fields are used for determining whether the ping should be accepted. There must be a match in the MOOS *time warp* used by the shoreside MOOS community and any node connected to the shore. This is always 1 when operating vehicles in the field, but may be set to a much larger number in simulation. The time warp is set with the parameter `MOOSTimeWarp` at the top of the .moos configuration file. The keyword field is optionally used, but if `uFldShoreBroker` is using a keyword, then nodes must include a matching keyword in their pings in order to be accepted.

The ping consists of three key pieces of information:

- The community name of the node,
- The IP address of the node,
- The port number on which the node is listening for UDP messages with its own local `pMOOSBridge` running.

Once this information is known by the shoreside broker, new bridges can be established for variables identified by the user. The only other information needed is (a) the name of the variable in the local MOOSDB, and (b) the name (alias) of the variable as it is to be known in the remote MOOSDB. Once a valid ping has been received and accepted, `uFldShoreBroker` is ready to establish bridge arrangements with its local MOOS bridge.

Vanilla Bridge Arrangements

The simplest bridge arrangment specifies (a) the variable as it is known locally, and (b) the variable name as it is to be known remotely. This is done with a `uFldShoreBroker` configuration line similar to:

```
BRIDGE = src=DEPLOY_ALL, alias=DEPLOY
```

For *each* unique incoming ping, a new bridge arrangement will be requested. By unique, we mean having a distinct community (remote vehicle) name. For example, if pings are received and accepted from *henry*, *james*, and *ike*, `uFldShoreBroker` would make three separate posts, perhaps looking like:

```
PMB_REGISTER = "src_var=DEPLOY_ALL, dest_community=henry, dest_host=2.56.111.1
                dest_port=9200, dest_alias=DEPLOY"
PMB_REGISTER = "src_var=DEPLOY_ALL, dest_community=james, dest_host=2.56.111.3
                dest_port=9200, dest_alias=DEPLOY"
PMB_REGISTER = "src_var=DEPLOY_ALL, dest_community=ike, dest_host=2.56.111.4
                dest_port=9200, dest_alias=DEPLOY"
```

At this point the behavior of pMOOSBridge would be functionally equivalent to the scenario where the following three lines were in the pMOOSBridge configuration block:

```
UDPSHARE= [DEPLOY_ALL]-> henry @ 12.56.111.1 : 9200 [DEPLOY]
UDPSHARE= [DEPLOY_ALL]-> james @ 12.56.111.3 : 9200 [DEPLOY]
UDPSHARE= [DEPLOY_ALL]-> ike   @ 12.56.111.4 : 9200 [DEPLOY]
```

Bridge Arrangements with Macros

The user may configure uFldShoreBroker with bridge arrangments containing a couple types of macros. For example, consider the configuration:

```
BRIDGE = src=DEPLOY_$V, alias=DEPLOY
```

The $V macro will expand to the name of the vehicle when it comes time to request a new bridge. If the newly received ping is from the node named *gilda*, the bridge request from the above pattern may look like:

```
PMB_REGISTER = "src_var=DEPLOY_GILDA, dest_community=henry, dest_host=2.56.111.1
               dest_port=9200, dest_alias=DEPLOY"
```

Note the vehicle name in the MOOS variable macro was expanded to be upper case, even though the ping information refered to the vehicle as *gilda*. This is just to aid in the convention that MOOS variable are typically all upper case. If one really want a literal expansion with no case altering, the macro $v, lower-case v, may be used instead. The macro is only respected as part of src field. In other words, if the bridge were configured with alias=DEPLOY_$V, the macro would not be expanded.

The other type of macro implemented is the $N macro, as in:

```
BRIDGE = src=LOITER_$N, alias=LOITER
```

The $N macro will expand to the integer value representing number of unique pings received thus far. For example, if three pings are received and accepted from *henry*, *james*, and *ike*, uFldShoreBroker would make three separate posts, perhaps looking like:

```
PMB_REGISTER = "src_var=LOITER_1, dest_community=henry, dest_host=2.56.111.1
               dest_port=9200, dest_alias=LOITER"
PMB_REGISTER = "src_var=LOITER_2, dest_community=james, dest_host=2.56.111.3
               dest_port=9200, dest_alias=LOITER"
PMB_REGISTER = "src_var=LOITER_3, dest_community=ike, dest_host=2.56.111.4
               dest_port=9200, dest_alias=LOITER"
```

This may be useful when used in conjunction with another MOOS process generating output generically for N vehicles, without having to know the vehicle names in advance.

Shortcut to a Common Bridge Arrangement - the QBRIDGE Parameter

A common usage pattern is to configure uFldShoreBroker to request two types of bridges for a given variable, for example:

```
BRIDGE = src=DEPLOY_ALL, alias=DEPLOY
BRIDGE = src=DEPLOY_$V,  alias=DEPLOY
BRIDGE = src=RETURN_ALL, alias=RETURN
BRIDGE = src=RETURN_$V,  alias=RETURN
```

This could be use in the shoreside community for easily commanding vehicles. When the user wishes to deploy all vehicles, a posting of `DEPLOY_ALL="true"` does the trick. If the user wishes only the vehicle *james* to return, a posting of `RETURN_JAMES="true"` may be be made. This pattern is so common that this shortcut is supported. This is done with the `QBRIDGE`, "quick bridge", parameter. The above four configuration lines could ba accomplished instead by:

```
QBRIDGE = DEPLOY, RETURN
```

3.3 Usage Scenarios the uFldShoreBroker Utility

The `uFldShoreBroker` was designed with a canonical command-and control scenario in mind. The idea is that the N deployed vehicles have a common autonomy protocol implemented. For example, a message to deploy or return a vehicle is the same message for each deployed vehicle. The idea is that two types of communication channels need to be established with `pMOOSBridge`, (a) messages sent to all vehicles, and (b) messages sent to a particular named vehicle. The convention proposed here is to do this with the two types of bridging described in the discussion of the `QBRIDGE` parameter, in Section 3.2.2. For a variable such as `DEPLOY`, a posting in the shoreside community to `DEPLOY_ALL` would go to all known vehicles, and a posting to `DEPLOY_HENRY` would only go to that particular vehicle.

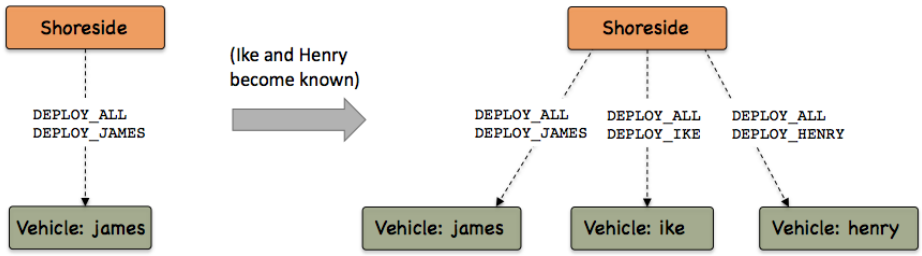


Figure 4: **Common uFldShoreBroker Usage Scenario:** As vehicles become known to the shoreside, each vehicle has two new bridges established. The first is the same for all vehicles to allow broadcasting, and the second bridge is unique to the particular vehicle, for individual command and control.

3.4 Terminal Output

The `uFldShoreBroker` application produces some useful information to the terminal on every iteration of the application. An example is shown in Listing 6 below. On line 1, the prevailing MOOS time warp in the shoreside community is shown. In Lines 2-5, the primary MOOS variables subscribed for and received are listed with their tallies.

Listing 6 - Example terminal output of the uFldShoreBroker tool.

```
0  ShoreBroker Status ----------- (298)
1   MOOS Time Warp:                 10
```

```
 2   Total PHI_HOST_INFO    received: 31
 3   Total NODE_BROKER_PING received: 415
 4   Total NODE_BROKER_ACK    posted: 390
 5   Total PMB_REGISTER       posted: 24
 6
 7  Node              IPAddress        Status            Elapsed  Skew
 8  --------------    ---------------  -------------     -------  ----
 9  henry             18.12.14.26      ok                1.0      0.6673
10  james             18.31.24.11      ok                1.0      0.6129
11  ike               18.31.215.13     ok                1.0      0.5149
12  gilda             18.3.224.246     timewarp (10!=1)  6.9      11926604570.9928
```

In lines 9-12, the status of each of the known vehicles is shown. The first three vehicles had their pings accepted. Their IP addresses are shown in the second column. Their status is shown in the third column (the status for *gilda* shows that it is running locally with a time warp of 1 which is why it was rejected). The elapsed time in the fourth column is the time since the last ping was received by the shoreside. The fifth column shows the time skew between the timestamp in the `NODE_BROKER_PING` message compared to the time it was received. Some of this is due to (a) latency in transmission, (b) latency due to App Ticks in brokers on both sides, and (c) clock disrepancy between the shoreside and the node computers. Note the skew for *gilda* is huge since the shoreside is in effect multiplying its clock time by 10, while *gilda* is not. It's also worth mentioning that the skew will be magnified for higher time warps. Currently incoming ping connection requests are not denied due to a high clock skew, but this may be implemented in the future.

4 The pHostInfo Utility: Detecting and Sharing Host Info

The `pHostInfo` application is a tool with a simple objective - determine the IP address of the machine on which it is running and post it to the MOOSDB. Although this information is available in a number of ways to a user at the keyboard, it may not be readily available for reasoning about within a MOOS community. Often, from an application's perspective, the host name is simply known and configured as *localhost*. This is fine for most purposes, but in situations where a user is on a machine where the IP address changes frequently, and the user is launching MOOS processes that talk to other machines, it may be very convenient to auto-determine the prevailing IP address and publish it to the MOOSDB. The typical usage scenario for `pHostInfo` is shown in Figure 5.

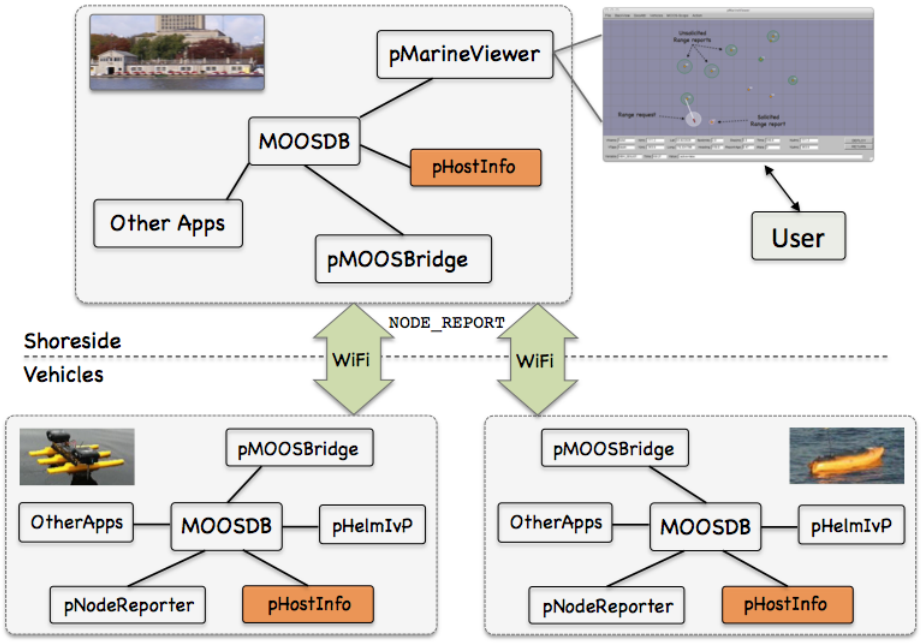


Figure 5: **Typical pHostInfo Topology:** A shoreside or topside community is receiving information from several deployed vehicles, in the form of node reports. The node reports contain time-stamped updated vehicle positions, from which the speed and distance measurements are derived and posted to the shoreside MOOSDB.

There are two scenarios where this is currently envisioned to be useful. The first is when the fielded vehicles are on the network with their IP addresses set via DHCP. For example if on the network via a cellular phone connection. The second is if the "vehicle" is a simulated vehicle running on a user's laptop also with its IP address set via DHCP. In both cases there may be another MOOS community with a known IP address, e.g., a shoreside community, to which the local vehicle wishes to inform it of its current IP address. This simple process however does not get involved in any activity regarding the communication to other MOOS communities, but simply tries to determine and post, the IP address, e.g., `PHI_HOST_IP="192.168.0.1"` for other applications to do as they see fit.

4.1 Overview of the pHostInfo Interface and Configuration Options

The pHostInfo application may be configured with a configuration block within a .moos file. Its interface is defined by its publications and subscriptions for MOOS variables consumed and generated by other MOOS applications. An overview of the set of configuration options and interface is provided in this section. If one has access to a command line where the pHostInfo has been built, interface information may also be seen by typing "pHostInfo --interface", and configuration information by typing "pHostInfo --example".

4.1.1 Configuration Parameters of pHostInfo

The following parameters are defined for pHostInfo.

TEMP_FILE_DIR: Directory where temporary files are written. Default is "~/".

4.1.2 MOOS Variables Published by pHostInfo

The primary output of pHostInfo to the MOOSDB are the following four variables. Once these variables are published, pHostInfo does not publish them again unless requested by receiving mail HOST_INFO_REQUEST. Thus pHostInfo is mostly idle once the below four variables are posted.

PHI_HOST_IP:	The single best guess of the Host's IP address.
PHI_HOST_IP_ALL:	A comma-separated list of IP addresses if multiple addresses detected.
PHI_HOST_IP_VERBOSE:	A comma-separated list of IP addresses, with source information, if multiple addresses detected.
PHI_HOST_PORT:	The port number of the MOOSDB for this community.
PHI_HOST_PORT_UDP:	The port number on which pMOOSBridge may be listening for UDP messages.

4.1.3 MOOS Variables Subscribed for by pHostInfo

The pHostInfo application subscribes to the following MOOS variable:

HOST_INFO_REQUEST:	A request to re-determine and re-post the platform's host information.
PMB_UDP_LISTEN:	The port number which pMOOSBridge may be configured to listen for UDP messages.

4.1.4 Command Line Usage of pHostInfo

The pHostInfo application is typically launched with pAntler, along with a group of other shoreside modules. However, it may be launched separately from the command line. The command line options may be shown by typing "pHostInfo --help":

Listing 7 - Command line usage for the pHostInfo tool.

```
0  Usage: pHostInfo file.moos [OPTIONS]
1
2  Options:
```

```
 3    --alias=<ProcessName>
 4        Launch pHostInfo with the given process
 5        name rather than pHostInfo.
 6    --example, -e
 7        Display example MOOS configuration block
 8    --help, -h
 9        Display this help message.
10    --version,-v
11        Display the release version of pHostInfo.
12
13  Note: If argv[2] is not of one of the above formats
14        this will be interpreted as a run alias. This
15        is to support pAntler launching conventions.
```

4.1.5 An Example MOOS Configuration Block

As of MOOS-IvP Release 4.2, most if not all MOOS apps are implemented to support the `-e` or `--example` command-line switches. To see an example MOOS configuration block, enter the following from the command-line:

```
$ pHostInfo -e
```

This will show the output shown in Listing 8 below.

Listing 8 - Example configuration of the pHostInfo application.

```
 0  ============================================================
 1  pHostInfo Example MOOS Configuration
 2  ============================================================
 3  Blue lines: Default configuration
 4
 5  ProcessConfig = pHostInfo
 6  {
 7    AppTick   = 4
 8    CommsTick = 4
 9
10    TEMP_FILE_DIR = ./
11  }
```

4.2 Usage Scenarios the pHostInfo Utility

4.3 Handling Multiple IP Addresses

It is possible that a machine has more than one valid IP address at any given time, e.g., if its ethernet cable is plugged in, and it has a wireless connection. In this case, `pHostInfo` will make a guess that the ethernet connection takes precedent, and it will report this in the variable `PHI_HOST_IP`. The full set of IP addresses can be found in the other postings. For example it may not be uncommon to see something like the following three postings at one time:

```
PHI_HOST_IP:         118.10.24.23
PHI_HOST_IP_ALL:     118.10.24.23,169.224.126.40
PHI_HOST_IP_VERBOSE: OSX_ETHERNET2=118.10.24.23,OSX_AIRPORT=169.224.126.40
```

4.4 A Peek Under the Hood

The `pHostInfo` application currently only works for GNU/Linux and Apple OS X. It determines the IP information by making a system call within C++. A system call when generated will act as if the argument were typed on the command line. In this case the system call is generated and the output is redirected to a file. In a second step, `pHostInfo` then tries to read the IP address information from those files.

In GNU/Linux, the system call is based on the `ifconfig` command. In OS X, the system call is based on the `networksetup` command. Rather than determining in `pHostInfo` whether the user is running in a GNU/Linux or OS X environment, the system calls for both are invoked. Presumably a system call on a command not found in the user's shell path will not generate something that is confusable with a valid IP address.

4.4.1 Temporary Files

The temporary files are written to the user's home directory by default. This may be changed with the `TEMP_FILE_DIR` cofiguration parameter, for example, `TEMP_FILE_DIR=/tmp`. The set of temporary files are put into a folder named `.phostinfo/`. The set of temporary files may look like:

```
  $ cd ~/.phostinfo
  $ ls -a .ipinfo*
.ipinfo_linux_ethernet.txt    .ipinfo_osx_airport.txt     .ipinfo_osx_ethernet2.txt
.ipinfo_osx_wifi.txt          .ipinfo_linux_wifi.txt      .ipinfo_osx_ethernet1.txt
.ipinfo_osx_ethernet.txt
```

Some of these files may be empty, or some may contain error output if one of the system commands was not found, or was given an improper argument. The `pHostInfo` app will try to parse all of them to find a valid IP address. If more than one IP address is found, then this handled in the manner described previously in Section 4.3.

4.4.2 Possible Gotchas

The system calls invoked by `pHostInfo` need to be in the users shell path. A typical user default environment would have these in their shell path anyway, but it may be worth checking if things aren't working properly. Below is a list of commands that are run under the hood, and their probable locations on your system.

```
For Linux:
  /sbin/ifconfig
  /bin/grep
  /usr/bin/cut
  /usr/bin/awk
  /usr/bin/print
For OS X:
  /usr/sbin/networksetup
```

5 The uFldNodeComms Utility: Simulating Intervehicle Comms

The `uFldNodeComms` application is a tool for handling node reports and messages between vehicles. Rather than directly sending node reports and messages between vehicles, `uFldNodeComms` acts as an intermediary to conditionally pass a report or message on to another vehicle, where conditions may be the inter-vehicle range or other criteria. The assumption is that `uFldNodeComms` is running on a topside or shoreside computer, and receiving information about the present physical location of deployed vehicles through node reports. The typical layout is shown in Figure 6.

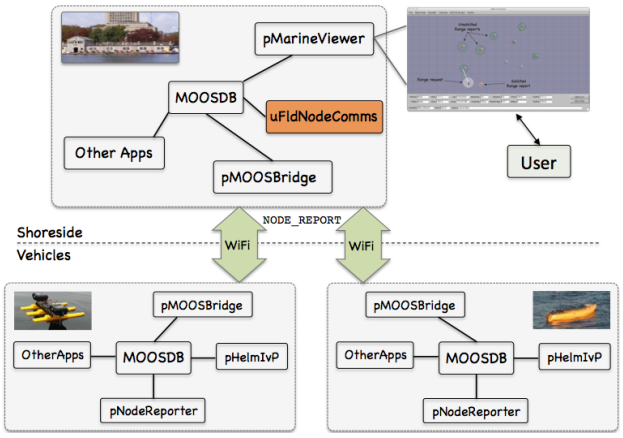


Figure 6: **Typical uFldNodeComms Topology:** A shoreside or topside community is receiving information from several deployed vehicles, in the form of node reports. The node reports contain time-stamped updated vehicle positions, from which the speed and distance measurements are derived and posted to the shoreside MOOSDB.

In short, `uFldNodeComms` subscribes for incoming node reports for any number of vehicles, and keeps the latest node report for each vehicle. On each iteration, for a each vehicle, if the node report has been updated, the report is published to a specially created MOOS variable for the other $n-1$ vehicles. A user-configured criteria is applied before publishing the new information. Typically this criteria involves the range between vehicles, but the criteria may be further involved. The idea between three vehicles *alpha*, *bravo*, and *charlie* is shown below in Figure 7.

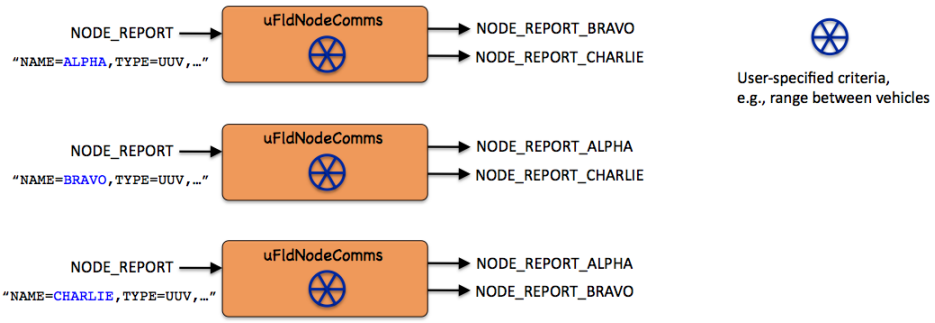


Figure 7: **Brokering by uFldNodeComms:** Each incoming node report is sent out to a specially named variable corresponding to one each of the other $n-1$ vehicles.

5.1 Overview of the uFldNodeComms Interface and Configuration Options

The `uFldNodeComms` application may be configured with a configuration block within a `.moos` file. Its interface is defined by its publications and subscriptions for MOOS variables consumed and generated by other MOOS applications. An overview of the set of configuration options and interface is provided in this section. If one has access to a command line where the `uFldNodeComms` has been built, interface information may also be seen by typing `"uFldNodeComms --interface"`, and configuration information by typing `"uFldNodeComms --example"`.

5.1.1 Configuration Parameters of uFldNodeComms

The following parameters are defined for `uFldNodeComms`.

`COMMS_RANGE:`	Max range outside which inter-vehicle node reports and node messages will not be sent. See page 5.2.1.
`CRITICAL_RANGE:`	Range within which inter-vehicle node reports will be shared even if group membership would otherwise disallow. See page 5.2.1.
`DEBUG:`	If true, further debugging informatin is produced to the terminal output.
`EARANGE:`	A parameter in the range of $[1, 10]$ for extending the range a vehicle may otherwise hear node reports from other vehicles. See page 5.2.1.
`GROUPS:`	If true, inter-vehicle node reports are shared only if two vehicles are in the same group. May be overridden if the two vehicles are within the critical range. See page 5.2.1.
`MIN_MSG_INTERVAL:`	The number of seconds required between messaged sends for any one source vehicle. See page 5.3.1.
`MAX_MSG_LENGTH:`	The total number of characters that may be sent in a string component of a node message. See page 5.3.1.
`STEALTH:`	A parameter in the range $[0, 1]$ for reducing the range other vehicles may otherwise hear the node reports from a source vehicle. See page 5.2.1.
`STALE_TIME:`	Time in seconds after which a vehicle will not receive node reports or messages unless a node report has been received by that vehicle. See page 5.2.1.
`VERBOSE:`	If true, status reports are displayed to the terminal during operation. See page 5.5.

5.1.2 MOOS Variables Published by uFldNodeComms

The primary output of `uFldNodeComms` to the MOOSDB are the node reports and node messages out to the recipient vehicles, and visual artifacts to be used for rendering the inter-vehicle communications.

NODE_MESSAGE_<VNAME>:	Node messages destined to be sent to a destination vehicle <VNAME> indicated as the recipient in the node message.
NODE_REPORT_<VNAME>:	Node reports destined to be sent to a given vehicle <VNAME> about the vehicle named in the node report.
VIEW_COMMS_PULSE:	A visual artifict for rendering the sending of a node report or node message between vehicles.

5.1.3 MOOS Variables Subscribed for by uFldNodeComms

The uFldNodeComms application subscribes to the following MOOS variables:

NODE_MESSAGE:	A node message from one vehicle to another.
NODE_REPORT:	A node report for a given vehicle from pNodeReporter.
NODE_REPORT_LOCAL:	Another name for a node report for a given vehicle from pNodeReporter.
UNC_STEALTH:	The extra stealth allowed a given vehicle to "hide" node reports to others.
UNC_EARANGE:	The extra range allowed a given vehicle to "hear" node reports of others.

5.1.4 Command Line Usage of uFldNodeComms

The uFldNodeComms application is typically launched with pAntler, along with a group of other shoreside modules. However, it may be launched separately from the command line. The command line options may be shown by typing "uFldNodeComms --help":

Listing 9 - Command line usage for the uFldNodeComms tool.

```
0    ============================================================
1    Usage: uFldNodeComms file.moos [OPTIONS]
2    ============================================================
3
4    Options:
5      --alias=<ProcessName>
6          Launch uFldNodeComms with the given process
7          name rather than uFldNodeComms.
8      --example, -e
9          Display example MOOS configuration block.
10     --help, -h
11         Display this help message.
12     --interface, -i
13         Display MOOS publications and subscriptions.
14     --version,-v
15         Display the release version of uFldNodeComms.
16
17   Note: If argv[2] does not otherwise match a known option,
18         then it will be interpreted as a run alias. This is
19         to support pAntler launching conventions.
```

5.1.5 An Example MOOS Configuration Block

As of MOOS-IvP Release 4.2, most if not all MOOS apps are implemented to support the -e or --example command-line switches. To see an example MOOS configuration block, enter the

following from the command-line:

```
$ uFldNodeComms -e
```

This will show the output shown in Listing 10 below.

Listing 10 - Example configuration of the uFldNodeComms application.

```
 0  ============================================================
 1  uFldNodeComms Example MOOS Configuration
 2  ============================================================
 3
 4  ProcessConfig = uFldNodeComms
 5  {
 6    AppTick   = 4
 7    CommsTick = 4
 8
 9    COMMS_RANGE    = 100       // default (in meters)
10    CRITICAL_RANGE = 30        // default (in meters)
11    STALE_TIME     = 5         // default (in seconds
12
13    MIN_MSG_INTERVAL = 30      // default (in seconds)
14    MAX_MSG_LENGTH   = 1000    // default (# of characters)
15
16    VERBOSE  = true            // default
17
18    STEALTH  = vname=alpha, stealth=0.8
19    EARANGE  = vname=alpha, earange=4.5
20
21    GROUPS   = true
22
23    PULSE_DURATION = 10;       // default (in seconds)
22  }
```

5.2 Handling Node Reports

Node reports may contain a bundle of useful information for sharing between vehicles. Perhaps the most important is the present vehicle position and trajectory. This may be used by the receiving vehicle for collision avoidance and formation keeping and so on. The vehicle position is also used by `uFldNodeComms` to determine if the node reports themselves are to be shared between vehicles. The inter-vehicle ranges derived from the `NODE_REPORT` messages are also used to determine if other more generic information contained in the `NODE_MESSAGE` variable is to be shared between vehicles.

5.2.1 The Criteria for Routing Node Reports

The basic criteria for sharing node reports is range. By default, the node report received for any one vehicle is passed on to *all* other known vehicles within comms range of the originating vehicle. The comms range is set by the parameter `COMMS_RANGE` as on line 9 in Listing 10 above. Starting with this as a baseline, a few other factors may be involved in determining whether a node report is shared.

Using Vehicle Group Information

If `uFldNodeComms` is configured with the parameter `GROUPS` set to true, as on line 21 in Listing 10, then node reports are only shared between vehicles having the same group name. The group name is a field contained in the node report itself, so the onus is on the vehicle to include this information as part of its report. The `pNodeReporter` application contains an optional configuration parameter `GROUP=GROUP_NAME` where the group information is declared for inclusion in all node reports. The motivation for the grouping option is to support multi-vehicle competitions where some vehicles want to convey positions to teammates, but not adversarial vehicles.

The `GROUPS` feature only affects the passing of node reports between vehicles. It does not affect the passing of node *messages* discussed further below. Node messages contain their addressee information explicitly.

Establishing and Inter-Vehicle Critical Range

When the two vehicles are within a range deemed critical, as set by the `CRITICAL_RANGE` configuration parameter as on line 10 in Listing 10, node reports are shared between vehicles regardless of the `COMMS_RANGE` parameter and the `GROUPS` parameter. The default for this parameter is 30 meters. The thought behind this feature is that, while it may be advantageous to not broadcast your own vehicle position to non group members for the purposes of a competition, it may be a good idea to share this information for the sake of collision avoidance.

Checking for Staleness

Since up-to-date inter-vehicle range information is used as part of the criteria in determining whether a vehicle receives a new node report from another, the position of the candidate recipient vehicle needs to reasonably up-to-date. The `STALE_TIME` configuration parameter may be set as on line 11 in Listing 10, to determine the amount of elapsed time without receiving a node report from a vehicle before it is considered stale. If a recipient vehicle becomes stale, it will also not receive node messages.

Bestowing a Vehicle with an Enhanced Stealth Property

By default, node reports are shared at equal distances between sender and receiver. In other words, when one vehicle comes into close enough range to receive node reports of another, the other the vehicle will also begin receiving node reports of the first vehicle. This puts each vehicle on equal footing if they are regarded in an adversarial context.

By using the `STEALTH` parameter, one vehicle may be given a bit of an advantage. The stealth parameter is assigned to a particular vehicle and is a number in the range $[0.5, 1]$. A message from source to destination is sent if:

$$actual\_range(src, dest) < comms\_range * stealth(src) \tag{1}$$

By default, the stealth factor for each vehicle is 1. A stealthy vehicle with a factor of one half, means the receiving vehicle needs to be twice as close as it would otherwise to receive the stealthy vehicle's node report or node message. The `CRITICAL_RANGE` parameter may be used to override

a vehicle's stealthiness with respect to sending node reports. That is, a message from source to destination is sent if:

$$actual\_range(src, dest) < max((comms\_range * stealth(src)), critical\_range) \quad (2)$$

The `STEALTH` value for a particular vehicle may be set in the MOOS configuration file as shown on line 18 in Listing 10. It may also be set dynamically by receiving mail on the variable `UNC_STEALTH`. For example the posting

```
UNC_STEALTH = vname=alpha, stealth=0.75
```

would immediately reset the stealth factor for vehicle *alpha* on the next iteration. The motivation for exposing this parameter via incoming MOOS mail is that another shoreside application, monitoring vehicle speed for example, may reward a vehicle operating in the field with increased stealth based on any criteria. This can be useful in constructing vehicle competitions.

Bestowing a Vehicle with an Enhanced Listening Property

In addition to the stealth property, a complementary property may be bestowed upon a vehicle using the `EARANGE` parameter. The earange parameter is assigned to a particular vehicle and is a number in the range $[1, 2]$. A message from source to destination is sent if:

$$actual\_range(src, dest) < comms\_range * earange(dest) \quad (3)$$

By default, the earange factor for each vehicle is 1. A vehicle with a factor of two may receive node reports of another vehicle at twice the range it may otherwise. In the end the stealth and earange properties may cancel each other out when their extreme values are factored together. Taken together a message from source to destination is sent if:

$$actual\_range(src, dest) < comms\_range * stealth(src) * earange(dest) \quad (4)$$

The `EARANGE` value for a particular vehicle may be set in the MOOS configuration file as shown on line 19 in Listing 10. It may also be set dynamically by receiving mail on the variable `UNC_EARANGE`. For example the posting

```
UNC_EARANGE = vname=alpha, earange=1.5
```

would immediately reset the earange factor for vehicle *alpha* on the next iteration. The motivation for exposing this parameter via incoming MOOS mail is that another shoreside application, monitoring vehicle speed for example, may reward a vehicle operating in the field with increased earange based on any criteria. This can be useful in constructing vehicle competitions.

5.2.2 Node Report Transmissions and pMOOSBridge

Node reports are communicated to recipient vehicles in coordination with the `pMOOSBridge` application. For each unique vehicle name discovered by `uFldNodeComms` via received node reports, it will publish a new MOOS variable `NODE_REPORT_NAME`. This variable will be published with the contents of other vehicles' node reports.

As shown in Figure 7, if three vehicles, *alpha*, *bravo*, and *charlie* become known, three corresponding MOOS variables NODE_REPORT_ALPHA, NODE_REPORT_BRAVO, and NODE_REPORT_CHARLIE will be published. The variable NODE_REPORT_ALPHA will be published with reports from *bravo* and *charlie* and so on. To make this happen, uFldNodeComms needs the corresponding bridge relationships from, for example, NODE_REPORT_ALPHA in the shoreside community to the variable NODE_REPORT in the alpha community on the alpha vehicle. This bridging relationship is established separately by the uFldShoreBroker application running in the shoreside community.

5.3 Handling Node Messages

Node messages are of a generic structure for sharing a MOOS variable-value pair between a source node and a destination node. A node message from vehicle *alpha* to vehicle *bravo* would be posted locally by *alpha* with something like:

```
NODE_MESSAGE = "src_node=alpha,dest_node=bravo,var_name=FOOBAR,string_val=hello"
```

This posting is bridged to the shoreside community running uFldNodeComms where it is considered for re-routing to the destination vehicle.

5.3.1 The Criteria for Routing Node Messages

The basic criteria for sharing node messages is (a) the message addressee, and (b) the range between the source and destination vehicles. Since uFldNodeComms is receiving and keeping track of incoming node reports from all vehicles, it has ready access to the inter-vehicle range between the source and destination nodes. The same criteria used for sending node *reports* is used for sending node messages. It must meet the range criteria as perhaps modified by the stealth and earange factors discussed earlier. The only difference is that the CRITICAL_RANGE parameter is irrelevant for the issue of sending node messages. This parameter was only used for node reports in the interest of safety and collision avoidance. There are however two other factors that may affect node message transmission, discussed next, message frequency and message size.

Enforcing a Minimum Time Between Node Messages

A maximum send frequency is enforced by requiring a minimum wait time between successful sends from a given source node. This minimum time is given by the parameter MIN_MSG_INTERVAL as on line 13 in Listing 10. The default is 30 seconds. This interval is defined by the time starting with a successful transmission of a node message from a source to any destination. A separate log is kept by uFldNodeComms for each known vehicle.

Enforcing a Maximum Node Message Length

The length of node messages may be limited with the parameter MAX_MSG_LENGTH, as on line 14 in Listing 10. The default maximum length is 1000 characters. The length of a message refers to the number of characters in the string_val field. For example, the length of the message

```
NODE_MESSAGE = "src_node=alpha,dest_node=bravo,var_name=FOOBAR,string_val=hello"
```

is five. Limiting the message length is a proxy for intervehicle communications where message packet length is constrained, as with acoustic communications for example.

Posting Messages to a Vehicle Group

A node message is typically addressed to another named vehicle. The sender may also address the message by group name. All vehicles in the group meeting the prevailing range criteria will receive the message. The group associated with the vehicle is declared in the node report sent by that vehicle. A node message using a group address is similar except for the use of the `dest_group` parameter

```
NODE_MESSAGE = "src_node=alpha,dest_group=red_team,var_name=FOOBAR,string_val=hello"
```

The sender has the option of indicating *both* a group name and vehicle name as the message destination. It may also specify more than one vehicle name explicitely. Thus the following is allowed:

```
NODE_MESSAGE = "src_node=alpha,dest_name=bravo:charlie:gilda,dest_group=red_team,
               var_name=FOOBAR,string_val=hello"
```

If a destination vehicle is specified twice in the list of destinations or implicitly in the named group, the message will be sent only once to the destination vehicle.

5.4 Visual Artifacts for Rendering Inter-Vehicle Communications

Each time a node report or message is sent to a vehicle by `uFldNodeComms`, another posting is made to the variable `VIEW_COMMS_PULSE`. This message may be subscribed for by another application using it to visually render the commuications events. The screen shot in Figure 8 below shows four vehicles. The two bottom vehicles are sharing node reports indicated by the red and blue comms pulses.

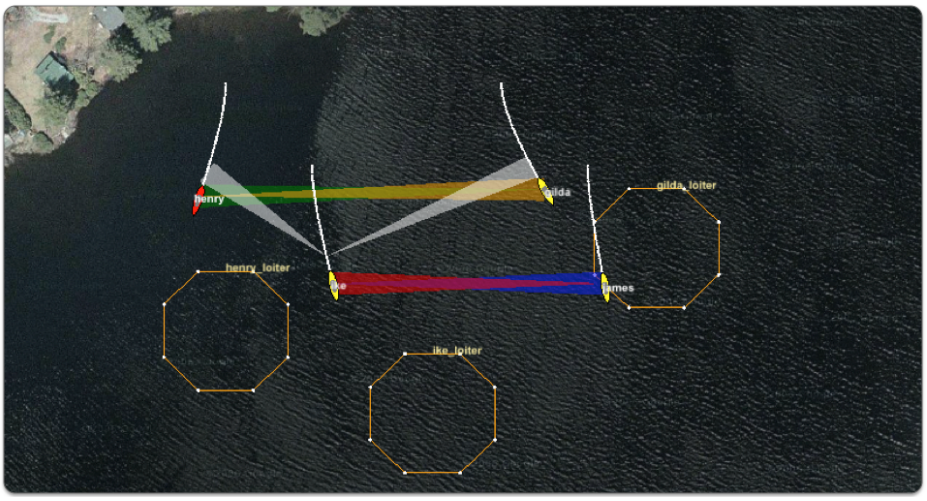


Figure 8: **Communication Pulses:** A visual artifact, a `VIEW_COMMS_PULSE`, is rendered between vehicles when either a node report or node message is generated. The white pulse indicates a node message, and the non-white pulses indicate a node report. The pulse widens from a point at the source vehicle to maximum widtth at the destination vehicle. A pulse will remain rendered in pMarineViewer for some number of seconds after initial post, unless it is replace by a new pulse with the same label.

The top two vehicles are also sharing node reports seen by the yellow and green pulses. The bottom left vehicle has also just sent a node message to the top two vehicles indicated by the two white pulse. Note the pair of white pulses were posted a few seconds prior to the present point in time since they point to vehicle positions just back in the vehicle history. The node reports are updated continously, constantly replacing the pulse just previously posted.

The pulse colors are chosen automatically by `uFldNodeComms`. They may be toggled off in `pMarineViewer` withe `'@'` key. The comms pulse is conveyed in a posting to the variable `VIEW_COMMS_PULSE`. The format is shown with the following example.

```
VIEW_COMMS_PULSE = "sx=109.63,sy=-60.06,tx=30.96,ty=-60.22,beam_width=7,duration=10,
                    fill=0.35,label=JAMES2IKE,edge_color=green,fill_color=red,
                    time=2652414456.59,edge_size=1"
```

Comms pulses may be generated by other applications besides `uFldNodeComms`, and may be consumed and rendered by other applications besides `pMarineViewer`. The definition of the comms pulse object and the methods for serializing and de-serializing betweeen object and string representation may be found in the `lib_geometry` library in the moos-ivp software tree.

5.5 Status Updates to the Terminal

The `uFldNodeComms` terminal output provides a summary of node reports and messages that have been received and re-routed to their recipients. This output may be turned off with the configuration `VERBOSE=false`. The default setting is `true`. An example report is shown in Listing 11 below. The first part of the report (lines 4-14 in this case) summarize first the total node reports received (lines 6-10) and the node reports sent (lines 12-14). At the end of lines 7-10, the elapsed time since the last received node report is shown.

Listing 11 - Example Terminal Output for the uFldNodeComms tool.

```
 0  ****************************************
 1  uFldNodeCommsSummary:
 2  ****************************************
 3
 4  Node Report Summary
 5  ===================================
 6          Total Received: 1297
 7                   GILDA: 326       (0.5)
 8                   HENRY: 325       (1.0)
 9                     IKE: 324       (0.5)
10                   JAMES: 322       (0.5)
11       ------------------
12              Total Sent: 154
13                     IKE: 85
14                   JAMES: 69
15
16
17  Node Message Summary
18  ===================================
19     Total Msgs Received: 3
20                     IKE: 3      (49.3)
21       ------------------
```

```
22              Total Sent: 3
23                   GILDA: 1
24                   HENRY: 2
25       ------------------
26      Total Blocked Msgs: 2
27                 Invalid: 0
28          Stale Receiver: 0
29              Too Recent: 1
30            Msg Too Long: 0
31           Range Too Far: 1
32       ------------------
33               Last Msgs:
34  src_node=ike,dest_group=GROUP12,var_name=FOOBAR,string_val=hello
35  src_node=ike,dest_group=GROUP12,var_name=FOOBAR,string_val=hello
36  src_node=ike,dest_group=GROUP12,var_name=FOOBAR,string_val=hello
```

The summary of node messages is shown in the second half of the report, in lines 17-36 in this case. The total messages received is shown on line 20, with a breakdown of where they have been received from in the following lines. Starting on line 22, a summary of sent node messages is given. First the total sent messages on line 22 and a breakdown of receivers in the following lines. A summary of blocked messages is given next, in this case in lines 26-31. The total number of blocked messages is given first, followed by the possible reasons for blocking in lines 27-31. Finally, the most recent (up to five) messages are dumped to the screen on the last lines of the report.

6 The uFldMessageHandler Utility: Handling Node Messages

The `uFldMessageHandler` application is a tool for handling incoming inter-node messages. In the uField Toolbox typical arrangement, these messages are arriving from a shoreside MOOS community from the `uFldNodeComms` and `pMOOSBridge` applications as shown below in Figure 9.

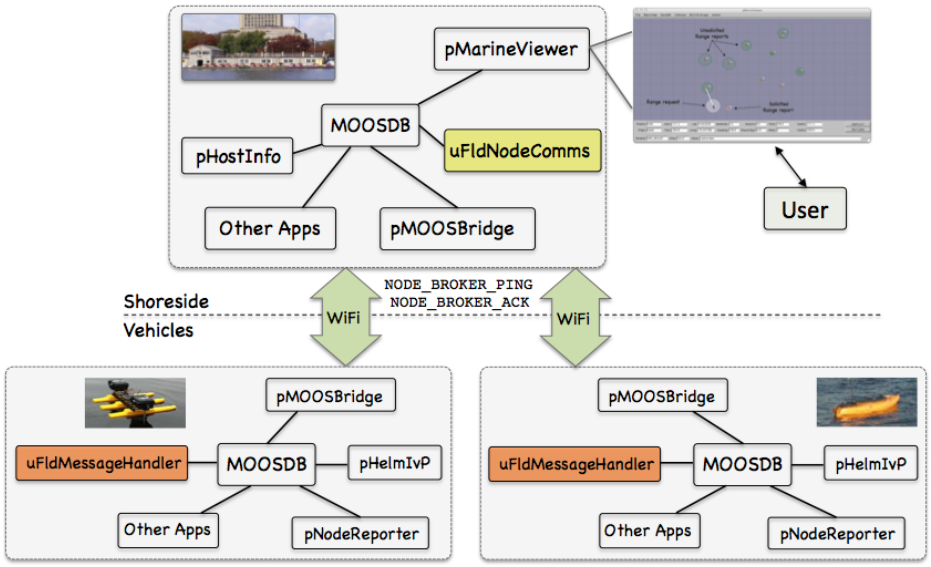


Figure 9: **Typical uFldMessageHandler Topology:** A vehicle (node) sends a message to another vehicle by wrapping the message content and addressee information in a single string sent to the shoreside. On the shoreside, the uFldNodeComms application redirects the message to the appropriate vehicle(s). The message is received on the vehicle by the uFldMessageHandler application which parses the MOOS variable and the variable value from the string and posts the variable-value pair to the local MOOSDB.

The functionality of `uFldMessageHandler` may be paraphrased:

- A source vehicle *alpha* wishes to send a message to vehicle *bravo* of the form `SPEED=2.5`.
- A local message is posted on vehicle *alpha* of the form:

```
NODE_MESSAGE_LOCAL = src_node=alpha,dest_node=bravo,src_var=SPEED,double_val=2.5
```

- The above message is bridged from *alpha* to the shoreside community using `pMOOSBridge`.
- The message is received in the shoreside community as the variable `NODE_MESSAGE` and handled by `uFldNodeComms` and republished as `NODE_MESSAGE_BRAVO`.
- The message is then bridged out to *bravo* using `pMOOSBridge` arriving in vehicle *bravo* as `NODE_MESSAGE`.
- On vehicle *bravo*, the `NODE_MESSAGE` is handled by `uFldMessageHandler`. The source variable and value are parsed and a post to the local MOOSDB on *bravo* is made, `SPEED=2.5`
- A scope on the MOOSDB on *bravo* would show the source of the `SPEED=2.5` posting to be `"uFldMessageHandler"`, and the auxiliary source would show `"alpha"`

6.1 Overview of the uFldMessageHandler Interface and Configuration Options

The uFldMessageHandler application may be configured with a configuration block within a .moos file. Its interface is defined by its publications and subscriptions for MOOS variables consumed and generated by other MOOS applications. An overview of the set of configuration options and interface is provided in this section. If one has access to a command line where the uFldMessageHandler has been built, interface information may also be seen by typing "uFldMessageHandler --interface", and configuration information by typing "uFldMessageHandler --example".

6.1.1 Configuration Parameters of uFldMessageHandler

The following parameters are defined for uFldMessageHandler.

VERBOSE:	If true, terminal output reports are generated on each iteration. The default is true.
STRICT_ADDRESSING:	If true, only messages with a destination specified by dest_node, matching the local community name are processed. Other messages with a destination specified by a group designation are ignored. The default is false.

6.1.2 MOOS Variables Published by uFldMessageHandler

The primary output of uFldMessageHandler to the MOOSDB are the messages posted by parsing incoming NODE_MESSAGE postings. A summary is also posted periodically to recap message handling totals.

UMH_SUMMARY_MSGS:	A summary of total messages, valid messages and rejected messages handled thus far.

6.1.3 MOOS Variables Subscribed for by uFldMessageHandler

The uFldMessageHandler application subscribes to the following MOOS variables:

NODE_MESSAGE: Incoming node messages of the form, by examples:

```
src_node=alpha,dest_node=bravo,src_var=SPEED,double_val=2.2
```

or

```
src_node=alpha,dest_node=bravo,src_var=SPEED,string_val=fast
```

or

```
src_node=alpha,dest_group=redteam,src_var=SPEED,double_val=2.2
```

6.1.4 Command Line Usage of uFldMessageHandler

The uFldMessageHandler application is typically launched with pAntler, along with a group of other shoreside modules. However, it may be launched separately from the command line. The command line options may be shown by typing "uFldMessageHandler --help":

Listing 12 - Command line usage for the uFldMessageHandler tool.

```
 0  ============================================================
 1  Usage: uFldMessageHandler file.moos [OPTIONS]
 2  ============================================================
 3
 4  SYNOPSIS:
 5  ------------------------------------
 6    The uFldMessageHandler tool is used for handling incoming
 7    messages from other nodes. The message is a string that
 8    contains the source and destination of the message as well as
 9    the MOOS variable and value. This app simply posts to the
10    local MOOSDB the variable-value pair contents of the message.
11
12  Options:
13    --alias=<ProcessName>
14        Launch uFldMessageHandler with the given process name
15        rather than uFldMessageHandler.
16    --example, -e
17        Display example MOOS configuration block.
18    --help, -h
19        Display this help message.
20    --interface, -i
21        Display MOOS publications and subscriptions.
22    --version,-v
23        Display the release version of uFldMessageHandler.
24
25  Note: If argv[2] does not otherwise match a known option,
26        then it will be interpreted as a run alias. This is
27        to support pAntler launching conventions.
```

6.1.5 An Example MOOS Configuration Block

To see an example MOOS configuration block, enter the following from the command-line: `"uFldMessageHandler -e"`. This will show the output shown in Listing 13 below.

Listing 13 - Example configuration of the uFldMessageHandler application.

```
 0  ============================================================
 1  uFldMessageHandler Example MOOS Configuration
 2  ============================================================
 3
 4  ProcessConfig = uFldMessageHandler
 5  {
 6    AppTick   = 4
 7    CommsTick = 4
 8
 9    verbose           = true    // the default
10    strict_addressing = false   // the default
11  }
```

6.2 Terminal Output - A Detailed Snapshot of the Broker Status

The `uFldMessageHandler` application produces some useful information to the terminal on every iteration of the application. An example is shown in Listing 14 below. The counter on the end of line 0 is incremented on each iteration of `uFldMessageHandler`.

On line 1, the name of the local community and the number of reports thus far generated to the terminal are printed. In lines 4-9, general tallies are shown of received, invalid, and rejected messages. In lines 11-17, the tallies for received messages sorted by source vehicle are shown. The variable-value columns reflect only the last received message.

Listing 14 - Example terminal output of the uFldMessageHandler tool.

```
 0  ***************************************
 1  uFldMessageHandler Summary: (james)(149)
 2  ***************************************
 3
 4  Overall Totals Summary
 5  ======================================
 6     Total Received Valid: 28
 7                   Invalid: 0
 8                  Rejected: 0
 9      Time since last Msg: 4.59508
10
11  Per Source Node Summary
12  ======================================
13    Source  Total  Elapsed  Variable  Value
14    ------  -----  -------  --------  -----
15     gilda     19      7.5    FOOBAR     35
16     henry      5      4.6    FOOBAR     44
17       ike      4     24.6    FOOBAR     20
18
19  Last Few Messages: (from oldest to newest)
20  ======================================
21  Valid Mgs:
22    src_node=henry,dest_node=all,var_name=FOOBAR,string_val=74
23    src_node=gilda,dest_node=all,var_name=FOOBAR,string_val=35
24    src_node=ike,dest_node=all,var_name=FOOBAR,string_val=20
25    src_node=gilda,dest_node=all,var_name=FOOBAR,string_val=35
26    src_node=henry,dest_node=all,var_name=FOOBAR,string_val=44
27  Invalid Mgs:
28    NONE
29  Rejected Mgs:
30    NONE
```

The information group starting on line 19 shows the last five received valid, invalid and rejected messages. Note that a rejected message may be rejected for being invalid, or if the destination field doesn't match, or if strict addressing is enabled and there is not a precise destination field match.

7 The uFldScope Utility: Multi-Vehicle Status Summary

The uFldScope application is a tool for collecting diverse sets of information regarding a field of vehicles remotely deployed. Suppose, for example, one is interested in monitoring, for each deployed vehicle, the (a) helm mode, (b) total distance travelled, (c) battery level, and (d) the number of times it has visited a certain beacon. Each piece of information may be embedded in one of a number of MOOS variables, perhaps along with a lot of other information of no concern. For example, a typical NODE_REPORT posting contains the helm mode, but the full string may look like:

```
NODE_REPORT= "NAME=alpha,TYPE=UUV,TIME=1252348077.59,X=51.71,Y=-35.50,LAT=43.824981,
              LON=-70.329755,SPD=2.00,HDG=118.85,DEPTH=4.63,LENGTH=3.8,MODE=LOITERING"
```

While there are several methods to scope on the above variable and pick out the helm mode, the goal of the uFldScope tool is to have this information readily visible for each vehicle perhaps alongside other key fields for all vehicles, in a continuously updated simple table like the following:

```
VName       MODE     TripDist  Speed  STREAMING(2)
=====    =======     ========  =====  (15)
alpha    LOITERING      66.8   1.96
bravo         PARK       0.0   0.00
charlie  RETURNING    1466.3   1.05
```

The assumption is that uFldScope is running on a topside computer, interacting with a user, and receiving information on deployed vehicles primarily through node reports or other summary report variables. The typical layout is shown in Figure 10

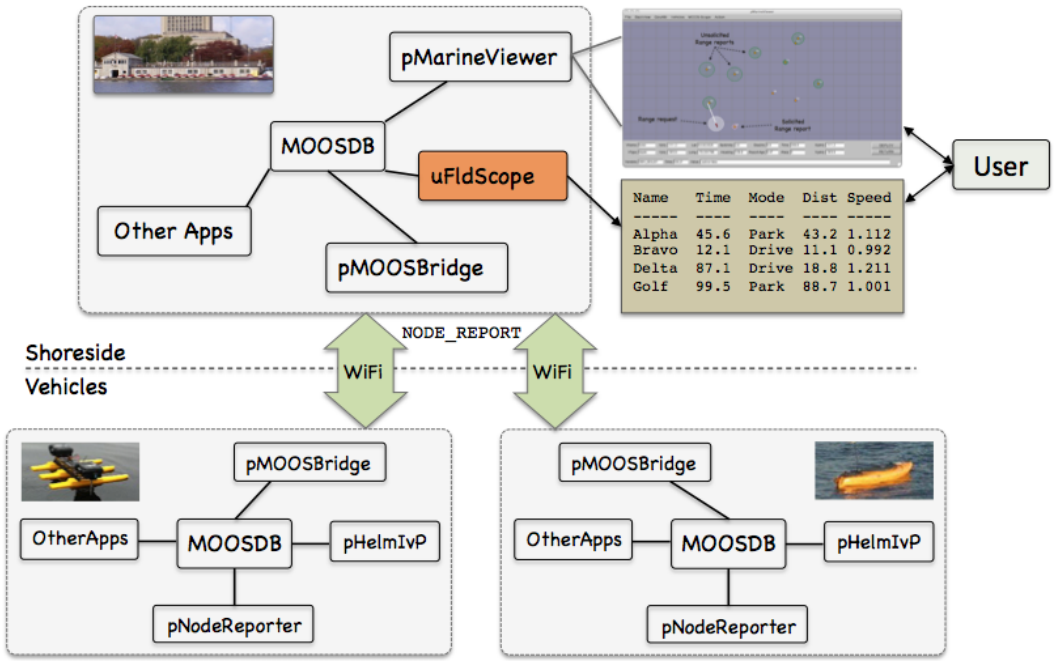


Figure 10: **Typical uFldScope Topology:** A shoreside or topside community is receiving information from several deployed vehicles. Key information is embedded in one of several possible MOOS report variables. The uFldScope tool runs in the topside community to parse key information from the variables and display them in a table format configured by the user.

7.1 Overview of the uFldScope Interface and Configuration Options

The `uFldScope` application may be configured with a configuration block within a `.moos` file. Its interface is defined by its publications and subscriptions for MOOS variables consumed and generated by other MOOS applications. An overview of the set of configuration options and interface is provided in this section.

7.1.1 Configuration Parameters of uFldScope

The following parameters are defined for `uFldScope`. A more detailed description is provided in other parts of this section. Parameters having default values indicate so in parentheses below.

`SCOPE:`	Description info include in main report along with source info.
`LAYOUT:`	An alternative table layout showing only selected fields in each report.

7.1.2 MOOS Variables Published by uFldScope

The primary output of `uFldScope` to the MOOSDB is the report output to the terminal. No publications are made to the MOOSDB.

7.1.3 MOOS Variables Subscribed for by uFldScope

The `uFldScope` application will only subscribe for the MOOS variables prescribed in the `SCOPE` configuration parameter(s).

7.1.4 Command Line Usage of uFldScope

Although the `uFldScope` application may be launched with pAntler, it is typically launched separately from the command line by the user. The command line options may be shown by typing `"uFldScope --help"`:

Listing 15 - Command line usage for the uFldScope tool.

```
 0  Usage: uFldScope file.moos [OPTIONS]
 1
 2  Options:
 3    --alias=<ProcessName>
 4        Launch uFldScope with the given process
 5        name rather than uFldScope.
 6    --example, -e
 7        Display example MOOS configuration block
 8    --help, -h
 9        Display this help message.
10    --version,-v
11        Display the release version of uFldScope.
```

7.1.5 An Example MOOS Configuration Block

As of MOOS-IvP Release 4.2, most if not all MOOS apps are implemented to support the `-e` or `--example` command-line switches. To see an example MOOS configuration block, enter the following from the command-line:

```
$ uFldScope -e
```

This will show the output shown in Listing 16 below.

Listing 16 - Example configuration of the uFldScope application.

```
 0  ============================================================
 1  uFldScope Example MOOS Configuration
 2  ============================================================
 3
 4  ProcessConfig = uFldScope
 5  {
 6    AppTick   = 4
 7    CommsTick = 4
 8
 9    SCOPE = var=NODE_REPORT,key=vname,fld=TIME,alias=Time
10    SCOPE = var=NODE_REPORT,key=vname,fld=MODE
11    SCOPE = var=SPEED_REPORT,key=vname,fld=avg_speed,alias=speed
12    SCOPE = var=ODOMETRY_REPORT,key=vname,fld=trip_dist
13    SCOPE = var=ODOMETRY_REPORT,key=vname,fld=total_dist
14
15    LAYOUT = trip_dist, total_dist
16    LAYOUT = MODE, speed, Time
17  }
```

7.2 Configuring the uFldScope Utility

The `uFldScope` utility may be configured to choose (a) which MOOS variables are scoped, (b) which fields in those messages are scoped, and (c) how that data is presented to the user. In all usage scenarios it is presumed that the messages are strings comprised of comma-separated variable=value pairs. For example:

```
NODE_REPORT= "NAME=alpha,TYPE=UUV,TIME=1252348077.59,X=51.71,Y=-35.50,LAT=43.824981,
             LON=-70.329755,SPD=2.00,HDG=118.85,DEPTH=4.63,LENGTH=3.8,MODE=LOITERING"
```

For each variable specified by the user for scoping, a *key* needs to also be specified identifying the vehicle name. In the above case, the key is the string `"NAME"`.

7.2.1 Configuring Scope Elements

A *scope element* corresponds to particular MOOS message and a particular field in that MOOS message. It also corresponds to a column in the tabular report presented to the user. For example, the second column in the table below, with the header `"MODE"`, corresponds to a scope element deriving its information from postings to `NODE_REPORT`, in the field `"MODE"`.

```
VName         MODE    TripDist  Speed  STREAMING(2)
=====      =======    ========  =====  (15)
alpha    LOITERING        66.8   1.96
bravo         PARK         0.0   0.00
charlie  RETURNING      1466.3   1.05
```

A scope element is configure in the mission configuration file with entries of the form:

```
SCOPE = var=<MOOSVAR>, key=<KEYNAME>, fld=<FIELDNAME>, alias=<ALIAS>
```

The <MOOSVAR> specifies the name of the MOOS variable. Multiple scope elements may use the same MOOS variable. The <KEYNAME> specifies the field in the message used to designate the name of the vehicle. There should be only one right answer for this, and if it is wrongly specified, the column for that scope element will simply be empty. The <FIELDNAME> specifies the other field in the message holding the information of interest. For example, line 10 in Listing 16 is the scope configuration resulting in the second column of the above tabular output. The <ALIAS> is a string to use in the column output header if the user doesn't want to just use the name of scoped field.

7.2.2 Configuring Scope Layouts

By default the tabular output produced by uFldScope contains a column for each scope element. If the number of scope elements is large the user may be interested, at times, in rendering only a subset of the scope elements. The user may define these subsets using the LAYOUT configuration parameter, of the form:

```
LAYOUT = <FLDNAME>, <FLDNAME>, ..., <FLDNAME>
```

The <FLDNAME> specifies the name of the field in a given scope element. Note that it is *possible* that the field names may be the same for two different scope elements. For this reason the field name used in the layout definition is the field name alias. This gives the user the chance to distinguish two otherwise identical field names. If the user does not specify an alias in configuration of scope element, the alias is by default the same as the field name.

For the uFldScope configuration shown in Listing 16, the following five scope elements are rendered to the user as follows.

```
VName          Time                   MODE  speed  trip_dist  total_dist  PAUSED(A)
=====  =============  ====================  =====  =========  ==========  (1800)
gilda  2651922526.73  MODE@ACTIVE:LOITERING   1.06      517.1       517.1
henry  2651922526.54  MODE@ACTIVE:LOITERING   1.17      526.4       526.4
  ike  2651922526.45  MODE@ACTIVE:LOITERING   1.07      520.1       520.1
james  2651922526.65  MODE@ACTIVE:LOITERING   1.16      515.1       515.1
```

Note the "(A)" at the end of the first line. This indicates that all scope elements are being presented. If the user hits the 'l' or 'L' keys the presentation will be toggled through the various layouts configured by the user. In this example case, the following two layouts may be selected:

```
VName          Time                   MODE  speed  PAUSED(1)
=====  =============  ====================  =====  (1802)
gilda   2651923071.3  MODE@ACTIVE:LOITERING   1.17
henry  2651923071.11  MODE@ACTIVE:LOITERING   1.17
  ike  2651923071.03  MODE@ACTIVE:LOITERING   1.18
james  2651923071.24  MODE@ACTIVE:LOITERING   1.17
```

Note that only the *Time*, *MODE*, and *speed* scope elements are produced, corresponding to the layout configured on Line 16 in Listing 16. Also note that the "(A)" on the first line switched to "(1)" to indicate that the first user-configured layout is being used. By hitting 'L' once more, the second user-configured layout will be instead shown:

```
VName  trip_dist  total_dist  PAUSED(2)
=====  =========  ==========  (1801)
gilda     1081.9      1081.9
henry     1072.1      1072.1
  ike     1117.7      1117.7
james     1067.0      1067.0
```

Note that the `"(1)"` on the first line switched to `"(2)"` to indicate that the second user-configured layout is being used. By hitting `'L'` once more, the full table will again be rendered.

7.2.3 Further Control of the Terminal Output

If multiple layouts have been configured, the user may either toggle through the list of layouts with the `'l'` or `'L'` key as mentioned above, or toggle between the last-used user-configured layout and the mode of rendering all scope elements, by using the `'a'` or `'A'` key.

By default the output produced to the terminal is refreshed on each iteration of the `uFldScope` application. This may be useful for watching a trend as time passes. The user may also wish to pause the output to take a careful look at the data. This may be done by hitting the `'p'` or `'P'` keys, or simply the spacebar. Subsequent similar keystrokes will keep the refresh mode in the paused mode, but the output will be refreshed to their current values before again pausing. Returning to the streaming mode may be done by hitting the `'r'` or `'R'` keys. At any time the user may also hit the `'h'` or `'H'` keys for a help menu.

8 The uFldPathCheck Utility: Checking Vehicle Path Properties

The `uFldPathCheck` application is a tool for summarizing a few properties in a field of vehicles remotely deployed. The primary focus is on summarizing the speed and distance travelled for each vehicle. Rather than relying on the vehicles themselves to calculate and report this information, the `uFldPathCheck` tool determines this information independently based on node reports from the vehicles. The assumption is that `uFldPathCheck` is running on a topside or shoreside computer, and receiving information about deployed vehicles primarily through node reports. The typical layout is shown in Figure 11

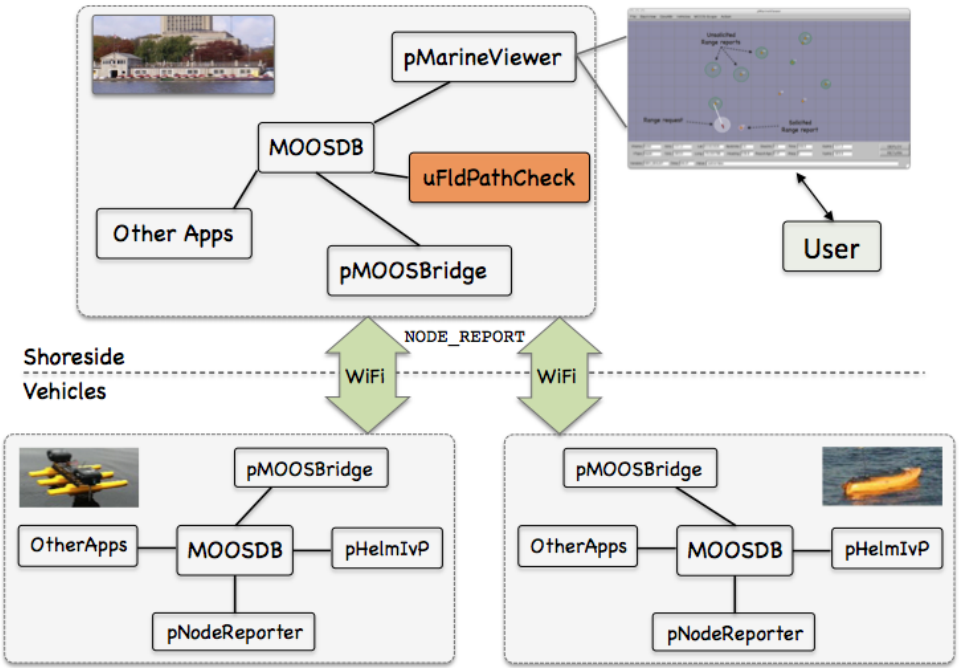


Figure 11: **Typical uFldPathCheck Topology:** A shoreside or topside community is receiving information from several deployed vehicles, in the form of node reports. The node reports contain time-stamped updated vehicle positions, from which the speed and distance measurements are derived and posted to the shoreside MOOSDB.

In short, `uFldPathCheck` subscribes for incoming node reports for any number of vehicles, and keeps a running history of positions for each vehicle. It uses this recent-history to post (a) the current speed noted per vehicle and (b) the distance travelled per vehicle. These two reports are posted in the `UPC_SPEED_REPORT` and `UPC_ODOMETRY_REPORT` variables to the MOOSDB. The odemetry report includes a total distance travelled, and a "trip-ometer" distance travelled since the last trip reset. The trip-ometer may be reset for a vehicle *alpha* when mail is received of the form `UPC_TRIP_RESET=alpha`. Examples of the posted output form are given below in Section 8.1.2.

8.1 Overview of the uFldPathCheck Interface and Configuration Options

The `uFldPathCheck` application may be configured with a configuration block within a `.moos` file. Its interface is defined by its publications and subscriptions for MOOS variables consumed and gen-

erated by other MOOS applications. An overview of the set of configuration options and interface is provided in this section.

8.1.1 Configuration Parameters of uFldPathCheck

The following parameters are defined for `uFldPathCheck`.

`HISTORY:` Length of queue used for calculating present speed. The default is 10.

8.1.2 MOOS Variables Published by uFldPathCheck

The primary output of `uFldPathCheck` to the MOOSDB are the speed and odometry reports.

`UPC_ODOMETRY_REPORT:` The current odometry information for a given vehicle.
`UPC_SPEED_REPORT:` The current speed for a given vehicle.

An example of the odemetry report:
`UPC_ODOMETRY_REPORT="vname=alpha,total_dist=4205.4,trip_dist=1105.2"`

An example of the odemetry report:
`UPC_SPEED_REPORT="vname=alpha,avg_speed=2.21"`

8.1.3 MOOS Variables Subscribed for by uFldPathCheck

The `uFldPathCheck` application subscribes to the following MOOS variables:

`NODE_REPORT:` A node report for a given vehicle from `pNodeReporter`.
`NODE_REPORT_LOCAL:` A node report for a given vehicle from `pNodeReporter`.
`UPC_TRIP_RESET:` The name of a vehicle to have its trip odemeter reset. vehicle.

8.1.4 Command Line Usage of uFldPathCheck

The `uFldPathCheck` application is typically launched with pAntler, along with a group of other shoreside modules. However, it may be launched separately from the command line. The command line options may be shown by typing `"uFldPathCheck --help"`:

Listing 17 - Command line usage for the uFldPathCheck tool.

```
0  Usage: uFldPathCheck file.moos [OPTIONS]
1
2  Options:
3    --alias=<ProcessName>
4        Launch uFldPathCheck with the given process
5        name rather than uFldPathCheck.
6    --example, -e
7        Display example MOOS configuration block
8    --help, -h
9        Display this help message.
10   --version,-v
11       Display the release version of uFldPathCheck.
```

8.1.5 An Example MOOS Configuration Block

As of MOOS-IvP Release 4.2, most if not all MOOS apps are implemented to support the `-e` or `--example` command-line switches. To see an example MOOS configuration block, enter the following from the command-line:

```
$ uFldPathCheck -e
```

This will show the output shown in Listing 18 below.

Listing 18 - Example configuration of the uFldPathCheck application.

```
 0  ============================================================
 1  uFldPathCheck Example MOOS Configuration
 2  ============================================================
 3
 4  ProcessConfig = uFldPathCheck
 5  {
 6    AppTick   = 4
 7    CommsTick = 4
 8
 9    HISTORY   = 10  (Default)
10  }
```

8.2 Usage Scenarios the uFldPathCheck Utility

The motivation for this tool is to have a module running on the shoreside capable of being used in a competition scenario. Especially in a simulated competition, there may be a need to limit the upper speed of a participating vehicle to ensure a level playing field between competitors. Since the vehicle simulators may be running on separate machines with particpants merely reporting node reports, there is no way to directly control the upper speed of the vehicles. By *monitoring* the upper speed, this leaves open the option of either (a) disqualifying a vehicle caught moving too fast, or (b) imposing a penalty on a speeding vehicle. The penalty chosen is outside the scope of this module but may include reducing the number of points awarded for certain accomlishments, adding more noise to simulated sensors, and so on. Since this usage scenario implies a "non-compliant" vehicle, we don't want to base the monitored speed on a value reported by the vehicle. Instead we calculate the speed based on successive time-stamped node reports with positions.

Likewise, the odometry information calculated and posted is also intended for use in a competition context. Conceivably, part of a competition may require a vehicle to periodically "re-fuel" after travelling a certain distance. By having the odemetry information calculated independently on the shoreside, constraints on total distance, or fuel calculations may be generated to impose on vehicle competitions. The `uFldPathCheck` application accepts the `UPC_ODOMETRY_RESET=VNAME` posting to reset the trip-ometer for a given vehicle, presumably after a re-fueling event is noted. The odometry information may also be used directly in competitions where minimizing the total path length is the primary objective.

9 The uFldHazardSensor: Simulating an Simple Hazard Sensor

The `uFldHazardSensor` application is a tool for simulating an on-board sensor that processes sonar image data and (a) detects image components that may represent a hazard, and (b) further classifies the detected components as being either a hazard or benign object. The idea is shown in Figure 12. The user configures the sensor by choosing one of N swath widths available to the given sensor, and by choosing a probability of detection, P_D. Based on these two user choices, and the particular performance characteristics of the sensor and sensor algorithms, a probability of false alarm, P_{FA} and probability of correct classification, P_C, follow.

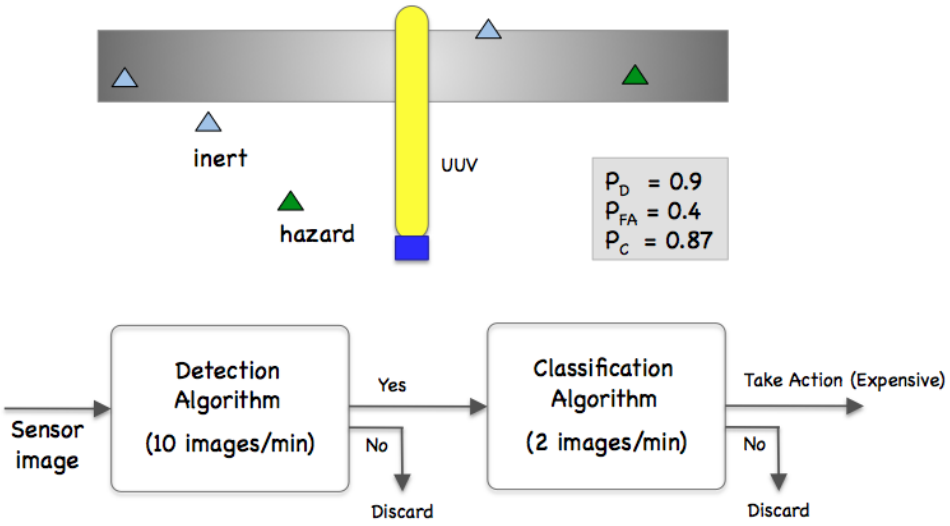


Figure 12: **Simulated Hazard Sensor:** A vehicle processes a series of sensor images which may or may not contain an object. The detection algorithm processes the image and rejects images it believes does not contain contain hazardous objects. It passes on images containing possible hazardous objects to a classifier, which makes a determination for each object in an incoming image as to whether or not the object is hazardous or benign.

In the `uFldHazardSensor` application, the objects, their classifications, and vehicle locations are known to the simulator, and a tidy `UHZ_HAZARD_REPORT` message is sent to the vehicle(s) as a proxy to the actual hazard sensor and the calculations that would otherwise reside on the vehicle. The simulated sensor is configured to have a finite list of sensor settings, each consisting of a swath width, ROC curve, and probability of correct classification. It's up to the user to choose the sensor setting and choose a P_D on the associated ROC curve, which determines the prevailing P_{FA}.

Typical Simulator Topology

The typical module topology is shown in Figure 13 below. Multiple vehicles may be deployed in the field, each periodically communicating with a shoreside MOOS community running a single instance of `uFldHazardSensor`. Each vehicle regularly sends a node report to the shoreside community, read by `uFldHazardSensor`. A vehicle "activates" its sensor by beginning to send a steady stream of messages to the shoreside under the variable `UHZ_SENSOR_REQUEST`. Each time the simulator receives this request, it will assess the vehicle's current position, and sensor settings, and may (depending

on how the dice are rolled and if a detection is made) post a UHZ_HAZARD_REPORT to be bridged out to the given vehicle.

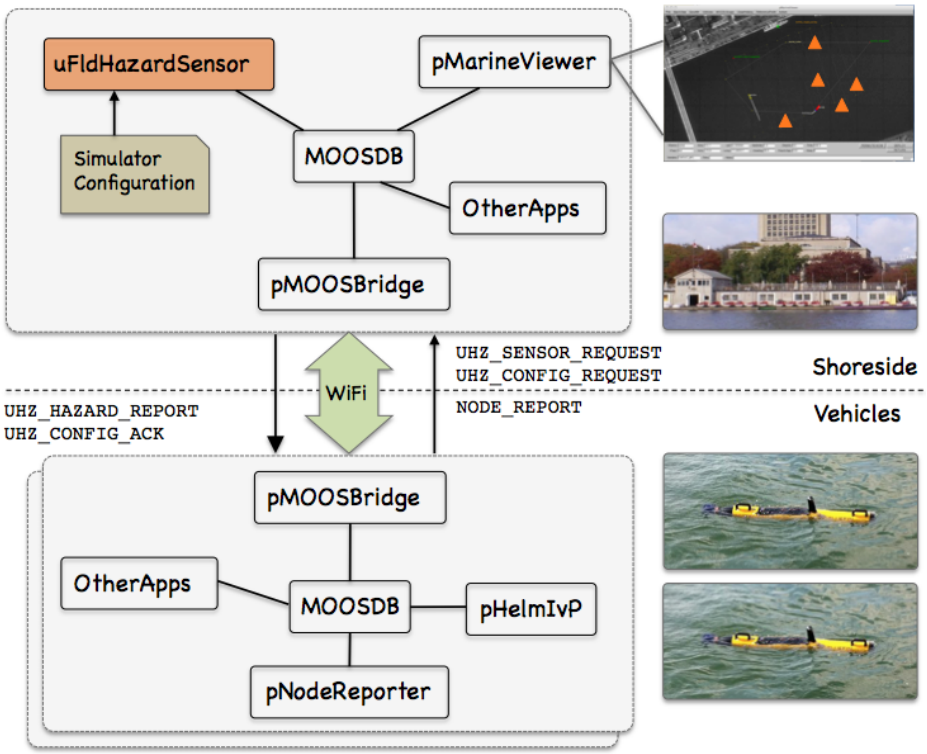


Figure 13: **Typical uFldHazardSensor Topology:** The simulator runs in a shoreside computer MOOS community and is configured with a hazard field containing both hazardous and benign objects. Vehicles accessing the simulator send a steady stream of messages (UHZ_SENSOR_REQUEST) and node reports to the shoreside community. The simulator continuously checks the connected vehicle's position against objects in the hazard field, and the sensor settings. When/if an object comes into sensor range, the simulator rolls the dice and if a detection is made, will send a UHZ_HAZARD_REPORT message to the vehicle. The vehicle may periodically re-configure its sensor setting by posting to UHZ_CONFIG_REQUEST. If the configuration request is acceptable, the simulator will respond with a message to UHZ_CONFIG_ACK bridge back out to the vehicle.

If running a pure simulation (no deployed vehicles), both MOOS communities may simply be running on the same machine configured with distinct ports. The pMOOSBridge application is shown here for communication between MOOS communities, but there are other alternatives for inter-community communication and the operation of uFldHazardSensor is not dependent on the manner of inter-communication communications.

9.1 The uFldHazardSensor Interface and Configuration Options

The uFldHazardSensor application may be configured with a configuration block within a .moos file. Its interface is defined by its publications and subscriptions for MOOS variables consumed and generated by other MOOS applications. An overview of the set of configuration options and interface is provided in this section.

9.1.1 Configuration Parameters of uFldHazardSensor

The following parameters are defined for `uFldHazardSensor`. A more detailed description is provided in other parts of this section. Parameters having default values indicate so in parentheses below.

DEFAULT_HAZARD_SHAPE:	Shape for rendering hazard objects ("triangle").
DEFAULT_HAZARD_COLOR:	Color for rendering hazard objects ("lightblue").
DEFAULT_HAZARD_WIDTH:	Width for rendering hazard objects (8).
DEFAULT_BENIGN_SHAPE:	Shape for rendering benign objects ("triangle").
DEFAULT_BENIGN_COLOR:	Color for rendering benign objects ("lightblue").
DEFAULT_BENIGN_WIDTH:	Width for rendering benign objects (8).
OPTIONS_SUMMARY_INTERVAL:	Duration (secs) between posting of options summary (10).
HAZARD_FILE:	Names a file containing the hazard field configuration.
SWATH_TRANSPARENCY:	Transparency used for rendering swath (0.25).
SEED_RANDOM:	If true, random number generated is seeded (false).
SENSOR_CONFIG:	Describes one of possibly many sensor config options.
SHOW_HAZARDS:	If false, hazard field visuals are not posted (true).
SHOW_SWATH:	If false, vehicle sensor swath visuals are not posted (true).
SHOW_REPORTS:	Duration attached to report circle postings (unlimited).
SWATH_LENGTH:	Extend of the sensor swath going bow to stern (5).
VERBOSE:	If true, turns on more verbose output to the terminal (false).

9.1.2 MOOS Variables Published by uFldHazardSensor

The primary output of `uFldHazardSensor` to the MOOSDB is posting of sensor reports, visual cues for the sensor reports, and visual cues for the hazard objects themselves.

UHZ_HAZARD_REPORT:	A report on a detection and classification made by the sensor for an object in the hazard field. It includes the name of the vehicle.
UHZ_HAZARD_REPORT_NAMEJ:	A report on a detection and classification made by the sensor (on vehicle `NAMEJ`) for an object in the hazard field.
UHZ_OPTIONS_SUMMARY:	A report the possible sensor settings available to the user.
UHZ_CONFIG_ACK:	An acknowledgment message sent to the vehicle verifying a requested sensor setting.
VIEW_CIRCLE:	A visual artifact for rendering a circle around a hazard, indicating the detection and classification.
VIEW_MARKER:	A visual artifact for rendering objects in the hazard field.
VIEW_POLYGON:	A visual artifact for rendering a rectangle around a vehicle, indicating a vehicle sensor field moving with the vehicle.

Example postings:

```
UHZ_HAZARD_REPORT        = vname=archie,x=51,y=11.3,hazard=true,label=12
```

```
  UHZ_HAZARD_REPORT_ARCHIE = x=51,y=11.3,hazard=true,label=12
  UHZ_CONFIG_ACK           = vname=archie,width=20,pd=0.9,pfa=0.53,pclass=0.91
  UHZ_OPTIONS_SUMMARY      = width=10,exp=6,pclass=0.9:width=25,exp=4,pclass=0.85
```

The vehicle name may be embedded in the MOOS variable name to facilitate distribution of report messages to the appropriate vehicle with `pMOOSBridge`.

9.1.3 MOOS Variables Subscribed for by uFldHazardSensor

The `uFldHazardSensor` application will subscribe for the following four MOOS variables:

`UHZ_SENSOR_REQUEST`:	A message from the vehicle indicating the sensor is active.
`UHZ_SENSOR_CONFIG`:	A message from the vehicle requesting one of the possible sensor settings and P_D choice from the ROC curve resulting from the sensor settings.
`NODE_REPORT`:	A report on a vehicle location and status.
`NODE_REPORT_LOCAL`:	A report on a vehicle location and status.

Example postings

```
  UHZ_SENSOR_REQUEST = vname=archie
  UHZ_CONFIG_REQUEST = vname=archie,width=50,pd=0.9
```

9.1.4 Command Line Usage of uFldHazardSensor

The `uFldHazardSensor` application is typically launched as a part of a batch of processes by pAntler, but may also be launched from the command line by the user. The command line options may be shown by typing `"uFldHazardSensor --help"`:

Listing 19 - Command line usage for the uFldHazardSensor tool.

```
============================================================
Usage: uFldHazardSensor file.moos [OPTIONS]
============================================================

Options:
  --alias=<ProcessName>
      Launch uFldHazardSensor with the given process
      name rather than uFldHazardSensor.
  --example, -e
      Display example MOOS configuration block.
  --help, -h
      Display this help message.
  --interface, -i
      Display MOOS publications and subscriptions.
  --version,-v
      Display release version of uFldHazardSensor.
  --verbose=<setting>
      Set verbosity. true or false (default)

Note: If argv[2] does not otherwise match a known option,
      then it will be interpreted as a run alias. This is
      to support pAntler launching conventions.
```

9.1.5 An Example MOOS Configuration Block

To see an example MOOS configuration block, enter the following from the command-line:

```
$ uFldHazardSensor -e
```

This will show the output shown in Listing 20 below.

Listing 20 - Example configuration of the uFldHazardSensor application.

```
============================================================
uFldHazardSensor Example MOOS Configuration
============================================================
ProcessConfig = uFldHazardSensor
{
  // Configuring visual preferences
  default_hazard_shape = triangle                // default
  default_hazard_color = green                   // default
  default_hazard_width = 8                       // default

  default_benign_shape = triangle                // default
  default_benign_color = light_blue              // default
  default_benign_width = 8                       // default
  swath_transparency   = 0.25                    // default

  sensor_config = width=25, exp=4, pclass=0.80
  sensor_config = width=50, exp=2, pclass=0.60
  sensor_config = width=10, exp=6, pclass=0.93
  hazard_file   = hazards.txt
  swath_length  = 5                              // default
  seed_random   = false                          // default
  verbose       = true                           // default

  show_hazards  = true  // default                // default
  show_swath    = true  // default                // default
  show_reports  = 60    // seconds (unlimited if unspecified)
}
```

9.2 Configuring the Hazard Field

The hazard field is a list of objects, each containing the below seven fields. The first three must be specified explicitly. The last four may be left unspecified, falling back to default values:

- x position
- y position
- type (`hazard` or `benign`)
- label (unique between entries)
- color (rendering hint, default: green for hazards, light_blue for benign objects)
- shape (rendering hint, default: triangle for both hazards and benign objects)
- width (rendering hint, default: 8 meters for both hazards and benign objects)

9.2.1 An Example Hazard Field

The field in Figure 14 below shows an example field. This is also the hazard field used in the example mission, `m10_jake`, described in Section 9.7.

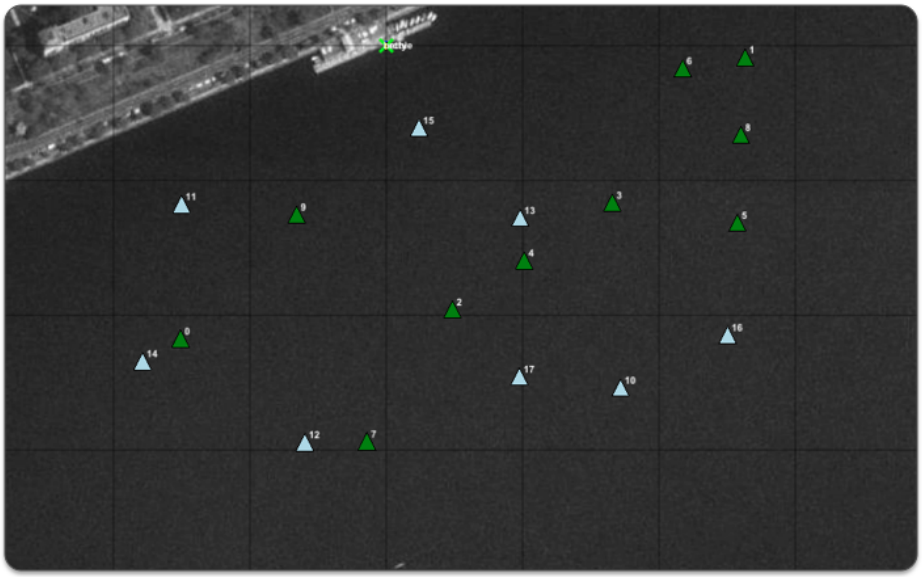


Figure 14: **Simulated Hazard Field:** A hazard field with 18 objects is shown, some are hazardous objects, some are benign objects.

The hazard field may be configured with either entries in the `uFldHazardSensor` configuration block, or by reading in a hazard field configuration file. The format for a configuration line is the same in both cases. For the hazard field shown in Figure 14, the configuration lines used were:

```
hazard = x=-151,y=-217.3,type=hazard
hazard = x=263.20001,y=-8.5,type=hazard
hazard = x=48.5,y=-195.7,type=hazard
hazard = x=165.90001,y=-116.4,type=hazard
hazard = x=101.3,y=-159.3,type=hazard
hazard = x=257.60001,y=-131.3,type=hazard
hazard = x=217.3,y=-16.7,type=hazard
hazard = x=-14.2,y=-293.60001,type=hazard
hazard = x=260.20001,y=-66.2,type=hazard
hazard = x=-65.8,y=-125.2,type=hazard
hazard = x=171.90001,y=-253.7,type=benign
hazard = x=-150.3,y=-117.5,type=benign
hazard = x=-59.8,y=-294.10001,type=benign
hazard = x=98.2,y=-127.7,type=benign
hazard = x=-178.8,y=-234,type=benign
hazard = x=24,y=-61,type=benign
hazard = x=250.3,y=-214.6,type=benign
hazard = x=97.7,y=-245.5,type=benign
```

Note that no label information was provided for any entry. The label used by default is the index of the object being added.

9.2.2 Automatically Generating a Hazard Field

There also exists in the moos-ivp tree a small utility call `gen_hazards`. This simple command line tool will generate a list of randomly generated objects of a specified type, given a convex polygon as input. For example:

```
$ gen_hazards --objects=5,hazard --objects=6,benign --polygon="0,0:50,0:50,50:0,50"
x=20.1,y=13.8,type=hazard
x=45.2,y=37.9,type=hazard
x=47.8,y=4.7,type=hazard
x=32.6,y=38,type=hazard
x=36.5,y=38.4,type=hazard
x=38.6,y=6.3,type=benign
x=32.7,y=25.1,type=benign
x=30.9,y=39.5,type=benign
x=30.3,y=23.1,type=benign
x=14.1,y=39.5,type=benign
x=13,y=21.5,type=benign
```

9.3 Configuring the Possible Sensor Settings

The `uFldHazardSensor` simulator needs to be configured with one or more *sensor settings*. A sensor setting is comprised of the following 3-tuple:

- Swath width
- ROC curve exponent
- Classifier constant

These tuples are set in the `uFldHazardSensor` configuration block, and may then be chosen dynamically by the autonomy system on the vehicle by posting to the MOOS variable `UHZ_CONFIG_REQUEST`, by simply naming the requested swath width.

9.3.1 Sensor Swath Width Options

The sensor swath width specifies the width of the sensor field at any given moment, from port to starboard. The sensor range on either side then is simply half the swath width. The swath *length* is set in the `uFldHazardSensor` configuration block and remains the same regardless of the swath width.

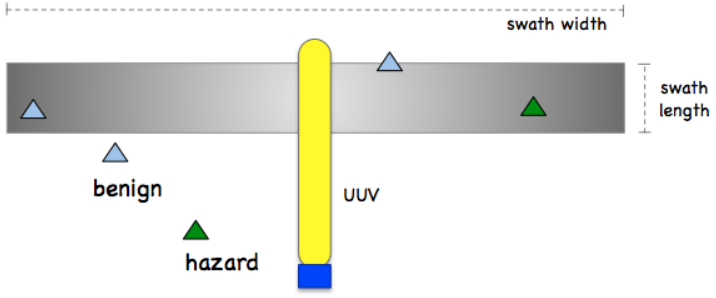


Figure 15: **Sensor Swath Parameter:** The swath width is a parameter that may be re-configured dynamically by the autonomy system. The swath length is set once in the `uFldHazardSensor` configuration block.

9.3.2 Sensor ROC Curve Configuration Options

The ROC curves used in this simulator are based on a simple relationship between probability of detection (P_D) and probability of false alarm (P_{FA}). An example ROC curve is shown in 16. This curve represents

$$P_D = P_{FA}^{0.25}$$

The user of a given sensor with this characteristic ROC curve must decide where to set the detection threshold (P_D). By selection a higher P_D, one must also have to live with a higher P_{FA}. Since the user usually approaches this problem by choosing the P_D, the curve in the figure below could also be described as:

$$P_{FA} = P_D^4$$

The P_{FA} typically follows from a user-chosen P_D. For this reason, the `uFldHazardSensor` simulator is configured by identifying the ROC curve solely by the exponent value in the above function.

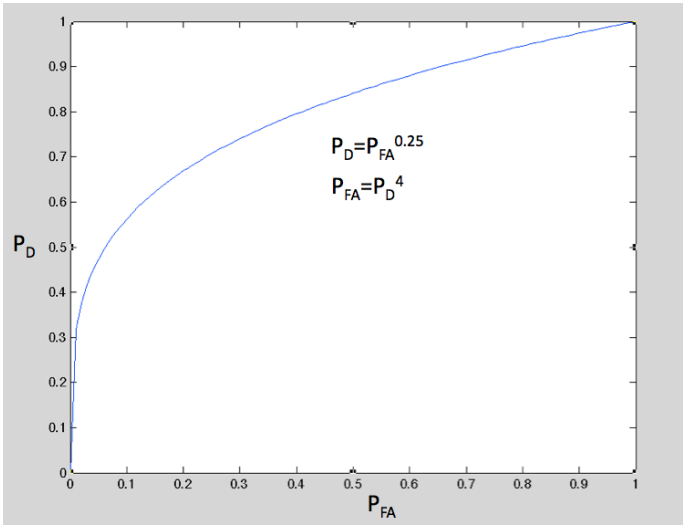


Figure 16: **Example Receiver Operating Characteristic (ROC) Curve:** The ROC curve relates the probability of detection (P_D) on the y-axis, vs. the probability of false alarm (P_{FA}) on the x-axis.

The above ROC may correspond to, for example, the below sensor configuration in the `uFldHazardSensor` configuration block:

```
sensor_config = width=25, exp=4, pclass=0.80
```

The sensor may be configured with more than one ROC curve, typically trading off a more desirable ROC curve for a small sensor swath. For example, the simulated sensor may be configured with the following five options in the configuration file:

```
sensor_config = width=80, exp=2, pclass=0.60
sensor_config = width=65, exp=4, pclass=0.75
sensor_config = width=50, exp=6, pclass=0.85
sensor_config = width=30, exp=12, pclass=0.93
sensor_config = width=18, exp=20, pclass=0.97
```

The idea is shown in Figure 17. When the sensor is configured with these options, it presents the user with *two* choices to make: the sensor width, and choice for P_D. This is done with the `UHZ_CONFIG_REQUEST` interface described in Section 9.3.4.

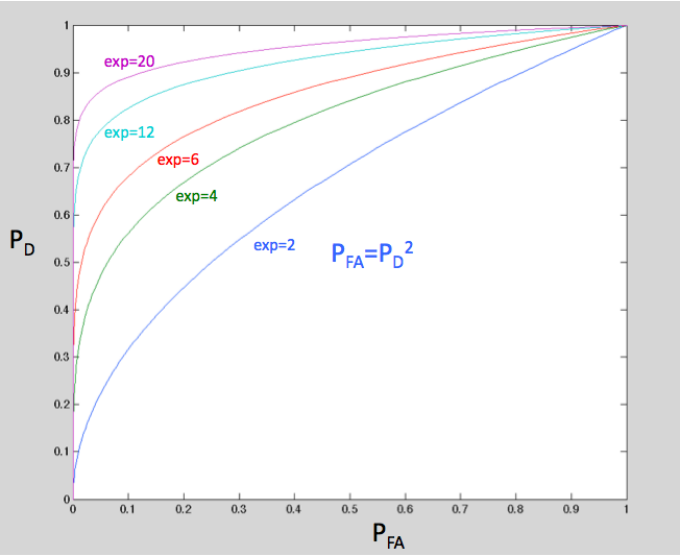


Figure 17: **Example ROC Curve Configuration Options:** By altering a single component, the exponent component of the expression $P_{FA} = P_D^{exp=2,4,6,12,20}$, the ROC curve characteristics may be varied from least desirable ($exp = 1$) to most desirable ($exp = 20$ or higher).

If the sensor user (a given vehicle) does not request a particular swath width and P_D, the simulator will choose one of the options as a default. The default swath width is selected by choosing the highest width that is not more than the average between the highest and lowest width. In the settings above, for example, the `width=30` is the highest width not greater than the average of the two extremes (49). The default P_D setting is 0.9 if left unspecified.

9.3.3 Classification Configuration Options

Classification refers to the process of handling a detected object and determining if it is either a hazard or a benign object (false alarm). The classification stage is shown in Figure 12. In the simulator, this is simply implemented by rolling the dice according to a single probability metric P_C. This is the probability that the classification determination is correct. Since P_C=1 and P_C=0 provide the same net utility, the range of values for P_C is from $[0.5, 1]$. This value is tied to a particular swath width and ROC curve characteristic. The idea is that a smaller swath width allows the sensor to provide a denser set of image data for a smaller area, which leads to both the

better ROC curve as well as crisper images for a classification algorithm (even if the classifying agent is a human examining images off-board).

9.3.4 Dynamic Resetting of the Sensor

The `uFldHazardSensor` simulator is configured with a set of configuration options, as described above, of which one must be chosen, along with a chosen P_D. This is done by the vehicle posting a configuration request of the form:

```
UHZ_CONFIG_REQUEST = vname=archie, width=32, pd=0.95
```

The first field tells the simulator which vehicle is making the request. The second field identifies the ROC curve and swath width. The last field identifies the point on the ROC curve and thus determines the P_{FA}. The `uFldHazardSensor` simulator processes this request by matching up the request to the set of possible options. If the exact swath width requested is not available, the next lowest is chosen. If the requested swath width is less than the lowest option, the lowest option is chosen. The simulator sends an acknowledgment to the vehicle in the form of:

```
UHZ_CONFIG_ACK_ARCHIE = width=30,pd=0.9,pfa=0.28,pclass=0.93
```

The bridging of the variable `UHZ_CONFIG_ACK_ARCHIE` to the vehicle archie is handled by `pMOOSBridge`, typically with dynamic registrations automatically configured by `uFldShoreBroker`.

9.3.5 Posting of Sensor Configuration Options

The sensor configuration options may be known to other MOOS processes by subscribing to the MOOS variable, `UHZ_OPTIONS_SUMMARY`. The following would be published:

```
UHZ_OPTIONS_SUMMARY = width=25,exp=4,pclass=0.85:width=50,exp=2,pclass=0.60 \
                      width=10,exp=6,pclass=0.93
```

corresponding to the following configuration of `uFldHazardSensor`:

```
sensor_config = width=25, exp=4, pclass=0.85
sensor_config = width=50, exp=2, pclass=0.60
sensor_config = width=10, exp=6, pclass=0.93
```

Since this information rarely if ever changes after its first posting, it is not posted very often. By default it is posted once every ten seconds. Since this posting is typically occuring on a shoreside computer, bridged to a remote vehicle, a long delay may not be desirable. This interval may be altered, for example to 5 seconds, by configuring `options_summary_interval=5`.

9.4 Configuring the Simulator Visual Preferences

As shown in Figure 13, the `uFldHazardSensor` is typically running in the shoreside community, alongside a GUI application like `pMarineViewer`. Certain messages are posted by `uFldHazardSensor` for visualization by default. They may be shut off, or have their properties altered by configuring the simulator as desired.

9.4.1 Configuring the Sensor Field Swath Rendering

The sensor field is normally rendered as a rectangle that moves along with the vehicle. This rectangle is drawn to exactly correspond to the sensor field. Rendering in `pMarineViewer` is accomplished by a posting of `VIEW_POLYGON` by `uFldHazardSensor`. This may be shut off with the following configuration:

```
show_swath = false
```

The swath is slightly transparent to allow for the objects to be seen under the swath. The default transparency is 0.2, but may be changed with the following configuration:

```
swath_transparency = 0.60
```

9.4.2 Configuring the Hazard Field Renderings

Rendering of the hazard field is done at the start of `uFldHazardSensor` by producing a `VIEW_MARKER` posting once for each object in the field. This may be disabled with:

```
show_hazards = false
```

Although each object in the hazard file may be configured with color, shape, and width, the default values may be provided for any object that leaves any one of these fields unspecified.

```
default_hazard_shape = triangle
default_hazard_color = green
default_hazard_width = 8
default_benign_shape = triangle
default_benign_color = green
default_benign_width = 8
```

9.4.3 Configuring the Sensor Report Renderings

Sensor reports are rendered as circles around the objects in the hazard field. If a detection is not made, no circle is rendered. If a detection is made and classified as a hazard, a yellow circle is rendered. If classified as benign, a white circle is rendered. All circle renderings are accomplished by `uFldHazardSensor` making a posting to `VIEW_CIRCLE`, then processed by `pMarineViewer`.

When multiple vehicles are using the simulator, it may be confusing after some time to know which vehicle was responsible for which circle rendered. For this reason, the `uFldHazardSensor` may be configured to have the circles to disappear after some number of seconds. This can be done with:

```
show_reports = 60
```

The above will result in each report circle being rendered for 60 seconds before disappearing from view.

9.5 Sensor Processing Algorithm

The sensor simulator handles multiple vehicles and multiple vehicle sensor setting choices. For each vehicle, the simulator is receiving node reports (`NODE_REPORT`), allowing the simulator to know each vehicle's present position. Of course the simulator also knows the hazard field configuration. Armed with these three pieces of information, (a) the sensor setting, (b) the vehicle position, and (c) the hazard field, it repeatedly operates in the manner shown in Figure 18 below.

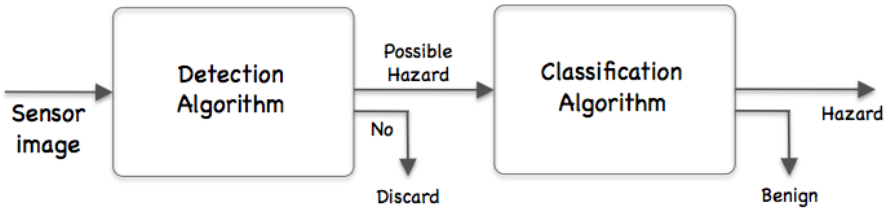


Figure 18: **Simulated Hazard Sensor:** Each time an object comes into the sensor field, it is processed once in the above chain. Once it goes out of the field, it may be processed again later if it comes back into the sensor field. If it does not pass the detection threshold, no action is taken by the simulator. If the object makes it to the classifier, the simulator will generate a report to the vehicle about the detection, and the classification determination.

Each time an object comes into the sensor field for a vehicle, the simulator makes a detection determination. If the object is a hazard (ground-truth identified as such in the hazard configuration file), the simulator will report a detection with probability P_D. If the object is benign, the simulator will report a detection with probability P_{FA}. If no detection is made, no further communication or action is taken by the simulator. If a detection *is* declared, the classification is determined by the simulator. A *correct* classification is made with probability P_C. In either classification case, a report is generated by the simulator to be bridged to the vehicle. Two report postings are made.

```
UHZ_HAZARD_REPORT_ARCHIE = x=51,y=11.3,hazard=true,label=12
UHZ_HAZARD_REPORT        = vname=archie,x=51,y=11.3,hazard=true,label=12
```

The first report is bridged to the vehicle. The second report allows one to scope on a single MOOS variable in the shoreside community to monitor all hazard reports.

9.6 Console Output Generated by uFldHazardSensor

Console output is generated by `uFldHazardSensor` in either the verbose mode or non-verbose mode. The verbose mode is enabled by default, and may be turned off with the line `verbose=false` in the configuration file.

9.6.1 Console Output in the Non-Verbose Mode

In the non-verbose mode, output is limited to single-character outputs, each indicating a different action taking place within the simulator. The meanings are given by:

- `'*'` is written on each iteration always.

- 'N' is written for each incoming node report handled.
- '$' is written for each sensor request.
- 'x' is written for a bad sensor config request (vehicle name mismatch).
- 'X' is written for a bad sensor config request (out-of-range value).
- '!' is written if a sensor config request fails due to null set of settings to choose from.

The above single-character messages have their counterparts in the verbose mode, in the "Internal Memos" section described below.

9.6.2 Console Output in the Verbose Mode

If the verbose setting is turned on, a succinct report is generated as the simulator progresses. Example output is shown below in Listing 21. The number on line 1 is incremented each time the report is written. The block of output in lines 4-10 shows the list of possible sensor configurations. Lines 12-16 show the sensor status for each vehicle known to the simulator. The first column names the vehicle. The next four columns indicate the prevailing sensor setting and chosen point on the ROC curve. The second-to-last column shows the number of times the given vehicle requested a reset in its sensor settings, and how long since the last reset request in parentheses. The last column shows the number of object detections made by the simulator for the given vehicle.

The last block of messages, beginning on line 18 in this case, may show a number of status or warning messages. The number times a message has been duplicated is shown in parentheses. The output on line 23, for example, is useful for confirming that the hazard field file has been properly read. The output on line 21 is useful for showing that vehicle archie is connected and its sensor is "on" since the counter for this message continues to increment.

Listing 21 - Example uFldHazardSensor console output.

```
 0  ****************************************
 1  uFldHazardSensor Summary:(39)
 2  ****************************************
 3
 4  Sensor Configuration Options
 5  ========================================
 6     Width     Exp  Classify
 7    ------  ------  --------
 8      10.0     6.0     0.930
 9      25.0     4.0     0.850
10      50.0     2.0     0.600
11
12  Sensor Settings for known vehicles:
13  ====================================
14  VName      Width      Pd     Pfa  Pclass       Resets  Detects
15  -------    -----    ----    ----  ------    ---------  -------
16  archie      25.0   0.900   0.656   0.850       1(167)        2
17
18  Internal Memos:
19  ====================================
20   (3) Detect/Classify report sent to vehicle: archie
21   (242) Sensor request received from: archie
22   (1) Setting sensor settings for:archie
23   (1) Total hazards: 18
```

9.7 The Jake Example Mission Using uFldHazardSensor

The *Jake* mission is distributed with the MOOS-IvP source code and contains a ready example of the `uFldHazardSensor` application, configured with hazard field in an included text file. Assuming the reader has downloaded the source code available at `www.moos-ivp.org` and built the code according to the discussion in Section 1.5, the *Jake* mission may be launched by:

```
$ cd moos-ivp/ivp/missions/m10_jake/
$ ./launch.sh 10
```

The argument, 10, in the line above will launch the simulation in `10x` real time. Once this launches, the `pMarineViewer` GUI application should launch and the mission may be initiated by hitting the `DEPLOY` button.

9.7.1 What is Happening in the Jake Mission

The Jake mission is comprised of two simulated vehicles, *archie* and *betty*. There are three MOOS communities launched, one for the shoreside and one each for the two vehicles. See Figure 13. The `uFldHazardSensor` simulator is running in the shoreside community. The Jake mission is comprised of two phases, the *broad-area-search* phase and the *reacquire* phase. In this mission, archie handles the first phase, passes his results to betty, who handles the second phase.

The Broad Area Search Phase

In the *broad-area-search* phase, a search region is given to archie, in which it is to search for a set of objects, some of which may be hazardous. Knowing nothing a priori about the location of the objects, only the region containing them, archie executes a lawnmower search pattern over this area, as shown in Figure 19. The snapshot in the figure depicts archie having executed most of its pattern, to the West proceeding East. The hazard field is rendered with actual hazards drawn in green triangles, and benign objects drawn in light blue squares. The circles represent detections reported by `uFldHazardSensor`. The yellow circles represent objects classified as hazards, and the white circles represent objects classified as benign.

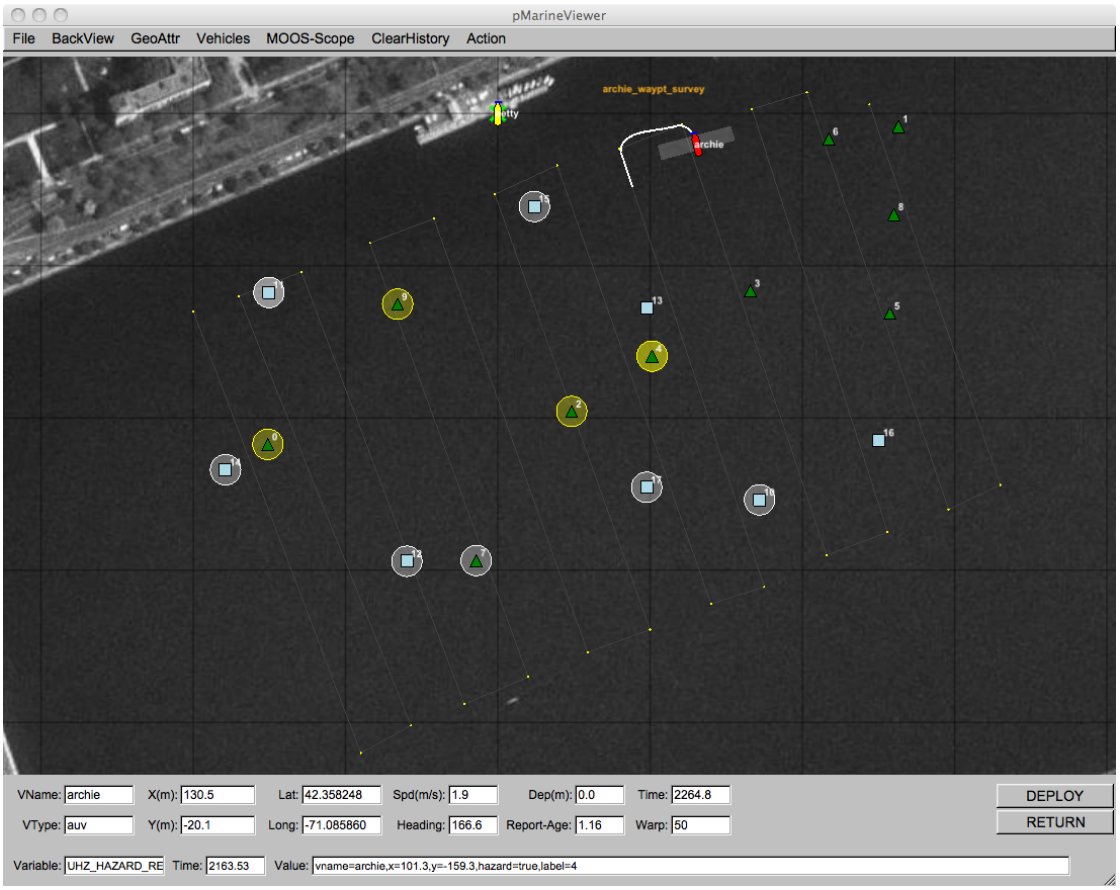


Figure 19: **Simulated Hazard Sensor:** A vehicle processes a series of sensor images which may or may not contain an object. The detection algorithm processes the image and rejects images it believes does not contain contain hazardous objects. It passes on images containing possible hazardous objects to a classifier, which makes a determination for each object in an incoming image as to whether or not the object is hazardous or benign.

In this mission, the vehicle's sensor is configured with a swath width of 25 meters (12.5 on either side), a probability of detection P_D=0.9, probability of false alarm P_{FA}=0.66, and probability of correct classification P_C=0.85. Note in the above snapshot, the vehicle has successfully detected all hazards, but also detected all but one benign object. It has correctly classified all but one object.

The Reacquire Phase

In the *reacquire* phase, the archie vehicle has returned to the dock, and betty vehicle has been a mission to revisit a set of points. In this case betty has an idea where those points lie and is following a simple path, as shown in Figure 20. Presumably the list of objects to visit and their locations have been communicated to betty from archie. (In this mission things were hard-coded, no message passing actually occurred.) The objective of betty is to use a sensor with a smaller swath width and better sensor processing algorithm to reduce the classification uncertainty associated with the objects being visited.

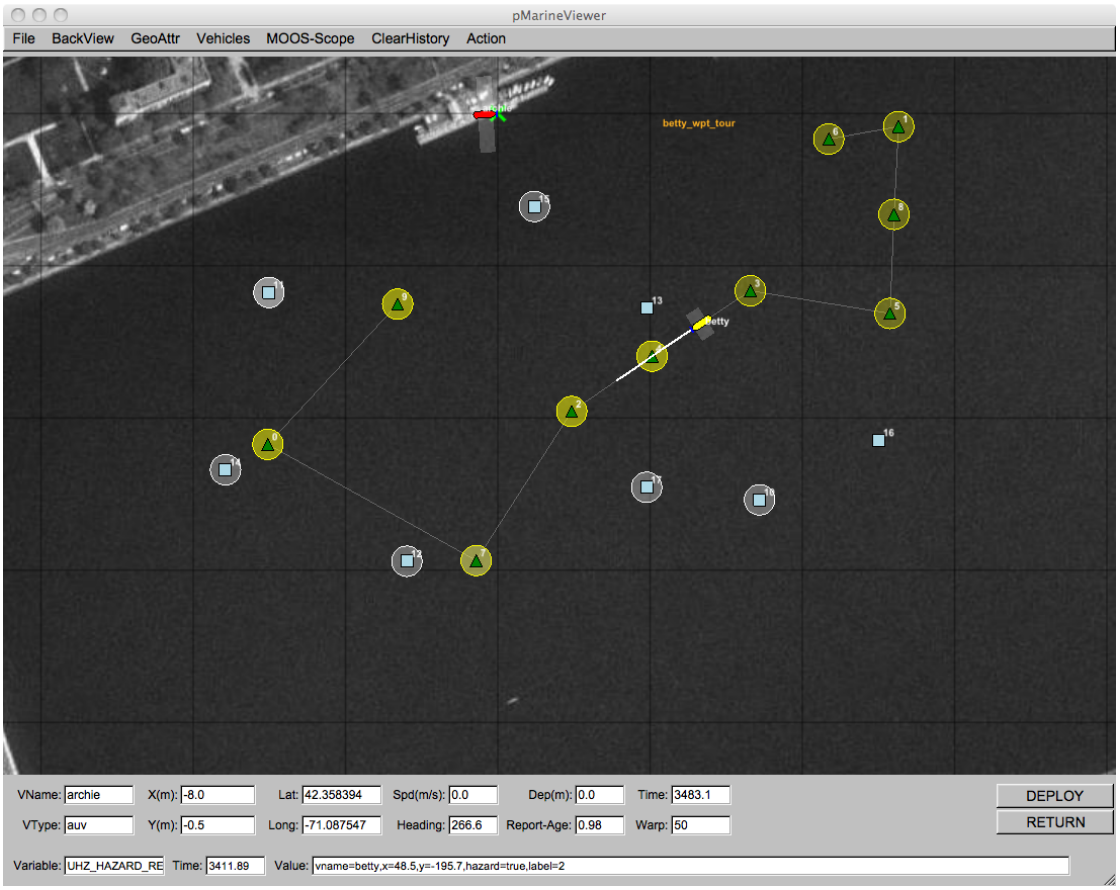


Figure 20: **A Reacquire Mission:** A second vehicle revisits a set of previously detected objects and seeks to reduce the uncertaintly associated with their initial classification made with a less reliable sensor.

In this mission, the vehicle's sensor is configured with a swath width of 10 meters (5 meters on either side), a probability of detection P_D=0.9, probability of false alarm P_{FA}=0.53, and probability of correct classification P_C=0.93.

10 The uFldBeaconRangeSensor Utility: Simulating Vehicle to Beacon Ranges

The `uFldBeaconRangeSensor` application is a tool for simulating an on-board sensor that provides a range measurement to a beacon where either (a) the vehicle knows its own position but is trying to determine the position of the beacon via a series of range measurements, or (b) the vehicle knows where the beacon(s) are but is trying to determine its own position based on the range measurements from one or more beacons at known locations.

The range-only sensor may be one that responds to a query, e.g., an acoustic ping, with an immediate reply, e.g. another acoustic ping or echo, from which the range from the the source to the beacon is determined by the time-of-flight of the message through the medium, e.g., the approxe speed ound through water. This idea is shown below on the left. Alternatively, if the beacon emits its message on a precise schedule with a clock precisely synchronized with the vehicle clock, the range measurement may be derived without requiring a separate query from the vehicle. This is the idea behind long baseline acoustic navigation, [3–6]. This idea is shown below on the right.

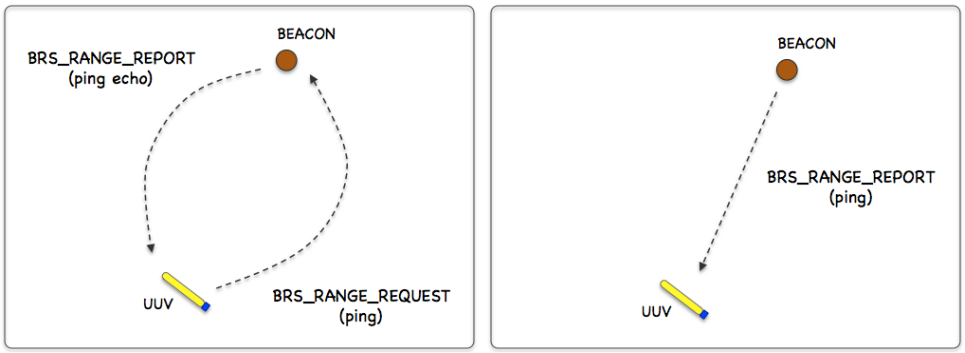


Figure 21: **Beacon Range Sensors:** A vehicle determines its range to a beacon by either (a) emitting a query and waiting for a reply, or (b) waiting for a message to be emitted on fixed schedule. In each case, the time-of-flight of the message through the medium is used to calculate the range.

In the `uFldBeaconRangeSensor` application, the beacon and vehicle locations are known to the simulator, and a tidy `BRS_RANGE_REPORT` message is sent to the vehicle(s) as a proxy to the actual range sensor and calculations that would otherwise reside on the vehicle. The MOOS app may be configured to have beacons provide a range report either (a) solicited with a range request, or (b) unsolicited. One may also configure the range at which a range request will be heard, and the range at which a range report will be heard. The app may be further configured to either (1) include the beacon location and ID, or (2) not include the beacon location or ID.

Typical Simulator Topology

The typical module topology is shown in Figure 22 below. Multiple vehicles may be deployed in the field, each periodically communicating with a shoreside MOOS community running a single instance of `uFldBeaconRangeSensor`. Each vehicle regularly sends a node report noted by the simulator to keep an updated calculation of each vehicle to each simulated beacon. When a beacon wants to

simulate a ping, or range request, it generates the BRS_RANGE_REQUEST message send to the shore. After the simulator calculates the range, a reply message, BRS_RANGE_REPORT is sent to the vehicle.

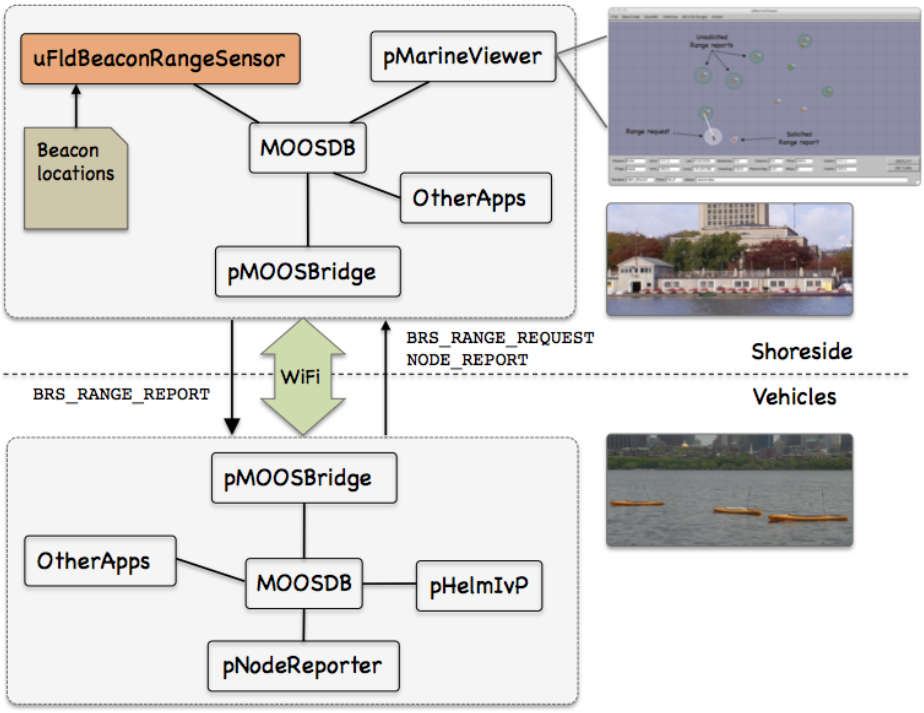


Figure 22: **Typical uFldBeaconRangeSensor Topology:** The simulator runs in a shoreside computer MOOS communicty and is configured with the beacon locations. Vehicles accessing the simulator periodically send node reports to the shoreside community. The simulator maintains a running estimate of the range between vehicles and beacons, modulo latency. A vehicle simulates a ping by sending a range request to shore and receiving a range report in return from the simulator.

If running a pure simulation (no deployed vehicles), both MOOS communities may simply be running on the same machine configured with distinct ports. The pMOOSBridge application is shown here for communication between MOOS communities, but there are other alternatives for inter-community communication and the operation of uFldBeaconRangeSensor is not dependent on the manner of inter-communication communications.

10.1 The uFldBeaconRangeSensor Interface and Configuration Options

The uFldBeaconRangeSensor application may be configured with a configuration block within a .moos file. Its interface is defined by its publications and subscriptions for MOOS variables consumed and generated by other MOOS applications. An overview of the set of configuration options and interface is provided in this section.

10.1.1 Configuration Parameters of uFldBeaconRangeSensor

The following parameters are defined for uFldBeaconRangeSensor. A more detailed description is provided in other parts of this section. Parameters having default values indicate so in parentheses

below.

BEACON:	Description of beacon location and properties.
DEFAULT_BEACON_FREQ:	Frequency of unsolicited beacon broadcasts ("never").
DEFAULT_BEACON_REPORT_RANGE:	Range at which a vehicle will hear a range report (100).
DEFAULT_BEACON_WIDTH:	Width of beacons (meters) when rendered (4).
DEFAULT_BEACON_COLOR:	Color of beacons when rendered ("red").
DEFAULT_BEACON_SHAPE:	Shape of beacons when rendered ("circle").
PING_PAYMENTS:	How pings treated w.r.t. ping wait time ("upon_response").
GROUND_TRUTH:	If true, ground truth is also reported when noise is added.
PING_WAIT:	Mandatory number of seconds between successive vehicle pings.
REACH_DISTANCE:	Range at which a vehicle ping will be heard (100).
REPORT_VARS:	Determines variable name(s) used for range report ("short").
RN_ALGORITHM:	Algorithm for adding random noise to range measurements.
VERBOSE:	If true, verbose status message terminal output (false).

10.1.2 MOOS Variables Published by uFldBeaconRangeSensor

The primary output of uFldBeaconRangeSensor to the MOOSDB is posting of range reports, visual cues for the range reports, and visual cues for the beacons themselves.

BRS_RANGE_REPORT:	A report on the range from a particular beacon to a particular vehicle.
BRS_RANGE_REPORT_NAMEJ:	A report on the range from a particular beacon to vehicle NAMEJ.
VIEW_MARKER:	A description for visualizing the beacon in the field. (Section 10.3)
VIEW_RANGE_PULSE:	A description for visualizing the beacon range report. (Section 10.3)

The range report format may vary depending on user configuration. Some examples:

```
BRS_RANGE_REPORT = "name=alpha,range=129.2,time=19473362764.169"
BRS_RANGE_REPORT = "name=alpha,range=129.2,id=23,x=54,y=90,time=19473362987.428"
BRS_RANGE_REPORT_ALPHA = "range=129.2,time=19473362999.761"
```

The vehicle name may be embedded in the MOOS variable name to facilitate distribution of report messages to the appropriate vehicle with pMOOSBridge.

10.1.3 MOOS Variables Subscribed for by uFldBeaconRangeSensor

The uFldBeaconRangeSensor application will subscribe for the following four MOOS variables:

BRS_RANGE_REQUEST:	A request to generate range reports for all beacons to all vehicles within range of the beacon.
NODE_REPORT:	A report on a vehicle location and status.
NODE_REPORT_LOCAL:	A report on a vehicle location and status.

10.1.4 Command Line Usage of uFldBeaconRangeSensor

The `uFldBeaconRangeSensor` application is typically launched as a part of a batch of processes by pAntler, but may also be launched from the command line by the user. The command line options may be shown by typing `"uFldBeaconRangeSensor --help"`:

Listing 22 - Command line usage for the uFldBeaconRangeSensor tool.

```
Usage: uFldBeaconRangeSensor file.moos [OPTIONS]

Options:
  --alias=<ProcessName>
      Launch uFldBeaconRangeSensor with the given process
      name rather than uFldBeaconRangeSensor.
  --example, -e
      Display example MOOS configuration block
  --help, -h
      Display this help message.
  --verbose=Boolean (true/false)
      Display diagnostics messages. Default is true.
  --version,-v
      Display the release version of uFldBeaconRangeSensor.
```

10.1.5 An Example MOOS Configuration Block

As of MOOS-IvP Release 4.2, most if not all MOOS apps are implemented to support the `-e` or `--example` command-line switches. To see an example MOOS configuration block, enter the following from the command-line:

```
$ uFldBeaconRangeSensor -e
```

This will show the output shown in Listing 23 below.

Listing 23 - Example configuration of the uFldBeaconRangeSensor application.

```
============================================================
uFldBeaconRangeSensor Example MOOS Configuration
============================================================
Blue lines:     Default configuration
Magenta lines:  Non-default configuration

{
  AppTick   = 4
  CommsTick = 4

  // Configuring aspects of vehicles in the sim
  reach_distance = default = 200   // or {nolimit}
  reach_distance = henry = 40      // meters
  ping_wait      = default = 30    // seconds
  ping_wait      = henry   = 120
  ping_payments  = upon_response   // or {upon_receipt, upon_request}

  // Configuring manner of reporting
  report_vars    = short  // or {long, both}
```

```
19    ground_truth   = true   // or {false}
20    verbose        = true   // or {false}
21
22    // Configuring default beacon properties
23    default_beacon_shape = circle    // or {square, diamond, etc.}
24    default beacon_color = orange    // or {red, green, etc.}
25    default_beacon_width = 4
26    default_beacon_report_range = 100
27    default_beacon_freq  = never     // or [0,inf]
28
29    // Configuring Beacon properties
30    beacon = x=200, y=435, label=01, report_range=45
31    beacon = x=690, y=205, label=02, freq=90
32    beacon = x=350, y=705, label=03, width=8, color=blue
33
34    // Configuring Artificial Noise
35    rn_algorithm  =  uniform,pct=0   // pct may be in [0,1]
36  }
```

10.2 Using and Configuring the uFldBeaconRangeSensor Utility

The `uFldBeaconRangeSensor` application is configured primarily with a set of beacons, and a policy for generating range reports to one or more simulated vehicles. The reports may be sent to the vehicles upon a query (solicited) or may be sent unsolicited based on a configured broadcast schedule for each beacon. The possible simulator configuration arrangements are explored by first considering a simple case shown in Figure 23 below, representing a vehicle navigating with three beacons.

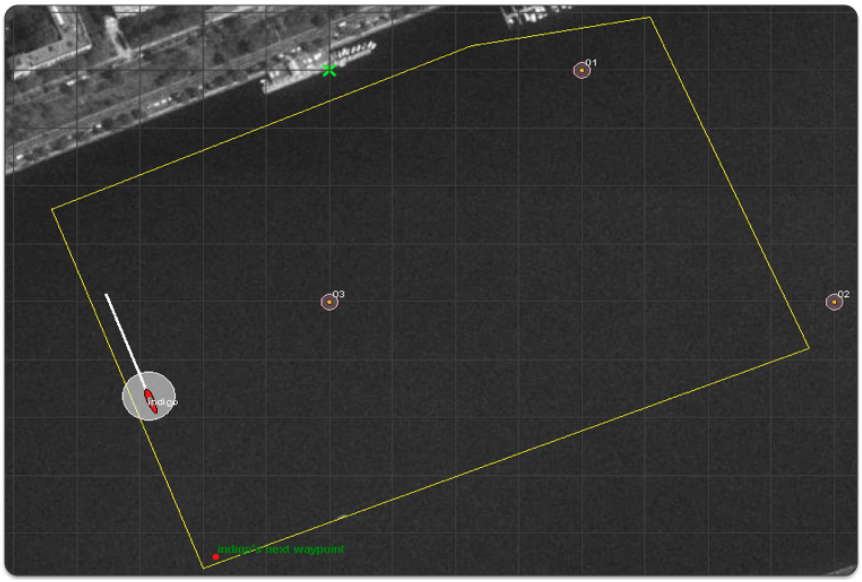


Figure 23: **Simulated LBL Beacons:** Three beacons are simulated, labelled 01, 02, and 03. The vehicle periodically issues a query to which the beacons immediately reply. The `uFldBeaconRangeSensor` application handles the queries and generates the range reports sent to each vehicle. The growing circles rendered around the vehicle and beacons represent the generation of the range query and range reports respectively.

The configuration for the `uFldBeaconRangeSensor` is shown in Listing 24 below. The three beacons are configured in lines 19-21. The configuration on line 7 indicates that a beacon query will be heard regardless of the range between the vehicle and the beacon. In the other direction, the range report from the beacon will only be heard if the vehicle is within 100 meters. Line 8 indicates that ping or range request will be honored by the simulator at most once every 30 seconds, and this clock is reset each time a range request is honored by the simulator with the configuration on line 9.

Listing 24 - Example configuration of the uFldBeaconRangeSensor application.

```
 1  ProcessConfig = uFldBeaconRangeSensor
 2  {
 3    AppTick   = 4   // Standard MOOSApp configurations
 4    CommsTick = 4
 5
 6    // Configuring aspects of the vehicles
 7    reach_distance = default = nolimit
 8    ping_wait      = 30
 9    ping_payments  = upon_accept
10
11    report_vars    = short
12
13    default_beacon_freq  = never       // Only on request (ping)
14    default_beacon_shape = circle
15    default_beacon_color = orange
16    default_beacon_width = 5
17    default_beacon_report_range = 100
18
19    beacon  = label=01, x=200, y=0
20    beacon  = label=02, x=400, y=-200
21    beacon  = label=03, x=0,   y=-200, color=red, shape=triangle, report_range=80
22  }
```

The three beacons in this example are configured on lines 19-21 with unique labels and locations. Each beacon has additional properties, such as its shape, color and width when rendered. Default values for these properties are given in lines 14-16, but may be overridded for a particular beacon as on line 22.

The key configuration line in this example is on line 13 which indicates the beacons by default never generate an unsolicited range report. Reports are only generated upon request. In this example, the simulated vehicle would receive successive groups of `brs_RANGE_REPORT` postings from all three simulated beacons, each time the vehicle posts a `BRS_RANGE_REQUEST` message to the MOOSDB. This example is runnable in the *indigo* example mission distributed with the MOOS-IvP source code and described a bit later in Section 10.4.

10.2.1 Configuring the Beacon Locations and Properties

One or more beacons may be configured by the `BEACON` configuration parameter provided in the `uFldBeaconRangeSensor` configuration block of the `.moos` file. Each beacon is configured with a line:

```
BEACON = <configuration>
```

The <configuration> component is a comma-separated list of parameter=value pairs, with the following possible parameters: x, y, label, freq, shape, width, color, and query_range. The following are typical examples:

```
beacon  = x=200, y=260, label=03, freq=10
beacon  = x=-40, y=150, label=04, freq=5:15, color=red, shape=circle, width=4, report_range=200
```

The x and y parameters specify the beacon locations in local coordinates. Like several other MOOS applications, the uFldBeaconRangeSensor app looks for a global parameter in the .moos configuration file naming the position of the datum, or 0,0 position in latitude, longitude coordinates. The label parameter provides a unique identifier for the beacon. If a beacon entry is provided using a previously used label, the new beacon will overwrite the prior beacon in the simulator. If no label is provided, an automatic label will be generated equivalent to the index of the new beacon. The freq parameter specifies, in seconds, how often *unsolicited* range reports are generated for each beacon. A simple numerical value may be given, or a colon-separated pair of values as shown above may be used to specify a uniformly random interval of possible durations. The duration between posts will be reset after each post. The report_range parameter specifies the distance, in meters, that a vehicle must be to hear a range report generated by a beacon. The shape parameter indicates the shape used by applications like pMarineViewer when rendering the beacon. The uFldBeaconRangeSensor application generates a VIEW_MARKER post to the MOOSDB for each beacon, once upon startup of the simulator. The VIEW_MARKER structure and possible shapes are described in the pMarineViewer section in [1]. The color parameter specifies the color to be used when rendering the beacon. Legal color strings are described in Appendix A. The width parameter is used to indicate the width, in meters, when rendering the beacon.

For convenience, default values for several of the above properties may be provided with the following five supported configuration parameters:

```
default_beacon_report_range = 100
default_beacon_shape  = circle
default_beacon_color  = orange
default_beacon_width  = 4
default_beacon_freq   = never
```

The above configuration values also represent all the default values. A beacon configuration that includes any of the above five parameters explicitly, will override any default values.

10.2.2 Unsolicited Beacon Range Reports

The simulator may be configured to have all its beacons periodically generate a range report to all vehicles within range. The schedule of reporting may be uniform across all beacons, or individually set for each beacon. The interval of time between reports may also be set to vary according to a uniformly random time interval. By default, beacons are configured to never generate unsolicited range reports unless their frequency parameter is set to something else besides the default value of "never". The default value for all beacons may be configured with the following parameter in the uFldBeaconRangeSensor configuration block with the following:

```
default_beacon_freq = 120
```

The parameter value is given in seconds. To configure an interval to vary randomly on each post within a given range, e.g., somewhere between one and two minutes, the following may be used instead:

```
default_beacon_freq = 60:120
```

To configure the simulator to never generate an unsolicited range report, i.e., only solicited reports, use the following:

```
default_beacon_freq = never
```

Upon each range report post to the MOOSDB, the interval until the next post is recalculated. The beacon schedule may also be configured to be unique to a given beacon. The beacon configuration line accepts the `freq` parameter as described earlier in Section 10.2.1. The configuration provided for an individual beacon overrides the default frequency configuration.

Once a beacon has generated a report, it will not generate another *unsolicited* report until after the prevailing time interval has passed. However, if the simulator detects that the beacon has been solicited for a range report via an explicit range request from a nearby vehicle, a range report may be generated immediately. In this case the clock counting down to the beacon's next unsolicited report is reset.

10.2.3 Solicited Beacon Range Reports

The `uFldBeaconRangeSensor` application accepts requests from vehicles, and may or may not generate one or more range reports for beacons within range of the vehicle making the request. In short, things operate like this: (a) a range request is received by `uFldBeaconRangeSensor` through its mailbox on the variable `RANGE_REQUEST`, (b) a determination is made as to whether the request is within range of the beacon and whether the request is allowed based on limits on the frequency of range requests, (c) a range report is generated and posted to the variable `BRS_RANGE_REPORT`. The following is an example of the range request format:

```
BRS_RANGE_REQUEST = "name=charlie"
```

Note that if a vehicle generates a range request triggering a range report from a beacon, the range report is sent to *all* vehicles within range of the beacon. Presumably the simulator has also received, at some point in the past, a node report, typically generated from the `pNodeReporter` application (described in the `pNodeReporter` section in [1]) running on the vehicle. So the simulator not only knows which vehicle is making the range request, but also where that vehicle is located. It needs the vehicle location to determine the range between the vehicle and beacon, to generate the requested range report. The simulator also uses this range information to decide if it wants regard the beacon as being close enough to hear the request, and whether the vehicle is close enough to the beacon to hear the report.

10.2.4 Limiting the Frequency of Vehicle Range Requests

From the perspective of operating a vehicle, one may ask: why not request a range report from all beacons as often as possible? There may be reasons why this is not feasible outside simulation.

Limits may exist due to power budgets of the vehicle and/or beacons, and there may be prevailing communications protocols that make it at least impolite to be pushing range requests through a shared communications medium.

To reflect this limitation, the `uFldBeaconRangeSensor` application may be configured to limit the frequency in which a vehicle's range request (or ping) will be honored with a range report reply. By default this frequency is set to once every 30 seconds for all vehicles. The default for all vehicles may be changed with the following configuration in the `.moos` file:

```
PING_WAIT = default = 60
```

If the time interval as above is set to 60 seconds, what happens if a vehicle requests a range report 40 seconds after its previous request? Is it simply ignored, needing to wait another 20 seconds? Or is the clock reset to zero forcing the vehicle to wait 60 seconds before a ping is honored? By default, the former is the case, but the simulator may be configured to the more draconian option with:

```
PING_PAYMENTS = upon_request
```

Suppose the minimum time interval has elapsed, but the querying/pinging vehicle is too far out of range from any beacon to hear even a single range report. Will the result be that the clock is reset to zero, forcing the vehicle to wait another 60 seconds before a query is honored? By default this is the case, but the simulator may be configured to not reset the clock unless the querying vehicle has received at least one range report for its query:

```
PING_PAYMENTS = upon_response
```

In short, the simulator configuration parameter, `PING_PAYMENTS`, may be configured with one of three options, `"upon_request"`, `"upon_response"`, or `"upon_accept"`, with the default being the latter.

10.2.5 Producing Range Measurements with Noise

In the default configuration of `uFldBeaconRangeSensor`, range reports are generated with the most precise range estimate as possible, with the only error being due to the latency of the communications generating the range request and range report. Additional noise/error may be added in the simulator for each range report with the following configuration parameter:

```
rn_algorithm   = uniform,pct=0.12     // Values in the range [0,1]
```

Currently the only noise algorithm supported is the generation of uniformly random noise on the range measurement. The noise level, θ, set with the parameter `rn_uniform_pct`, will generate a noisy range from an otherwise exact range measurement r, by choosing a value in the range $[\theta r, r + \theta r]$. The range without noise, i.e., the ground truth, may also be reported by the simulator if desired by setting the configuration parameter:

```
ground_truth = true
```

This will result in an additional MOOS variables posted, `BRS_RANGE_REPORT_GT`, with the same format as `BRS_RANGE_REPORT`, except the reported range will be given without noise.

10.2.6 Console Output Generated by uFldBeaconRangeSensor

Information regarding the startup of the `uFldBeaconRangeSensor` application may be monitored from an open console window where `uFldBeaconRangeSensor` is launched. If the verbose setting is turned on, further output is generated as the simulator progresses and receives range requests and generates range reports. The verbose setting may be turned on from the command line, `--verbose`, or in the mission file configuration block with `verbose=true`. Example output is shown below in Listing 25. Certain startup information common across all MOOS applications can be found in lines 1-14, and lines 33-36 in the example below. The block of output in lines 16-31 provides startup feedback unique to `uFldBeaconRangeSensor`. This cannot be turned off with the verbose setting, and should appear in blue in the console window. The Figlog summary in lines 17-22 provides feedback on the the configuration parameters provided in the `uFldBeaconRangeSensor` block of the mission file. Following this, e.g. in lines 23-30, a summary of the simulator model is given, showing the configured beacons, rendering hints, and other simulator policies.

Listing 25 - Example uFldBeaconRangeSensor console output.

```
 1  ************************************************
 2  *                                              *
 3  *       This is MOOS Client                    *
 4  *       c. P Newman 2001                       *
 5  *                                              *
 6  ************************************************
 7
 8  ---------------MOOS CONNECT----------------------
 9    contacting a MOOS server localhost:9000 -  try 00001
10    Contact Made
11    Handshaking as "uFldBeaconRangeSensor"
12    Handshaking Complete
13    Invoking User OnConnect() callback...ok
14  ------------------------------------------------
15
16  Simulated Range Sensor starting...
17  ================================================
18  Figlog Summary:
19  Messages: (0)
20  Warnings: (0)
21  Errors: (0)
22  ================================================
23  SRS Model - # Beacons: 3
24    [0]:x=0,y=-200,label=03,color=orange,type=circle,width=4
25    [1]:x=400,y=-200,label=02,color=orange,type=circle,width=4
26    [2]:x=200,y=0,label=01,color=orange,type=circle,width=4
27  Default Beacon Color: orange
28  Default Beacon Shape: circle
29  Default Beacon Width: 4
30   NodeRecords:: 0
31  Simulated Range Sensor started.
32
33  uFldBeaconRangeSensor is Running:
34          AppTick   @ 4.0 Hz
35          CommsTick @ 4 Hz
36          Time Warp @ 6.0
```

```
37
38 Received first NODE_REPORT for: indigo
39 ***************************
40 Range request received from: indigo
41    Elapsed time: 167.24879
42    Query accepted by uFldBeaconRangeSensor.
43    Range report sent from beacon[03] to vehicle[indigo]
44    Range report sent from beacon[02] to vehicle[indigo]
45    Range report sent from beacon[01] to vehicle[indigo]
46 ***********************
47 Unsolicited beacon reports:
48    Range report sent from beacon[03] to vehicle[indigo]
49    Range report sent from beacon[01] to vehicle[indigo]
50 ****************************************
```

Unless the verbose setting is turned on, the output ending on line 37 above should be the last output written to the console for the duration of the simuator.

In the verbose mode, the simulator will produce event-based output as shown in the example above beginning on line 38. The asterisks in lines 39, 46, and 50 are not merely visual separators. An asterisk represents a single receipt of a `NODE_REPORT` message. Receiving node reports is essential for the operation of the simulator and this provides a bit of visual verification that this is occurring. Presumably node reports are being received much more often than range requests and range reports are handled, as is the case in the above example. The first time a node report is received for a particular vehicle, an announcement is made as shown on line 38.

In addition to handling incoming node reports, on any given iteration, the simulator may also handle an incoming range request, or may generate an unsolicited range report based on a beacon schedule. Console output for incoming range requests may look like that shown in lines 40-45 above. First the range request and the requesting vehicle is announced as online 40. The elapsed time since the vehicle last made a range request is shown as on line 41. If the request is honored by the simulator, this is indicated as shown on line 42. Otherwise a reason for denial may be shown. If the query is accepted, range reports may be generated for one or more vehicles. For each such vehicle, a line announcing the new report is generated, as in lines 43-45. On an iteration where unsolicited range reports are generated, output similar to that shown in lines 47-49 will generated. For each report, the beacon and receiving vehicle are named.

10.3 Interaction between uFldBeaconRangeSensor and pMarineViewer

The `uFldBeaconRangeSensor` application will post certain messages to the MOOSDB that may be subscribed for by GUI based applications like `pMarineViewer` for visualizing the posting of `BRS_RANGE_REPORT` and `BRS_RANGE_REQUEST` messages, as well as visualizing the beacon locations. A snapshot of the `pMarineViewer` window is shown below, with one vehicle and several beacons.

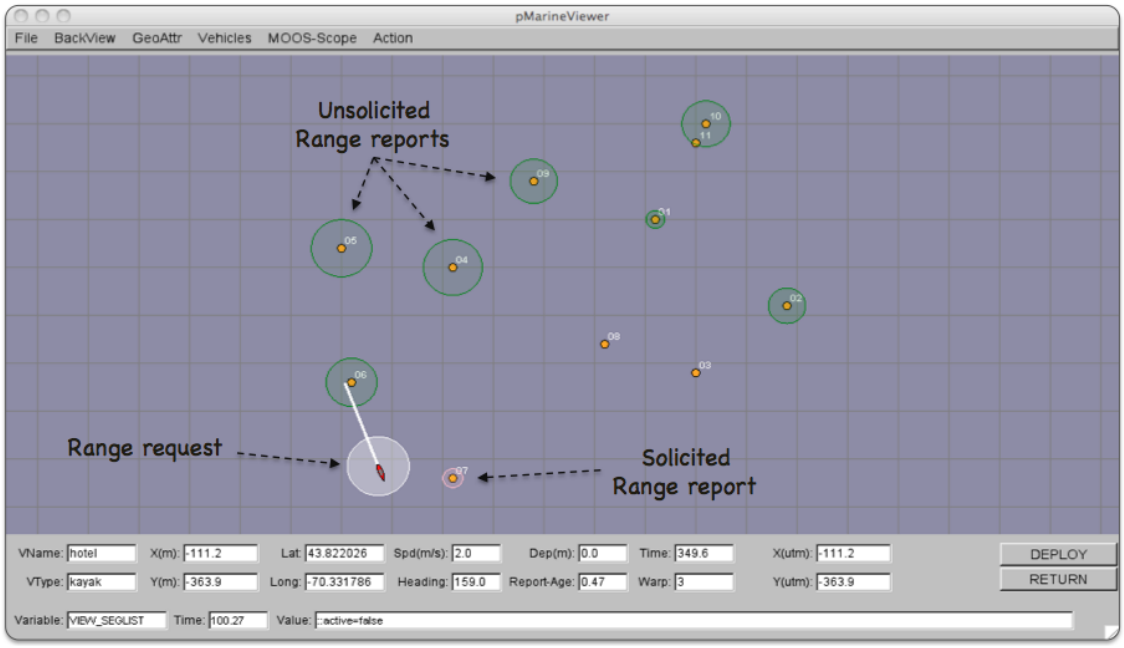


Figure 24: **Beacons in the pMarineViewer:** The `VIEW_RANGE_PULSE` message is passed to pMarineViewer to render unsolicited range reports (here in green), range requests from a vehicle (here in white), and solicited range reports in response to a range requeste (here in pink). The viewer alse renders the beacons and their labels upon receiving `VIEW_MARKER` messages posted by the uFldBeaconRangeSensor application. The pulses are only momemtarily visible until another `VIEW_RANGE_PULSE` message is received.

The `VIEW_MARKER` Data Structure

The `uFldBeaconRangeSensor` application, upon startup, posts the beacon locations in the form of the `VIEW_MARKER` data structure. This MOOS variable is one of the default variables registered for by the `pMarineViewer` application. The types of supported markers are described in the `pMarineViewer` section in [1]. The marker type, size and color are configurable in the `uFldBeaconRangeSensor` configuration block. The user may use the variation in marker rendering to correspond to variation in beacon reporting or querying characteristics.

The `RANGE_PULSE` Data Structure

A *range pulse* message is used by the `uFldBeaconRangeSensor` application to convey visually the generation of a range report, or the receipt of a range request. The pulse is rendered as a ring with a growing radius having either the beacon or the vehicle at the center. A pulse eminating from a

beacon indicates a range report, and a pulse eminating from a vehicle indicates a range request. By default different colors may be used for solicited and unsolicited range reports. In Figure 24 for example, the green rings represent unsolicited reports, the white ring represents a range request made by the vehicle, and the pink ring represents a response to the range request made by the one beacon within range to the vehicle.

The `RANGE_PULSE` message is a data structure implemented in the `XYRangePulse` class, and usually passed through the MOOS variable `VIEW_RANGE_PULSE` with the following format:

```
VIEW_RANGE_PULSE = <pulse>
```

The `<pulse>` component is a series of comma-separated `parameter=value` pairs. The supported parameters are: `x`, `y`, `radius`, `duration`, `time`, `fill`, `fill_color`, `label`, `edge_color`, and `edge_size`.

The `x` and `y` parameters convey the center of the pulse. The `radius` parameter indicates the radius of the circle at its maximum. The `duration` parameter is the number of seconds the pulse will be rendered. The pulse will grow its radius linearly from zero meters at zero seconds to `radius` meters at `duration` seconds. The `fill` parameter is in the range $[0, 1]$, where 0 is full transparency (clear) and 1 is fully opaque. The pulse transparency increases linearly as the range ring is rendered. The starting transparency at $radius = 0$ is given by the `fill` parameter. The transparency at the maximum radius is zero. The `fill_color` parameter specifies the color rendered to the internal part of the range pulse. The choice of legal colors is described in Appendix A. The `label` is a string that uniquely identifies the range instance to consumers like `pMarineViewer`. As with other objects in `pMarineViewer`, if it receives an object the same label and type as one previously received, it will replace the old object with the new one in its memory. The `edge_color` parameter describes the color of the ring rendered in the range pulse. The `edge_size` likewise describes the line width of the rendered ring. The `time` parameter indicates the UTC time at which the range pulse object was generated.

Below is an example string representing a range pulse. Each field, with the exception of the x, y position, also indicates the default values if left unspecified:

```
VIEW_RANGE_PULSE = x=-40,y=-150,radius=40,duration=15,fill=0.25,fill_color=green,label=04
                   edge_color=green,time=3892830128.5,edge_size=1
```

Exercise 10.1: Poking a RangePulse for visualizing in pMarineViewer.

- Try running the Alpha mission described in [2], using the `uPokeDB` tool.
- Poke the MOOSDB with:

  ```
  $ uPokeDB alpha.moos VIEW_RANGE_PULSE="x=100,y=-50,radius=40,duration=15,fill=0.25,
  fill_color=green,label=04,time=@MOOSTIME"
  ```

  A range pulse should appear in the viewer 100 meters East and 50 meters South of the vechicle's starting position. Note the special macro `@MOOSTIME`, which uPokeDB will expand to the UTC time at which the poke was made.

10.4 The Indigo Example Mission Using uFldBeaconRangeSensor

The *indigo* mission is distributed with the MOOS-IvP source code and contains a ready example of the `uFldBeaconRangeSensor` application, configured with three beacons acting as long baseline (LBL) beacons as described at the beginning of Section 10.2. Assuming the reader has downloaded the source code available at `www.moos-ivp.org` and built the code according to the discussion in Section 1.5, the *indigo* mission may be launched by:

```
$ cd moos-ivp/ivp/missions/s9_indigo/
$ ./launch.sh 10
```

The argument, 10, in the line above will launch the simulation in `10x` real time. Once this launches, the `pMarineViewer` GUI application should launch and the mission may be initiated by hitting the `DEPLOY` button. The vehicle will traverse a survey pattern over the rectangular operation region shown in Figure 23, periodically generating a range request to the three beacons. With each range request, a white range pulse should be visible around the vehicle. Almost immediately afterwards, a range report for each beacon is generated and a red range pulse around each beacon is rendered. The snapshot in Figure 23 depicts a moment in time where the range report visual pulses are beginning to grow around the beacons and the range request visual pulse is still visible around the one vehicle.

How does the simulated vehicle generate a range request? In practice a user may implement an intelligent module to reason about when to generate requests, but in this case the `uTimerScript` application is used by creating a script that repeats endlessly, generating a range request once every 25-35 seconds. The script is also conditioned on `(NAV_SPEED > 0)`, so the pinging doesn't start until the vehicle is deployed. The configuration for the script can be seen in the `uTimerScript` configuration block in the `indigo.moos` file. More on the `uTimerScript` application can be found in the `uTimerScript` section in [1].

Examining the Log Data from the Indigo Mission

After the launch script above has launched the simulation, the script should leave the console user with the option to "Exit and Kill Simulation" by hitting the '2' key. Once the vehicle has been deployed and traversed to one's satisfaction, exit the script. A log directory should have been created by the `pLogger` application in the directory where the simulation was launched. The directory name should be begin with `MOOSLog_` with the remainder of the directory name composed from the timestamp of the launch.

Let's take a look at some of the data related to the simulation and `uFldBeaconRangeSensor` in particular. A dump of the entire file reveals a deluge of information. To look at the information relevant to the `uFldBeaconRangeSensor` application, the file is pruned with the `aloggrep` tool:

```
$ aloggrep MOOSLog_21_2_2011_____22_32_48.alog uFldBeaconRangeSensor uTimerScript
```

This produces a subset of the alog file similar to that shown in Listing 26, showing only log entries made by either the `uFldBeaconRangeSensor` application, or the `uTimerScript` application which generated all the range requests as described above. The first three posts made to the

MOOSDB by uFldBeaconRangeSensor are the VIEW_MARKER posts representing a visual cue for the pMarineViewer application to render the three beacons.

Listing 26 - A subset of the data logged from the Indigo example mission's alog file.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% LOG FILE:       ./MOOSLog_22_2_2011_____17_27_06/MOOSLog_22_2_2011_____17_27_06.alog
%% FILE OPENED ON  Tue Feb 22 17:27:06 2011
%% LOGSTART               23371445284.8
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
55.697      VIEW_MARKER          uFldBeaconRangeSensor  x=0,y=-200,label=03,color=orange,type=circle,width=4
55.698      VIEW_MARKER          uFldBeaconRangeSensor  x=400,y=-200,label=02,color=orange,type=circle,width=4
55.698      VIEW_MARKER          uFldBeaconRangeSensor  x=200,y=0,label=01,color=orange,type=circle,width=4
100.663     BRS_RANGE_REQUEST    uTimerScript           name=indigo
100.846     VIEW_RANGE_PULSE     uFldBeaconRangeSensor  x=-97.65,y=-64.84,radius=50,duration=6,...
100.846     BRS_RANGE_REPORT     uFldBeaconRangeSensor  vname=indigo,range=166.7446,id=03,time=23371445385.6
100.846     VIEW_RANGE_PULSE     uFldBeaconRangeSensor  x=0,y=-200,radius=40,duration=15,...
100.846     BRS_RANGE_REPORT     uFldBeaconRangeSensor  vname=indigo,range=515.6779,id=02,time=23371445385.6
100.846     VIEW_RANGE_PULSE     uFldBeaconRangeSensor  x=400,y=-200,radius=40,duration=15,...
100.847     BRS_RANGE_REPORT     uFldBeaconRangeSensor  vname=indigo,range=304.6305,id=01,time=23371445385.6
100.847     VIEW_RANGE_PULSE     uFldBeaconRangeSensor  x=200,y=0,radius=40,duration=15,...
160.419     BRS_RANGE_REQUEST    uTimerScript           name=indigo
160.597     VIEW_RANGE_PULSE     uFldBeaconRangeSensor  x=-197.96,y=-129.16,radius=50,duration=6,...
160.597     BRS_RANGE_REPORT     uFldBeaconRangeSensor  vname=indigo,range=210.2533,id=03,time=23371445445.3
160.597     VIEW_RANGE_PULSE     uFldBeaconRangeSensor  x=0,y=-200,radius=40,duration=15,...
160.597     BRS_RANGE_REPORT     uFldBeaconRangeSensor  vname=indigo,range=602.1416,id=02,time=23371445445.3
160.597     VIEW_RANGE_PULSE     uFldBeaconRangeSensor  x=400,y=-200,radius=40,duration=15,...
160.597     BRS_RANGE_REPORT     uFldBeaconRangeSensor  vname=indigo,range=418.3951,id=01,time=23371445445.3
160.597     VIEW_RANGE_PULSE     uFldBeaconRangeSensor  x=200,y=0,radius=40,duration=15,...
```

The first range request is generated at time 100.663 by the uTimerScript. The uFldBeaconRangeSensor application receives this mail and posts a range pulse at time 100.846 conveying the range request from the vehicle, e.g., the white circle in Figure 23. The range request is met immediately and two posts are generated for each beacon. The VIEW_RANGE_PULSE message indicates the simulator has generated a range report for the beacon (the red circle in Figure 23. The BRS_RANGE_REPORT message is the actual range report to be used by the vehicle as it sees fit.

10.4.1 Generating Range Report Data for Matlab

The log files generated as in Listing 26 above may processed to form a table of values suitable for Matlab processing. The alog2rng tool may be run on an alog file from the command line:

```
$ alog2rng MOOSLog_21_2_2011_____22_32_48.alog
```

This will generate a table of data like that below. The left column is the timestamp from the log file. The next N columns are the range measurements from each beacon. And the last three columns are the "ground truth" vehicle position and heading. The last three columns may be excluded with the --nav=false switch on the command line.

```
Time       03          02          01          NAV_X      NAV_Y      NAV_HDG
100.846    166.7446    515.6779    304.6305    -99.371    -65.917    238.000
160.597    210.2533    602.1416    418.3951    -198.538   -130.397   197.456
224.844    163.4205    558.5143    433.6937    -155.744   -248.991   159.000
```

The alog2rng tool is part of the Alog-Toolbox along with other tools for examining and modifying alog files generated by pLogger.

11 The uFldContactRangeSensor Utility: Detecting Contact Ranges

The `uFldContactRangeSensor` application is a tool for simulating range measurements to off-board contacts, as a proxy for an on-board active sonar sensor. The range-only measurements are provided conditionally, depending on the range between the pinging vehicle and the contact. The simulator may optionally be configured to provide range measurements with noise.

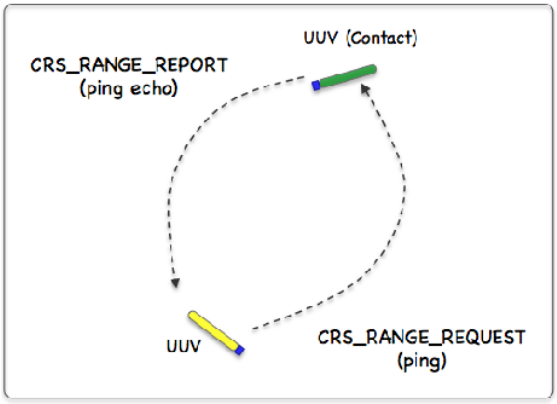


Figure 25: **Simulated Active Sonar:** A vehicle determines its range to another vehicle by producing a simulated sonar ping (a range request to the simulator), and the simulator conditionally responds to the querying vehicle with a report containing the range to nearby vehicles. All vehicles send frequent and regular node reports to the simulator so the simulator can report the range between any two vehicles at any time. The simulator may or may not reply to the range request depending on the range between the two vehicles and thresholds configured by the user.

In the `uFldContactRangeSensor` application, the beacon and vehicle locations are known to the simulator, and a tidy `RANGE_REPORT` message is sent to the vehicle(s) as a proxy to the actual range sensor and calculations that would otherwise reside on the vehicle. The MOOS app may be configured to have beacons provide a range report either (a) solicited with a range request, or (b) unsolicited. One may also configure the range at which a range request will be heard, and the range at which a range report will be heard. The app may be further configured to either (1) include the beacon location and ID, or (2) not include the beacon location or ID.

Typical Topology

The typical module topology is shown in Figure 26 below. Multiple vehicles may be deployed in the field, each periodically communicating with a shoreside MOOS community running a single instance of `uFldContactRangeSensor`. Each vehicle regularly sends a node report noted by the simulator to keep an updated calculation of each vehicle to each simulated beacon. When a vehicle wants to simulate a ping, or range request, it generates the `CRS_RANGE_REQUEST` message sent to the shore. After the simulator calculates the range, a reply message, `CRS_RANGE_REPORT` message is sent to the vehicle, using `pMOOSBridge` or similar app.

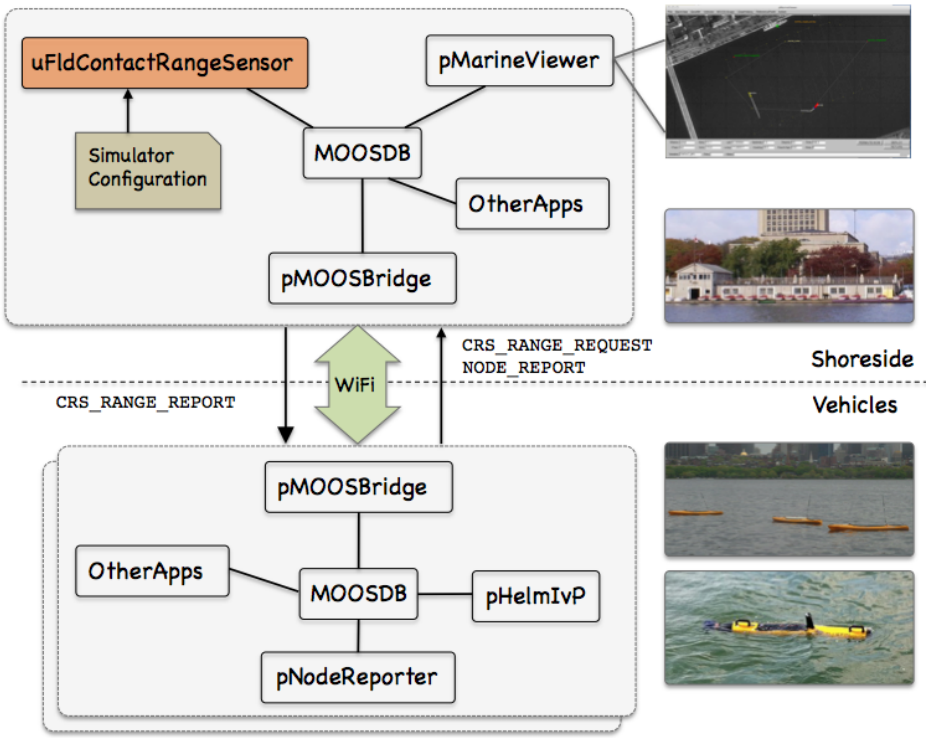


Figure 26: **Typical uFldContactRangeSensor Topology:** The simulator runs in a shoreside computer MOOS community. All deployed vehicles periodically send node reports to the shoreside community. The simulator maintains a running estimate of the range between vehicles, modulo latency. A vehicle simulates a ping by sending a range request to shore and receiving a range report in return from the simulator. The simulator also posts visual artifacts (`VIEW_RANGE_PULSE` messages) read by the pMarineViewer app optionally running shoreside.

If running a pure simulation (no physically deployed vehicles), both MOOS communities may simply be running on the same machine configured with distinct ports. The `pMOOSBridge` application is shown here for communication between MOOS communities, but there are other alternatives for inter-community communication and the operation of `uFldContactRangeSensor` is not dependent on the manner of inter-communication communications.

11.1 The uFldContactRangeSensor Interface and Configuration Options

The `uFldContactRangeSensor` application may be configured with a configuration block within a `.moos` file. Its interface is defined by its publications and subscriptions for MOOS variables consumed and generated by other MOOS applications. An overview of the set of configuration options and interface is provided in this section.

11.1.1 Configuration Parameters of uFldContactRangeSensor

The following parameters are defined for `uFldContactRangeSensor`. A more detailed description is provided in other parts of this section. Parameters having default values indicate so in parentheses below.

REACH_DISTANCE:	Distance out to which the pinging vehicle will be heard (100).
REPLY_DISTANCE:	Distance out to which the pinged vehicle will be heard (100).
PING_WAIT:	Minimum seconds enforced between pings (30).
PING_COLOR:	Visual preference: color of initiating ping message (white).
REPLY_COLOR:	Visual preference: color of replying message (chartreuse).
RN_ALGORITHM:	Algorithm for adding random noise to the range measurement
REPORT_VARS:	Determines variable name(s) used for range report ("short").
VERBOSE:	If true, verbose status message terminal output (false).

11.1.2 MOOS Variables Published by uFldContactRangeSensor

The primary output of `uFldContactRangeSensor` to the MOOSDB is posting of range reports and visual cues for the range reports.

CRS_RANGE_REPORT:	A report on the range from a particular vehicle to the pinging vehicle.
CRS_RANGE_REPORT_NAMEJ:	A report on the range from a particular named NAMEJ, to the pinging vehicle.
VIEW_RANGE_PULSE:	A description for visualizing the beacon range report.

The range report format may vary depending on user configuration. Some examples:

```
CRS_RANGE_REPORT = "name=alpha,range=129.2,time=19473362764.169"
CRS_RANGE_REPORT = "name=alpha,range=129.2,target=jackal,x=54,y=90,time=19473362987.428"
CRS_RANGE_REPORT_ALPHA = "range=129.2,time=19473362999.761"
```

The name of the vehicle requesting the report (generating the ping) may be embedded in the MOOS variable name to facilitate distribution of report messages to the appropriate vehicle with `pMOOSBridge`.

11.1.3 MOOS Variables Subscribed for by uFldContactRangeSensor

The `uFldContactRangeSensor` application will subscribe for the following four MOOS variables:

CRS_RANGE_REQUEST:	A request to generate range reports for all targets to all vehicles within range of the target.
NODE_REPORT:	A report on a vehicle location and status.
NODE_REPORT_LOCAL:	A report on a vehicle location and status.

11.1.4 Command Line Usage of uFldContactRangeSensor

The `uFldContactRangeSensor` application is typically launched as a part of a batch of processes by pAntler, but may also be launched from the command line by the user. The command line options may be shown by typing `"uFldContactRangeSensor --help"`:

Listing 27 - Command line usage for the uFldContactRangeSensor tool.

```
0  Usage: uFldContactRangeSensor file.moos [OPTIONS]
1
2  Options:
3    --alias=<ProcessName>
4        Launch uFldContactRangeSensor with the given process
5        name rather than uFldContactRangeSensor.
6    --example, -e
7        Display example MOOS configuration block
8    --help, -h
9        Display this help message.
10   --version,-v
11       Display the release version of uFldContactRangeSensor.
```

11.2 Configuring the uFldContactRangeSensor Parameters

The `uFldContactRangeSensor` simulator has two range configuration parameters:

```
REACH_DISTANCE = default = <meters>    // or "nolimit"
REPLY_DISTANCE = default = <meters>    // or "nolimit"
```

The simulator uses these parameters to decide whether a ping (range request) will reach other nearby vehicles. Two separate range parameters are used so that the simulator can support scenarios where some vehicles have a more powerful sonar than others, and some targets are harder to detect (absorb or deflect more energy) than others. Both parameters are given in meters.

When a range request is received by the simulator, it knows which vehicle is making the request since the requesting vehicle's name comprises the range request. For example:

```
MOOS Variable       Community   Value
-------------       ---------   -----
CRS_RANGE_REQUEST   archie      name=archie
```

The simulator maintains a record of the location of all known vehicles, by receiving `NODE_REPORT` messages regularly from each vehicle known to the simulator. When a range request is made, the simulator looks up the `REACH_DISTANCE` for the requesting vehicle, and the `REPLY_DISTANCE` for the target vehicle and if the sum, `REACH_DISTANCE + REPLY_DISTANCE` is less than the actual present range between the two vehicles, a range report is generated. For example:

```
MOOS Variable       Community   Value
-------------       ---------   -----
CRS_RANGE_REPORT    shoreside   vname=archie,range=126.54,target=jackal,time=19656022406.44
```

If the user provides no configuration parameters, all vehicles will default to have the same reach and reply distances of 100 meters. The default values may be overridden with something like:

```
REACH_DISTANCE = default = 120
REPLY_DISTANCE = default = 80
```

The above two lines, in effect, are the same as `REACH_DISTANCE = 100` and `REPLY_DISTANCE = 100`. Things become interesting when individual vehicles are given values different from the default. Consider the more advantageously configured vehicle, *victor*, below:

```
REACH_DISTANCE = victor = 250
REPLY_DISTANCE = victor = 20
```

If either the reach or reply distance for a given pair of vehicles is set to `nolimit`, then a range report will always be generated regardless of current range between the two vehicles. Future enhancements to this simulator module may include the factoring of vehicle speed and relative bearing to one another in the threshold determination of sending range reports.

11.3 Limiting the Frequency of Vehicle Range Requests

From the perspective of operating a vehicle, one may ask: why not request a range report (ping) as often as possible? There may be reasons why this is not feasible outside simulation. Limits may exist due to power budgets of the vehicle, and there may be prevailing protocols that make it at least impolite to be frequently pinging.

To reflect this limitation, the `uFldContactRangeSensor` utility may be configured to limit the frequency in which a vehicle's range request (or ping) will be honored with a range report reply. By default this frequency is set to once every 30 seconds for all vehicles. The default for all vehicles may be changed with the following configuration in the `.moos` file:

```
PING_WAIT = default = 60
```

The limits for a particular vehicle may be set with a similar configuration line:

```
PING_WAIT = henry = 90
```

11.4 Producing Range Measurements with Noise

In the default configuration of `uFldContactRangeSensor`, range reports are generated with the most precise range estimate as possible, with the only error being due to the latency of the communications generating the range request and range report. Additional noise/error may be added in the simulator for each range report with the following configuration parameter:

```
rn_algorithm   = uniform,pct=0.12     // Values in the range [0,1]
```

Currently the only noise algorithm supported is the generation of uniformly random noise on the range measurement. The noise level, θ, set with the parameter `rn_uniform_pct`, will generate a noisy range from an otherwise exact range measurement r, by choosing a value in the range $[\theta r, r + \theta r]$. The range without noise, i.e., the ground truth, may also be reported by the simulator if desired by setting the configuration parameter:

```
ground_truth = true
```

This will result in an additional MOOS variables posted, `CRS_RANGE_REPORT_GT`, with the same format as `CRS_RANGE_REPORT`, except the reported range will be given without noise.

11.5 An Example MOOS Configuration Block

As of MOOS-IvP Release 4.2, most if not all MOOS apps are implemented to support the `-e` or `--example` command-line switches. To see an example MOOS configuration block, enter the following from the command-line:

```
$ uFldContactRangeSensor -e
```

This will show the output shown in Listing 28 below.

Listing 28 - Example configuration of the uFldContactRangeSensor application.

```
 0  ============================================================
 1  uFldContactRangeSensor Example MOOS Configuration
 2  ============================================================
 3
 4  ProcessConfig = uFldContactRangeSensor
 5  {
 6    AppTick   = 4
 7    CommsTick = 4
18
19    // Configuring aspects of the vehicles in the sim
10    reach_distance = default = 100  // in meters or {nolimit}
11    reply_distance = default = 100  // in meters or {nolimit}
12    ping_wait      = default = 30   // in seconds
13
14    // Configuring manner of reporting
15    report_vars    = short // or {long, both}
16    ground_truth   = true  // or {false}
17    verbose        = true  // or {false}
18
19    // Configuring visual artifacts
20    ping_color     = white
21    reply_color    = chartreuse
22
23    // Configuring Artificial Noise
24    rn_algorithm   = uniform,pct=0
25  }
```

The console output uses color to discern default configurations from non-default configurations. Any line rendered as a default configuration may be omitted without any net change to the configuration.

11.5.1 Console Output Generated by uFldContactRangeSensor

Information regarding the startup of the `uFldContactRangeSensor` application may be monitored from an open console window where `uFldContactRangeSensor` is launched. If the verbose setting is turned on, further output is generated as the simulator progresses and receives range requests and generates range reports. The verbose setting may be turned on from the command line, `--verbose`, or in the mission file configuration block with `verbose=true`. Example output is shown below in Listing 29. Certain startup information, common across all MOOS application,s can be found in lines 0-13. The block of output in lines 16-36 provides startup information specific to `uFldContactRangeSensor`. This cannot be turned off with the verbose setting, and should appear in

blue in the console window. These lines convey the configuration settings of `uFldContactRangeSensor` read from the MOOS configuration block like the one shown previously in Listing 28.

Listing 29 - Example uFldContactRangeSensor console output.

```
 0  ************************************************
 1  *                                              *
 2  *       This is MOOS Client                    *
 3  *       c. P Newman 2001                       *
 4  *                                              *
 5  ************************************************
 6
 7  ---------------MOOS CONNECT----------------------
 8    contacting a MOOS server localhost:9200 -  try 00001
 9    Contact Made
10    Handshaking as "uFldContactRangeSensor"
11    Handshaking Complete
12    Invoking User OnConnect() callback...ok
13  ------------------------------------------------
14
15
16  Simulated Active Sonar starting...
17  ========================================
18  Acoustic Sonar Simulator Model:
19  ========================================
20  Default Reach Distance:   100
21  Default Reply Distance:   100
22  Default Ping Wait:        30
23  Random Noise Algorithm:   uniform
24  Random Noise Uniform Pct: 0.04
25  Ping Color:               white
26  Reply Color:              chartreuse
27  Ground Truth Reporting:   true
28  ReachMap:
29    VName: archie  Dist: 190
30  ReplyMap:
31    VName: jackal  Dist: 50
32  PingWaitMap:
33    VName: archie  PingWait: 125
34  Contact Records: 0
35  ========================================
36  Simulated Active Sonar started.
37
38  uFldContactRangeSensor is Running:
39           AppTick   @ 4.0 Hz
40           CommsTick @ 4 Hz
41  %*%*************************************************************
42  *************************************************************
43  **********************
44  Range request received from: archie
45     Elapsed time: 46.04512
46     Range Request accepted by uFldContactRangeSensor.
47     Range Request resets the clock for vehicle.
48     Range Report sent from targ vehicle[jackal] to receiver vehicle[archie]
49  DBPost: VIEW_RANGE_PULSE  x=-40.53,y=-33.66,radius=50,duration=6,fill=,label=archie_ping,
50                            edge_color=white,fill_color=white,time=10483324813.71,edge_size=1
51  DBPost: CRS_RANGE_REPORT  vname=archie,range=181.6211,target=jackal,time=10483324813.705
52  DBPost: CRS_RANGE_REPORT_GT  vname=archie,range=179.033,target=jackal,time=10483324813.705
53  DBPost: VIEW_RANGE_PULSE  x=-186.18,y=-137.77,radius=40,duration=15,label=jackal_ping_reply,
54                            edge_color=chartreuse,fill_color=chartreuse,time=10483324813.71
55  ******************************************************
56  Range request received from: archie
57     Elapsed time: 31.61322
58     Denied: Range Request exceeds maximum ping frequency.
```

```
59  ************************************************************
60  Range request received from: archie
61     Elapsed time: 57.75229
62     Range Request accepted by uFldContactRangeSensor.
63     Range Request resets the clock for vehicle.
64     Range Report sent from targ vehicle[jackal] to receiver vehicle[archie]
65  DBPost: VIEW_RANGE_PULSE  x=-140.04,y=-93.47,radius=50,duration=6,fill=,label=archie_ping,
66                            edge_color=white,fill_color=white,time=10483324871.46,edge_size=1
67  DBPost: CRS_RANGE_REPORT  vname=archie,range=132.5609,target=jackal,time=10483324871.457
68  DBPost: CRS_RANGE_REPORT_GT  vname=archie,range=132.2973,target=jackal,time=10483324871.457
69  DBPost: VIEW_RANGE_PULSE  x=-192.85,y=-214.77,radius=40,duration=15,label=jackal_ping_reply,
70                            edge_color=chartreuse,fill_color=chartreuse,time=10483324871.46
71  ********************************************************************************************
```

Unless the verbose setting is turned on, the output ending on line 40 above should be the last output written to the console for the duration of the simulator.

In the verbose mode, the simulator will produce event-based output as shown in the example above beginning on line 41. The asterisks in lines 41-43 are not merely visual separators. An asterisk represents a single receipt of a `NODE_REPORT` message. Receiving node reports is essential for the operation of the simulator and this provides a bit of visual verification that this is indeed occurring. Presumably node reports are being received much more often than range requests and range reports are handled, as is the case in the above example. The first time a node report is received for a particular vehicle, rather than an asterisk output, an percent-sign, `'%'`, is output instead, as on line 41 for the two vehicles in this example.

In addition to handling incoming node reports, on any given iteration, the simulator may also handle an incoming range request. Console output for incoming range requests may look like that shown in lines 44-54 above. First the range request and the requesting vehicle is announced as online 44. The elapsed time since the vehicle last made a range request is shown as on line 45. If the request is honored by the simulator, this is indicated as shown on line 46. Otherwise a reason for denial may be shown, as on line 58. If the request is accepted, range reports may be generated for one or more vehicles. For each such vehicle, a line showing the new report is generated, as on line 67.

If artificial noise is being applied by the simulator, e.g., as in line 26 in Listing 28, and the `ground_truth` parameter is set to `true`, then `uFldContactRangeSensor` will also generate a "ground truth" report alongside the normal range report. This is indicated in the console output as in lines 52 and 68 above. This ground truth report may not be communicated to the vehicle, but may be used later for post-mission analysis.

11.6 Interaction between uFldContactRangeSensor and pMarineViewer

The uFldContactRangeSensor application will post certain messages to the MOOSDB that may be subscribed for by GUI based applications like pMarineViewer for visualizing the posting of CRS_RANGE_REPORT and CRS_RANGE_REQUEST messages. A *range pulse* message is used by the uFldContactRangeSensor application to convey visually the generation of a range report, or the receipt of a range request. The pulse is rendered as a ring with a growing radius having the vehicle at the center. A pulse emanating By default different colors may be used for range requests and range reports. The RANGE_PULSE message is a data structure implemented in the XYRangePulse class, and usually passed through the MOOS variable VIEW_RANGE_PULSE with the following format:

```
VIEW_RANGE_PULSE = <pulse>
```

The <pulse> component is a series of comma-separated parameter=value pairs. The supported parameters are: x, y, radius, duration, time, fill, fill_color, label, edge_color, and edge_size.

The x and y parameters convey the center of the pulse. The radius parameter indicates the radius of the circle at its maximum. The duration parameter is the number of seconds the pulse will be rendered. The pulse will grow its radius linearly from zero meters at zero seconds to radius meters at duration seconds. The fill parameter is in the range $[0, 1]$, where 0 is full transparency (clear) and 1 is fully opaque. The pulse transparency increases linearly as the range ring is rendered. The starting transparency at $radius = 0$ is given by the fill parameter. The transparency at the maximum radius is zero. The fill_color parameter specifies the color rendered to the internal part of the range pulse. The choice of legal colors is described in Appendix A. The label is a string that uniquely identifies the range instance to consumers like pMarineViewer. As with other objects in pMarineViewer, if it receives an object the same label and type as one previously received, it will replace the old object with the new one in its memory. The edge_color parameter describes the color of the ring rendered in the range pulse. The edge_size likewise describes the line width of the rendered ring. The time parameter indicates the UTC time at which the range pulse object was generated.

Below is an example string representing a range pulse. Each field, with the exception of the x, y position, label, and time, indicates the default values if left unspecified:

```
VIEW_RANGE_PULSE = x=-40,y=-150,radius=40,duration=15,fill=0.25,fill_color=green,label=04
                   edge_color=green,time=3892830128.5,edge_size=1
```

One further note to developers of other apps perhaps wishing to also generate a range pulse for viewing - the recommended manner to generate a range pulse string is to create an instance of the XYRangePulse class using the defined class interface. The string should be obtained by invoking the serialization method for that class. This will better ensure compatibility as the software evolves. The class is defined in lib_geometry in the MOOS-IvP tree.

11.7 The Hugo Example Mission Using uFldContactRangeSensor

The *hugo* mission is distributed with the MOOS-IvP source code and contains a ready example of the uFldContactRangeSensor application. Assuming the reader has downloaded the source code available at www.moos-ivp.org and built the code according to the discussion in Section 1.5, the *hugo* mission may be launched by:

```
$ cd moos-ivp/ivp/missions/m8_hugo/
$ ./launch.sh 10
```

The argument, 10, in the line above will launch the simulation in `10x` real time. Once this launches, the `pMarineViewer` GUI application should launch and the mission may be initiated by hitting the `DEPLOY` button. Two vehicles should be visibly moving, one surface vehicle labeled `"archie"` and one UUV labeled `"jackal"`, as shown in Figure 27.

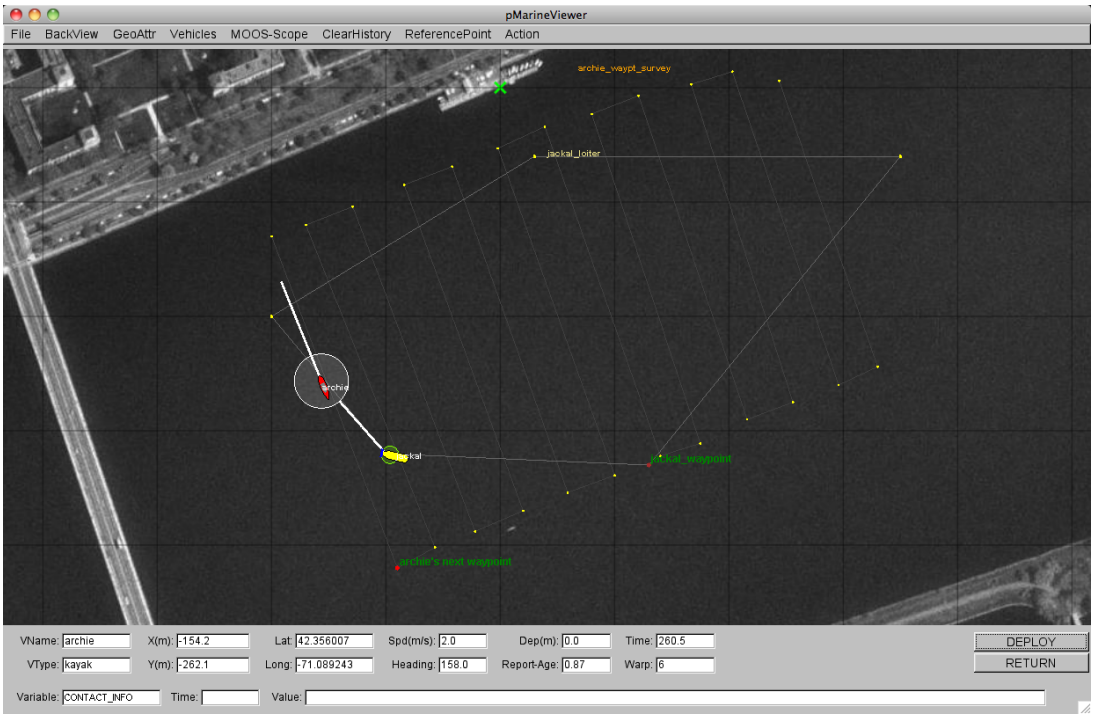


Figure 27: **The Hugo Example Mission:** The Hugo example mission involves two simulated vehicles. The first vehicle is a surface vehicle, *archie*, traversing a lawnmower shaped search pattern. The second vehicle, is a UUV, *jackal*, traversing a polygon pattern overlapping the first pattern. Periodically *archie* emits a ping (range request). This example contains three distinct MOOS communities - one for each simulated vehicle, and one for the shoreside community running `uFldContactRangeSensor`.

The surface vehicle will traverse a survey pattern over the region shown in Figure 27, periodically generating a range request (ping). With each range request, a white range pulse should be visible around the vehicle. Almost immediately afterwards, a range report for each neighboring vehicle within range is generated and a green range pulse around each "target" vehicle is rendered. The snapshot in Figure 27 depicts a moment in time where the range request visual pulses are around both vehicles. The larger pulse around the surface vehicle indicates it was generated just prior to the reply pulse around the UUV.

How does the simulated vehicle generate a range request? In practice a user may implement an intelligent module to reason about when to generate requests, but in this case the `uTimerScript` application is used by creating a script that repeats endlessly, generating a range request once every 25-35 seconds. The script is also conditioned on `(NAV_SPEED > 0)`, so the pinging doesn't

start until the vehicle is deployed. The configuration for the script can be seen in the uTimerScript configuration block in the shoreside.moos file. More on the uTimerScript application can be found in the uTimerScript section in [1].

Examining the Log Data from the Hugo Mission

After the launch script above has launched the simulation, the script should leave the console user with the option to "Exit and Kill Simulation" by hitting the '2' key. Once the vehicles have been deployed and traversed to one's satisfaction, exit the script. Three log directories should have been created by three separate invocations of the pLogger application in the directory where the simulation was launched. The three directories correspond to the three MOOS communities participating in this simulation, and should have the corresponding prefixes, LOG_ARCHIE, LOG_JACKAL, and LOG_SHORESIDE, with the remainder of the directory names composed from the timestamp of the launch.

Let's take a look at some of the data related to the simulation and uFldContactRangeSensor in particular. The uFldContactRangeSensor app is part of the shoreside community (see Figure 26). The shoreside log file resides in the LOG_SHORESIDE_* directory, and has the .alog suffix. A dump of the entire file reveals a deluge of information. To look at the information relevant to the uFldContactRangeSensor application, the file is pruned with the aloggrep tool described in the Alog Toolbox section in [1]:

```
$ aloggrep LOG_SHORESIDE_12_7_2011_____08_12_05.alog uFldContactRangeSensor uTimerScript
```

This produces a subset of the alog file similar to that shown in Listing 30, showing only log entries made by either the uFldContactRangeSensor application, or the uTimerScript application which generated all the range requests as described above.

Even though the below log file represents the logged data from the shoreside MOOS community, it also contains entries of postings bridged from other communities. Upon each range request generated by uTimerScript running on the *archie* vehicle, it its bridged to the shoreside community and logged as CRS_RANGE_REQUEST entries shown below. If the simulator decides to honor the range request, it posts a VIEW_RANGE_PULSE message which can be seen below with the label "archie_ping". If the range request is honored by the simulator it also generates the CRS_RANGE_REPORT message. If the simulator is configured to add random noise to the range measurements, and to report ground truth alongside the noisy measurements, the simulator will also post CRS_RANGE_REPORT_GT message as shown below.

In cases where the range request is *not* honored by the simulator (perhaps because it violated the minimum wait time between pings), the CRS_RANGE_REQUEST message is simply logged by the logger and ignored by the simulator. See the entry at timestamp 104.761 below.

Listing 30 - A subset of the Shoreside log file in the Hugo example mission.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% LOG FILE:       ./LOG_SHORESIDE_12_7_2011_____08_12_05/LOG_SHORESIDE_12_7_2011_____08_12_05.alog
%% FILE OPENED ON  Tue Jul 12 08:12:05 2011
%% LOGSTART               23588509053.6
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
76.758   CRS_RANGE_REQUEST   uTimerScript     name=archie
78.157   VIEW_RANGE_PULSE       uFldContactRangeSensor x=-49.31,y=-38.86,radius=50,duration=6,label=archie_ping,...
```

```
78.157   CRS_RANGE_REPORT    uFldContactRangeSensor vname=archie,range=176.6565,target=jackal,time=23588509131.764
78.157   VIEW_RANGE_PULSE       uFldContactRangeSensor x=-187.67,y=-144.6,radius=40,duration=15,label=jackal_reply,...
78.157   CRS_RANGE_REPORT_GT uFldContactRangeSensor vname=archie,range=174.1391,target=jackal,time=23588509131.764
104.761  CRS_RANGE_REQUEST   uTimerScript     name=archie
130.417  CRS_RANGE_REQUEST   uTimerScript     name=archie
132.118  VIEW_RANGE_PULSE       uFldContactRangeSensor x=-141.97,y=-94.55,radius=50,duration=6,label=archie_ping,...
132.119  CRS_RANGE_REPORT    uFldContactRangeSensor vname=archie,range=131.1348,target=jackal,time=23588509185.725
132.119  CRS_RANGE_REPORT_GT uFldContactRangeSensor vname=archie,range=130.8741,target=jackal,time=23588509185.725
132.119  VIEW_RANGE_PULSE       uFldContactRangeSensor x=-191.18,y=-215.82,radius=40,duration=15,label=jackal_reply,...
163.599  CRS_RANGE_REQUEST   uTimerScript     name=archie
165.819  VIEW_RANGE_PULSE       uFldContactRangeSensor x=-197.98,y=-128.86,radius=50,duration=6,label=archie_ping,...
165.819  CRS_RANGE_REPORT    uFldContactRangeSensor vname=archie,range=122.0918,target=jackal,time=23588509219.426
165.819  CRS_RANGE_REPORT_GT uFldContactRangeSensor vname=archie,range=127.1016,target=jackal,time=23588509219.426
165.820  VIEW_RANGE_PULSE       uFldContactRangeSensor x=-161.42,y=-250.59,radius=40,duration=15,label=jackal_reply,...
190.192  CRS_RANGE_REQUEST   uTimerScript     name=archie
216.311  CRS_RANGE_REQUEST   uTimerScript     name=archie
217.910  VIEW_RANGE_PULSE       uFldContactRangeSensor x=-168.48,y=-226,radius=50,duration=6,label=archie_ping,...
217.910  CRS_RANGE_REPORT    uFldContactRangeSensor vname=archie,range=96.955,target=jackal,time=23588509271.516
217.910  CRS_RANGE_REPORT_GT uFldContactRangeSensor vname=archie,range=98.0072,target=jackal,time=23588509271.516
217.910  VIEW_RANGE_PULSE       uFldContactRangeSensor x=-112.19,y=-306.23,radius=40,duration=15,label=jackal_reply,...
245.493  CRS_RANGE_REQUEST   uTimerScript     name=archie
273.729  CRS_RANGE_REQUEST   uTimerScript     name=archie
275.877  VIEW_RANGE_PULSE       uFldContactRangeSensor x=-125.38,y=-332.68,radius=50,duration=6,label=archie_ping,...
275.877  CRS_RANGE_REPORT    uFldContactRangeSensor vname=archie,range=88.6767,target=jackal,time=23588509329.484
275.877  CRS_RANGE_REPORT_GT uFldContactRangeSensor vname=archie,range=85.9737,target=jackal,time=23588509329.484
275.877  VIEW_RANGE_PULSE       uFldContactRangeSensor x=-39.7,y=-325.58,radius=40,duration=15,label=jackal_reply,...
```

In this example, the range requests are made by *archie*, and range reports are generated by `uFldContactRangeSensor`, and reported back to *archie*, but not the other vehicle. A quick look at the log file for *archie* reveals this, in Listing 31 below. Each range request is followed by a range report unless it violates the minimum ping wait time set in the simulator, which in this case is 30 seconds.

```
$ cd missions/m8_hugo/LOG_ARCHIE_12_7_2011_____08_12_03/
$ aloggrep LOG_ARCHIE_12_7_2011_____08_12_03.alog uFldContactRangeSensor uTimerScript
```

Listing 31 - A subset of the Archie log file in the Hugo example mission.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% LOG FILE:       ./LOG_ARCHIE_12_7_2011_____08_12_03/LOG_ARCHIE_12_7_2011_____08_12_03.alog
%% FILE OPENED ON  Tue Jul 12 08:12:03 2011
%% LOGSTART               23588509017.6
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
112.767   CRS_RANGE_REQUEST  uTimerScript     name=archie
114.166   CRS_RANGE_REPORT   uFldContactRangeSensor vname=archie,range=176.6565,target=jackal,time=23588509131.764
140.770   CRS_RANGE_REQUEST  uTimerScript           name=archie
166.426   CRS_RANGE_REQUEST  uTimerScript           name=archie
168.127   CRS_RANGE_REPORT   uFldContactRangeSensor vname=archie,range=131.1348,target=jackal,time=23588509185.725
199.608   CRS_RANGE_REQUEST  uTimerScript           name=archie
201.828   CRS_RANGE_REPORT   uFldContactRangeSensor vname=archie,range=122.0918,target=jackal,time=23588509219.426
226.201   CRS_RANGE_REQUEST  uTimerScript           name=archie
252.319   CRS_RANGE_REQUEST  uTimerScript           name=archie
253.919   CRS_RANGE_REPORT   uFldContactRangeSensor vname=archie,range=96.955,target=jackal,time=23588509271.516
281.502   CRS_RANGE_REQUEST  uTimerScript           name=archie
309.738   CRS_RANGE_REQUEST  uTimerScript           name=archie
311.886   CRS_RANGE_REPORT   uFldContactRangeSensor vname=archie,range=88.6767,target=jackal,time=23588509329.484
344.566   CRS_RANGE_REQUEST  uTimerScript           name=archie
```

A Colors

Below are the colors used by IvP utilities that use colors. Colors are case insensitive. A color may be specified by the string as shown, or with the '_' character as a separator. Or the color may be specified with its hexadecimal or floating point form. For example the following are equivalent: "darkblue", "DarkBlue", "dark_blue", "hex:00,00,8b", and "0,0,0.545". In the latter two styles, the '%', '$', or '#' characters may also be used as a delimiter instead of the comma if it helps when embedding the color specification in a larger string that uses its own delimeters. Mixed delimeters are not supported however.

antiquewhite, (fa,eb,d7)
aqua (00,ff,ff)
aquamarine (7f,ff,d4)
azure (f0,ff,ff)
beige (f5,f5,dc)
bisque (ff,e4,c4)
black (00,00,00)
blanchedalmond(ff,eb,cd)
blue (00,00,ff)
blueviolet (8a,2b,e2)
brown (a5,2a,2a)
burlywood (de,b8,87)
cadetblue (5f,9e,a0)
chartreuse (7f,ff,00)
chocolate (d2,69,1e)
coral (ff,7f,50)
cornsilk (ff,f8,dc)
cornflowerblue(64,95,ed)
crimson (de,14,3c)
cyan (00,ff,ff)
darkblue (00,00,8b)
darkcyan (00,8b,8b)
darkgoldenrod (b8,86,0b)
darkgray (a9,a9,a9)
darkgreen (00,64,00)
darkkhaki (bd,b7,6b)
darkmagenta (8b,00,8b)
darkolivegreen(55,6b,2f)
darkorange (ff,8c,00)
darkorchid (99,32,cc)
darkred (8b,00,00)
darksalmon (e9,96,7a)
darkseagreen (8f,bc,8f)
darkslateblue (48,3d,8b)
darkslategray (2f,4f,4f)
darkturquoise (00,ce,d1)
darkviolet (94,00,d3)
deeppink (ff,14,93)
deepskyblue (00,bf,ff)
dimgray (69,69,69)
dodgerblue (1e,90,ff)
firenrick (b2,22,22)
floralwhite (ff,fa,f0)
forestgreen (22,8b,22)
fuchsia (ff,00,ff)
gainsboro (dc,dc,dc)
ghostwhite (f8,f8,ff)
gold (ff,d7,00)
goldenrod (da,a5,20)
gray (80,80,80)
green (00,80,00)
greenyellow (ad,ff,2f)
honeydew (f0,ff,f0)
hotpink (ff,69,b4)
indianred (cd,5c,5c)
indigo (4b,00,82)
ivory (ff,ff,f0)
khaki (f0,e6,8c)
lavender (e6,e6,fa)
lavenderblush (ff,f0,f5)
lawngreen (7c,fc,00)
lemonchiffon (ff,fa,cd)
lightblue (ad,d8,e6)
lightcoral (f0,80,80)
lightcyan (e0,ff,ff)
lightgoldenrod(fa,fa,d2)
lightgray (d3,d3,d3)
lightgreen (90,ee,90)

lightpink (ff,b6,c1)
lightsalmon (ff,a0,7a)
lightseagreen (20,b2,aa)
lightskyblue (87,ce,fa)
lightslategray(77,88,99)
lightsteelblue(b0,c4,de)
lightyellow (ff,ff,e0)
lime (00,ff,00)
limegreen (32,cd,32)
linen (fa,f0,e6)
magenta (ff,00,ff)
maroon (80,00,00)
mediumblue (00,00,cd)
mediumorchid (ba,55,d3)
mediumseagreen(3c,b3,71)
mediumslateblue(7b,68,ee)
mediumspringgreen(00,fa,9a)
mediumturquoise(48,d1,cc)
mediumvioletred(c7,15,85)
midnightblue (19,19,70)
mintcream (f5,ff,fa)
mistyrose (ff,e4,e1)
moccasin (ff,e4,b5)
navajowhite (ff,de,ad)
navy (00,00,80)
oldlace (fd,f5,e6)
olive (80,80,00)
olivedrab (6b,8e,23)
orange (ff,a5,00)
orangered (ff,45,00)
orchid (da,70,d6)
palegreen (98,fb,98)
paleturquoise (af,ee,ee)
palevioletred (db,70,93)
papayawhip (ff,ef,d5)
peachpuff (ff,da,b9)
pelegoldenrod (ee,e8,aa)
peru (cd,85,3f)
pink (ff,c0,cb)
plum (dd,a0,dd)
powderblue (b0,e0,e6)
purple (80,00,80)
red (ff,00,00)
rosybrown (bc,8f,8f)
royalblue (41,69,e1)
saddlebrowm (8b,45,13)
salmon (fa,80,72)
sandybrown (f4,a4,60)
seagreen (2e,8b,57)
seashell (ff,f5,ee)
sienna (a0,52,2d)
silver (c0,c0,c0)
skyblue (87,ce,eb)
slateblue (6a,5a,cd)
slategray (70,80,90)
snow (ff,fa,fa)
springgreen (00,ff,7f)
steelblue (46,82,b4)
tan (d2,b4,8c)
teal (00,80,80)
thistle (d8,bf,d8)
tomatao (ff,63,47)
turquoise (40,e0,d0)
violet (ee,82,ee)
wheat (f5,de,b3)
white (ff,ff,ff)
whitesmoke (f5,f5,f5)
yellow (ff,ff,00)
yellowgreen (9a,cd,32)

References

[1] Michael R. Benjamin. MOOS-IvP Autonomy Tools Users Manual Release 4.2.1. Technical Report MIT-CSAIL-TR-2011-036, MIT Computer Science and Artificial Intelligence Lab, July 2011.

[2] Michael R. Benjamin, Paul M. Newman, Henrik Schmidt, and John J. Leonard. An Overview of MOOS-IvP and a Users Guide to the IvP Helm Autonomy Software. Technical Report MIT-CSAIL-TR-2010-041, MIT Computer Science and Artificial Intelligence Lab, August 2010.

[3] Mary M. Hunt, William M. Marquet, Donald A. Moller, Kenneth R. Peal, Woollcott K. Smith, and Rober C. Spindel. An Acoustic Navigation System. Technical Report WHOI-74-6, Woods Hole Oceanographic Institution, Woods Hole, Massachusetts, December 1974.

[4] P. A. Milne. *Underwater Acoustic Positioning Systems*. Gulf Publishing Co., Houston TX, January 1983.

[5] Roger P. Sokey and Thomas C. Austin. Sequential Long-Baseline Navigation for REMUS, an Autonomous Underwater Vehicle. *Proceedings of SPIE, the Internation Society for Optical Engineering*, 3711:212–219a, 1999.

[6] Louis L. Whitcomb, Dana R. Yoerger, and Hanumant Singh. Combined Doppler/LBL Based Navigation of Underwater Vehicles. In *11th International Symposium on Unmanned Untethered Submersible Technology (UUST99)*, Durham, New Hampshire, August 1999.

Index

Command Line Usage
  pHostInfo, 28
  uFldBeaconRangeSensor, 72
  uFldContactRangeSensor, 86
  uFldHazardSensor, 56
  uFldMessageHandler, 42
  uFldNodeBroker, 16
  uFldNodeComms, 33
  uFldPathCheck, 51
  uFldScope, 46
  uFldShoreBroker, 21
Configuration Parameters
  pHostInfo, 28
  uFldBeaconRangeSensor, 70, 72
  uFldContactRangeSensor, 85, 86
  uFldHazardSensor, 55, 56
  uFldMessageHandler, 42
  uFldNodeBroker, 15, 16
  uFldNodeComms, 32, 33
  uFldPathCheck, 51
  uFldScope, 46
  uFldShoreBroker, 20, 21

MOOS
  Acronym, 10
  Background, 9
  Documentation, 12
  Operating Systems, 11
  Source Code, 10
  Sponsors, 10

pHostInfo, 27
  Command Line Usage, 28
  Configuration Parameters, 28
    TEMP_FILE_DIR, 28
  Publications, 28
  Subscriptions, 28
Publications
  pHostInfo, 28
  uFldBeaconRangeSensor, 71
  uFldContactRangeSensor, 86
  uFldHazardSensor, 55
  uFldMessageHandler, 42
  uFldNodeBroker, 15
  uFldNodeComms, 32
  uFldPathCheck, 51
  uFldScope, 46
  uFldShoreBroker, 20

Source Code
  Building, 10
  Obtaining, 10
Subscriptions
  pHostInfo, 28
  uFldBeaconRangeSensor, 71
  uFldContactRangeSensor, 86
  uFldHazardSensor, 56
  uFldMessageHandler, 42
  uFldNodeBroker, 15
  uFldNodeComms, 33
  uFldPathCheck, 51
  uFldScope, 46
  uFldShoreBroker, 20

uFldBeaconRangeSensor, 69
  Command Line Usage, 72
  Configuration Parameters, 70, 72
    BEACON, 71
    DEFAULT_BEACON_COLOR, 71
    DEFAULT_BEACON_FREQ, 71
    DEFAULT_BEACON_REPORT_RANGE, 71
    DEFAULT_BEACON_SHAPE, 71
    DEFAULT_BEACON_WIDTH, 71
    GROUND_TRUTH, 71
    PING_PAYMENTS, 71
    PING_WAIT, 71
    REACH_DISTANCE, 71
    REPORT_VARS, 71
    RN_ALGORITHM, 71
    VERBOSE, 71
  Publications, 71
  Subscriptions, 71
uFldContactRangeSensor, 84
  Command Line Usage, 86

Configuration Parameters, 85, 86
PING_COLOR, 85
PING_WAIT, 85
REACH_DISTANCE, 85
REPLY_COLOR, 85
REPLY_DISTANCE, 85
REPORT_VARS, 85
RN_ALGORITHM, 85
VERBOSE, 85
Publications, 86
Subscriptions, 86
uFldHazardSensor, 53
Command Line Usage, 56
Configuration Parameters, 55, 56
DEFAULT_HAZARD_COLOR, 55
DEFAULT_HAZARD_SHAPE, 55
DEFAULT_HAZARD_WIDTH, 55
HAZARD_FILE, 55
OPTIONS_SUMMARY_INTERVAL, 55
SEED_HAZARDS, 55
SEED_RANDOM, 55
SEED_REPORTS, 55
SEED_SWATH, 55
SENSOR_CONFIG, 55
SWATH_LENGTH, 55
SWATH_TRANSPARENCY, 55
VERBOSE, 55
Publications, 55
Subscriptions, 56
uFldMessageHandler, 41
Command Line Usage, 42
Configuration Parameters, 42
STRICT_ADDRESSING, 42
VERBOSE, 42
Publications, 42
Subscriptions, 42
uFldNodeBroker, 14
Command Line Usage, 16
Configuration Parameters, 15, 16
BRIDGE, 15
KEYWORD, 15
TRY_SHORE_HOST, 15
Publications, 15
Subscriptions, 15
uFldNodeComms, 31
Command Line Usage, 33
Configuration Parameters, 32, 33
COMMS_RANGE, 32, 34, 35
CRITICAL_RANGE, 32, 35
DEBUG, 32
EARANGE, 32, 36
GROUPS, 32
MIN_MSG_INTERVAL, 32, 37
MIN_MSG_LENGTH, 32, 37
STALE_TIME, 32, 35
STEALTH, 32, 35
VERBOSE, 32
Publications, 32
Subscriptions, 33
uFldPathCheck, 50
Command Line Usage, 51
Configuration Parameters, 51
HISTORY, 51
Publications, 51
Subscriptions, 51
uFldScope, 45
Command Line Usage, 46
Configuration Parameters, 46
LAYOUT, 46
SCOPE, 46
Publications, 46
Subscriptions, 46
uFldShoreBroker, 19
Command Line Usage, 21
Configuration Parameters, 20, 21
BRIDGE, 20
QBRIDGE, 20, 24, 25
Publications, 20
Subscriptions, 20
uTimerScript, 82, 93
Simulated Range Requests, 82, 93

Index of MOOS Variables

BRS_RANGE_REPORT, 71
BRS_RANGE_REQUEST, 71
CRS_RANGE_REPORT, 86
CRS_RANGE_REQUEST, 86
HOST_INFO_REQUEST, 28
NODE_BROKER_ACK, 15, 17, 20
NODE_BROKER_PING, 15, 18, 20
NODE_MESSAGE, 32, 33, 37, 42
NODE_REPORT_LOCAL, 33, 51, 56, 71, 86
NODE_REPORT, 32, 33, 36, 51, 56, 71, 86
PHI_HOST_INFO, 15, 17, 18, 20
PHI_HOST_IP_ALL, 28
PHI_HOST_IP_VERBOSE, 28
PHI_HOST_IP, 28
PHI_HOST_PORT_UDP, 28
PHI_HOST_PORT, 28
PMB_REGISTER, 15, 17, 20
PMB_UDP_LISTEN, 28
UHZ_CONFIG_ACK, 55
UHZ_HAZARD_REPORT, 55
UHZ_OPTIONS_SUMMARY, 55
UHZ_SENSOR_CONFIG, 56
UHZ_SENSOR_REQUEST, 56
UMH_SUMMARY_MSGS, 42
UNC_EARANGE, 33, 36
UNC_STEALTH, 33, 36
UPC_ODOMETRY_REPORT, 51
UPC_SPEED_REPORT, 51
UPC_TRIP_RESET, 51
VIEW_CIRCLE, 55
VIEW_COMMS_PULSE, 32
VIEW_MARKER, 71, 86
VIEW_POLYGON, 55
VIEW_RANGE_PULSE, 71, 86