Goby Underwater Autonomy Project

User Manual for Version 1.0

<https://launchpad.net/goby>

Contents

1 Introduction

1.1 What is Goby?

The Goby Underwater Autonomy Project is an autonomy architecture tailored for marine robotics. It can be considered a direct descendant of the MOOS [1], with inspiration from LCM [2]. The motivation for Goby was the desire to seamlessly integrate acoustic networking (and other low bandwidth channels found in marine robotics) into the autonomy middleware.

The Goby autonomy architecture (`goby-core`) is still in rapid experimental development but you are welcome to begin playing with it. The Goby Acoustic Communications libraries (`goby-acomms`) form the majority of the stable contribution for Version 1.0. See the Developers' documentation for details on these libraries at [3]. Users of the MOOS application `pAcommsHandler` should see Appendix A.

Goby allows you to

- create custom applications (hereafter Goby applications) that communicate with other Goby applications in a publish/subscribe manner using custom designed message objects provided by the Google Protocol Buffers (protobuf) project [4]. This message passing is mediated by an application called the Goby daemon (`gobyd`)
- log message data using a choice of Structured Query Language (SQL) backends (SQLite3 [5] or PostgreSQL [6]), allowing a choice between simplicity and power. This SQL logger is seamlessly integrated with the protobuf messaging. SQL provides a well-known and standards compliant way to easily access data at runtime and during post-processing.
- log debugging output in a flexible manner to either the terminal window or a file or both, with fine-grained control over the verbosity.
- robustly configure your Goby applications both using a text configuration file and/or command line options by writing a configuration schema in protobuf. Gone are the days of manual command line and configuration file parsing and validity checking. Only fields allowed in the schema are accepted by the parser, greatly reducing syntax errors in the configuration files.

1.2 Structure of this Manual

This manual is designed to start slow with introductory features and then ramp up to more powerful features for advanced users. Please read as far as you wish and then as soon as possible get your feet wet. In fact, you may want to go download and install Goby now before reading further: `https://launchpad.net/goby`. Once you are familiar with the workings of Goby, you will be interested in reading the separate Developers' manual available at [3].

1.3 How to get help

The Goby community is here to support you. This is an open source project so we have limited time and resources, but you will find that many are willing to contribute their help, with the hope that you will do the same as you gain experience. Please consult these resources and people, probably in this order of preference:

1. This user manual.
2. Questions and Answers on Launchpad: `https://answers.launchpad.net/goby`.
3. The developers' documentation: `http://gobysoft.com/doc`.
4. Email the listserver goby@mit.edu. Please sign up first: `http://mailman.mit.edu/mailman/listinfo/goby`.
5. Email the lead developer (T. Schneider): tes@mit.edu.

2 The Hello World example

Goby is currently written entirely in C++. We hope to support more languages in the future, but we feel that C++ is a good blend of elegance, speed, and power. While the core of Goby is based on a number of advanced C++ techniques, you only need a small amount of C++ knowledge to get started writing your own Goby application. If you are new to programming and C++, we recommend Prata's *C++ Primer Plus* [7]. If you are experienced in programming but new to C++, we recommend Stroustrup's *The C++ Programming Language* [8]. The website `www.cplusplus.com` is an excellent online reference.

This complete example is located at the end of this chapter in section 2.7. It's probably a good idea to download and install Goby now so you can try this out for yourself: `https://launchpad.net/goby`.

This example involves passing a single type of message (class HelloWorldMsg) from one Goby application (`hello_world1_g`[1]) to another (`hello_world2_g`). Since Goby has a star topology, `gobyd` will mediate this transaction.

For this example we will write two Goby applications and one protobuf message. See Fig. 2.1 for the software structure of this example.

2.1 Meeting goby::core::ApplicationBase

`goby::core::ApplicationBase` is the building block (base class) upon which we will make our Goby applications (which will be derived classes of ApplicationBase). `ApplicationBase` provides us with a number of tools; the main ones are:

- a constructor `ApplicationBase()` that reads the command line and configuration (we will learn about this later) and connects to the Goby daemon (`gobyd`) for us.
- a virtual method `loop()` that is called at a regular frequency (10 Hertz by default).
- a method `subscribe()` which tells `gobyd` that we wish to receive all messages of this type.
- a method `newest()` which returns the newest (latest received) message of a given type that we have previously called `subscribe()` for. We will learn how to filter the subscriptions later.

[1] you can name your applications whatever you want, but we like appending "_g" to the end to indicate that this is a Goby application.

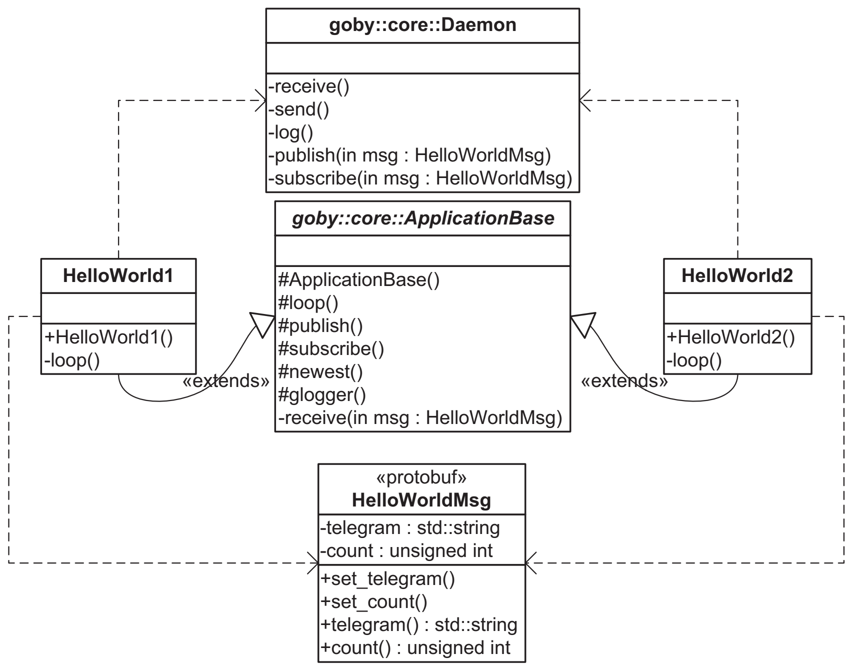

Figure 2.1: Structure diagram of the Hello World example showing the classes and dependencies. `HelloWorld1` and `HelloWorld2` are both Goby applications and thus are classes derived (solid arrows) from `goby::core::ApplicationBase`. Both of them (dashed arrows) depend on the Protocol Buffers message `HelloWorldMsg` because they use it to communicate. They also both depend on `goby::core::Daemon` (gobyd) for passing this message. See Fig. 2.2 for the sequence of sending a message in this example.

- a method `publish()` allowing us to publish messages to `gobyd` and thereby to any subscribers of that type.
- a method `glogger()` which acts just like std::cout[2] and lets us write to the debug (terminal window / text file) goby *logger*.

[2] `glogger()` accesses an instantiation of goby::util::FlexOstream, a derived class of std::ostream. std::cout is also a derived class of std::ostream.

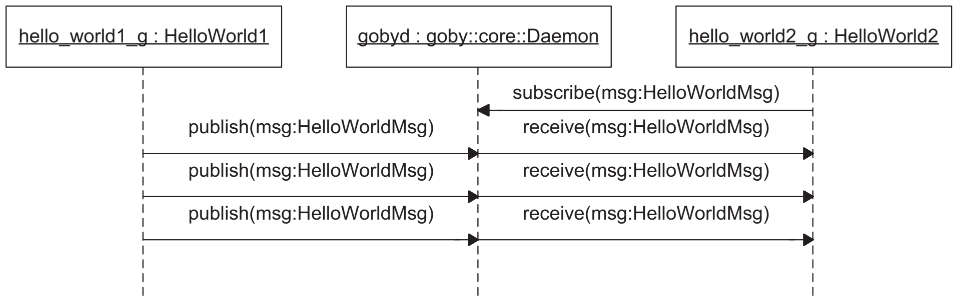

Figure 2.2: Sequence diagram of the Hello World example. `HelloWorld2` subscribes for all messages of type `HelloWorldMsg` after which `HelloWorld1` publishes repeatedly a `HelloWorldMsg` that is processed and sent by `gobyd` to all subscribers (in this case just `HelloWorld2`).

2.2 Creating a simple Google Protocol Buffers Message: HelloWorldMsg

Google Protocol Buffers (protobuf) allows us to create custom objects for holding and transmitting data in a structured fashion. Transmitting data typically is done in a long string of bytes. However, humans do not view the world as a string of bytes. We think and communicate using tangible and intangible objects. For example, a baseball might be described by its diameter, color, weight, and materials. Goby (using protobuf objects) allows messages to be formed using this more natural object-based representation.

The protobuf language is simple with a syntax similar to that of C. Protobuf messages are written in .proto files and passed to the protobuf compiler (`protoc`) which generates C++ code to pass to the C++ compiler (`gcc` on Linux). Protobuf messages can contain a number of basic types (or vectors of these types) as well as nested messages. Fields are labeled as required, optional or repeated (essentially a vector). Required fields must be filled in; clearly, optional fields can be omitted. This might be a good time to read the Protocol Buffers tutorial [9] to get a feel for the language and usage.

As you become familiar with using protobuf, the language reference [10] will help you in creating .proto files and the generated code reference [11] will assist you in accessing the C++ classes created by the .proto files when passed through `protoc`.

For this example, we wish to send "hello world" (of course) so we need a string to hold our message that we will call 'telegram':

```
required string telegram = 1;
```

The `= 1` simply indicates that 'telegram' is the first field in the message `HelloWorldMsg`. Furthermore, we want to keep track of how many times we've said "hello" so we'll add an unsigned integer called 'count'

```
required uint32 count = 2;
```

The resulting .proto file is given in section 2.7.1.

We chose 'required' to prefix both fields because we feel that a valid `HelloWorldMsg` must contain both a 'telegram' and a 'count.' `uint32` is an unsigned (non-negative) 32 bit integer. The numbers following the "=" sign are unique identifiers for each field. These numbers can be chosen however one likes as long as they are unique within a given protobuf message. Ascending numbers in the order fields are declared in the file is a reasonable choice.

This .proto file is "compiled" into a class with the same name as the message (`HelloWorldMsg`). This class is accessed by including a header file with the same name as the .proto file, but with ".proto" replaced with ".pb.h". Furthermore, we can set the contents of this class using calls ("mutators" or "setters") that are the same as the field name (i.e. 'telegram' or 'count') prepended with "set_":

```
// C++
#include "hello_world.pb.h"

// create and populate a ``HelloWorldMsg'' called `msg'
HelloWorldMsg msg;
msg.set_telegram("hello world");
msg.set_count(3);
```

and access them using these methods ("accessors" or "getters") that have the same function name as the field name:

```
// C++
// print information about `msg' to the screen
std::cout << msg.telegram() << ": " << msg.count() << std::endl;
```

2.3 Learning how to *publish*: HelloWorld1

To create a Goby application, one needs to

- create a derived class of `goby::core::ApplicationBase`. We also must include the goby core header (`#include "goby/core/core.h"`).

- run the application using the `goby::run()` function. Because goby::core::ApplicationBase reads our configuration (including command line options) for us, we also pass `argv` and `argc` to `run()`.

That is all one needs to create a valid working Goby application. All together the "bare-bones" Goby application looks like:

```
#include "goby/core/core.h"

class HelloWorld1 : public goby::core::ApplicationBase {};

int main(int argc, char* argv[])
{
    return goby::run<HelloWorld1>(argc, argv);
}
```

However, we would like our application to do a little bit more.

ApplicationBase provides a virtual method called `loop()` that is called on some regular interval (it is the *synchronous event* in Goby), by default 10 Hertz. By overloading `loop()` in our derived class `HelloWorld1`, we can do any kind of synchronous work that needs to be done without tying up the CPU all the time[3]. In this example, we will create a simple message (of type HelloWorldMsg which we previously designed in chapter 2.2) and publish it to `gobyd` and thus all subscribers (we create a subscriber in chapter 2.4).

Let's walk through each line of our `loop()` method:

```
1 void loop()
2 {
3    static int i = 0;
4    HelloWorldMsg msg;
5    msg.set_telegram("hello world!");
6    msg.set_count(++i);
7    glogger() << "sending: " << msg << std::endl;
8    publish(msg);
9 }
```

Line 1: `loop()` takes no arguments and returns nothing (void). We declare (line 3) a static integer[4] to keep track of how many times we have looped and thus print

[3] in between calls to `loop()`, ApplicationBase handles incoming subscribed messages

[4] static in this context means that the variable will keep its value across calls to the function `loop()`.

an increasing integer value. Then we create a HelloWorldMsg called msg (line 4) and set the values of its fields (lines 5 and 6). We then publish a human debugging log message using `glogger()` (just like std::cout or other std::ostreams), which will be put to the terminal window in verbose mode[5]. Finally, we publish our message (line 8). The entirety of the code for `hello_world1_g` is listed in section 2.7.2.

2.4 Learning how to *subscribe*: HelloWorld2

Now that our `hello_world1_g` application is publishing a message, we would like to create an application that subscribes for it. To subscribe for a message, we typically provide two things:

- The type of the message we want to subscribe for (e.g. `HelloWorldMsg`).
- A method or function that should be called when we receive that type (a callback).

Subscriptions typically take place in the constructor (here, `HelloWorld2::HelloWorld2()`), but can happen at any time as needed (within `loop()`, for example). You subscribe for a type once, and then you will continue to receive all other applications' publishes to that type.

We subscribe for a type using a call to `subscribe()` that looks like this:

```
subscribe<HelloWorldMsg>(&HelloWorld2::receive_msg, this);
```

While a bit complicated at first, this call should make sense shortly. It reads "`subscribe` for all messages of type `HelloWorldMsg` and when you receive one, call the method `HelloWorld2::receive_msg` which is a member of `this` class (`HelloWorld2`)."[6]. The method provided as a callback (here `receive_msg()`) must have the signature

```
void func(const ProtoBufMessage&);
```

where `ProtoBufMessage` is the type subscribed for (here, `HelloWorldMsg`). `receive_msg()` has that signature

[5] goby provides operator« for google::protobuf::Message objects as a wrapper for google::protobuf::Message::DebugString()

[6] You can call a member function (method) of another class by passing the pointer to the desired class instantiation instead of `this`. Alternatively, you can call a non-class function by just giving its pointer, e.g. `subscribe(&receive_msg)`.

```
void HelloWorld2::receive_msg(const HelloWorldMsg& msg);
```

and thus is a valid callback for this subscription. After subscribing, `receive_msg()` will be called immediately (an asynchronous event) upon receipt of a message of type `HelloWorldMsg` unless

- `loop()` is in the process of being called or
- another message callback is in the process of being called.

In these cases, `receive_msg()` is called as soon as the blocking method returns. For this example, inside of `receive_msg()` we simply post the message to the debug log:

```
void receive_msg(const HelloWorldMsg& msg)
{
   glogger() << "received: " << msg << std::endl;
}
```

The full source listing for `hello_world2_g` can be found in section 2.7.3.

2.5 Compiling our applications using CMake

CMake [12], while still lacking in documentation, is probably the easiest way to build software these days, especially for cross platform support. I will briefly walk through building a Goby application using CMake within the larger Goby project configuration. If you look at the `CMakeLists.txt` file in 2.7.4, you can see the steps needed to add our new applications to the project:

```
1 protobuf_generate_cpp(PROTO_SRCS PROTO_HDRS hello_world.proto)
2 add_executable(hello_world1_g hello_world1.cpp ${PROTO_SRCS} ${PROTO_HDRS})
3 target_link_libraries(hello_world1_g goby_core)
```

Line 1 tells CMake to add "hello_world.proto" to the files needed to be precompiled by the Google Protocol Buffers compiler `protoc`. protobuf_generate_cpp is provided by the CMake module goby/cmake_modules/FindProtobufGoby.cmake. Line 2 adds our application `hello_world1_g` to the list to be compiled by the C++ compiler, using the sources `hello_world1.cpp` and the generated Protocol Buffers code. We append "_g" as a convention to quickly recognize Goby applications. Line 3 links our application against the goby_core library, which provides goby::core::ApplicationBase, our base class.

Adding `hello_world2_g` is directly analogous.

2.6 Trying it all out: running from the command line

Now, assuming you've compiled everything, we can run the example.

You'll need three terminal windows, one for `gobyd`, and one for each of our "hello world" applications. You need to start `gobyd` first

```
> gobyd -p hello_auv
```

I've gone ahead and named this platform "hello_auv". The platform name is a unique identifier for both intra- and inter-vehicle communications in Goby. Now we can launch our two applications (order doesn't matter), with the added "-v" flag to indicate we want verbose terminal output:

```
> hello_world1_g -p hello_auv -v
> hello_world2_g -p hello_auv -v
```

You should see `hello_world1_g` passing messages to `hello_world2_g` every 1/10th second.

2.7 Code

This entire example can be browsed online at http://bazaar.launchpad.net/~goby-dev/goby/1.0/files/head:/share/examples/core/ex1_hello_world.

2.7.1 goby/share/examples/core/ex1_hello_world/hello_world.proto

```
// see http://code.google.com/apis/protocolbuffers/docs/cpptutorial.html
// http://code.google.com/apis/protocolbuffers/docs/proto.html

message HelloWorldMsg
{
  required string telegram = 1;
  required uint32 count = 2;
}

```

2.7.2 goby/share/examples/core/ex1_hello_world/hello_world1.cpp

```
// for goby::core::ApplicationBase
#include "goby/core/core.h"

// autogenerated Protocol Buffers header
#include "hello_world.pb.h"

// allows us to directly output protobuf messages to streams
using goby::core::operator<<;

// create our Goby Application with ApplicationBase as a public base
class HelloWorld1 : public goby::core::ApplicationBase
{
private:
    // loop() is a virtual method of ApplicationBase that is called
    // at 10 Hz (by default)
    void loop()
        {
            static int i = 0;
            // create a message of type HelloWorldMsg (defined in
            // hello_world.proto)
            HelloWorldMsg msg;
            // set the fields we need
            msg.set_telegram("hello world!");
            msg.set_count(++i);

            glogger() << "sending: " << msg << std::endl;

            // publish it to `gobyd` who will send to all subscribers
            publish(msg);
        }
};

int main(int argc, char* argv[])
{
    // start up our application (ApplicationBase will read argc and
    // argv for us)
    return goby::run<HelloWorld1>(argc, argv);
}
```

2.7.3 goby/share/examples/core/ex1_hello_world/hello_world2.cpp

```
#include "goby/core/core.h"
#include "hello_world.pb.h"
```

```
using goby::core::operator<<;

class HelloWorld2 : public goby::core::ApplicationBase
{
public:
    HelloWorld2()
        {
            // subscribe for all messages of type HelloWorldMsg
            subscribe<HelloWorldMsg>(&HelloWorld2::receive_msg, this);
        }

private:
    void receive_msg(const HelloWorldMsg& msg)
        {
            // print to the log the newest received "HelloWorldMsg"
            glogger() << "received: " << msg << std::endl;
        }
};

int main(int argc, char* argv[])
{
    return goby::run<HelloWorld2>(argc, argv);
}
```

2.7.4 goby/share/examples/core/ex1_hello_world/CMakeLists.txt

```
# tells CMake to generate the *.pb.h and *.pb.cc files from the *.proto
protobuf_generate_cpp(PROTO_SRCS PROTO_HDRS hello_world.proto)

# add these executables to the project
add_executable(hello_world1_g hello_world1.cpp ${PROTO_SRCS} ${PROTO_HDRS})
add_executable(hello_world2_g hello_world2.cpp ${PROTO_SRCS} ${PROTO_HDRS})

# and link in the goby_core library
target_link_libraries(hello_world1_g goby_core)
target_link_libraries(hello_world2_g goby_core)
```

3 The GPS Driver example

Marine robots need to know where they are. The simplest way now is to use a GPS receiver. While this works only when the robot is on the surface of the ocean, it is one of the most accurate forms of positioning available and thus used as a starting point for undersea dead reckoning using Doppler Velocity Loggers (DVLs) or Inertial Measurement Units (IMUs). Therefore, reading a GPS receiver's output into a usable form for decision making is a useful and necessary ability for our marine robot. This example shows how we might do this using Goby.

Typically we might also need to know the depth of our vehicle. This is often determined by measuring the ambient pressure. In this example, we will simulate the scalar depth reading of such a pressure sensor.

Finally, it is often useful to have an aggregate of the vehicle's status that includes a snapshot of the vehicle's location, orientation, speed, heading, and perhaps other factors such as battery life and health. For this example, we call such a message a `NodeReport` and provide an application `node_reporter_g` that compiles the reports from the GPS and the depth sensor into a single message. To extend this example, we could add data from other sources, such as an inertial measurement unit (IMU) or Doppler Velocity Logger (DVL).

As the first example, the files for this example are located at the end of the chapter in section 3.6.

3.1 Reading *configuration* from files and command line: DepthSimulator

"DepthSimulator" is reads a starting depth value from a configuration file and reports that value as the current depth, perturbed slightly by a random value. It's a primitive constant depth simulator, but allows us to illustrate another feature of Goby, the configuration file reader.

Goby reads configuration text files and the command line using protobuf, in a similar manner messages are defined for passing between applications. The Goby application author provides a .proto file containing a protobuf message that defines all possible valid configuration values for the given application in the form of a protobuf message. Then the application instantiates a copy of this configuration message and passes it to the `goby::core::ApplicationBase` constructor with reads the configuration text file and/or command line options. If the configuration text file and/or command line options properly populate the provided proper configuration protobuf message, the message is returned to the derived class (the Goby application). Otherwise, execution of the application ends with a useful error message for the user explaining the errors involved with the passed configuration.

Thus, for the `DepthSimulator` we define a protobuf message called `DepthSimulatorConfig`:

```
message DepthSimulatorConfig
{
  required AppBaseConfig base = 1;
  required double depth = 2;
}
```

An embedded message of type `AppBaseConfig` is always provided for configuring parameters common for all Goby applications, such as the frequency that the virtual method `loop()` is called, the name of the application to use with `gobyd` (if different from the compiled name), and the name of the platform that this application belongs on (and thus which `gobyd` to connect to if multiple `gobyd`s are running on a single computer). The `AppBaseConfig` message is defined in goby/src/core/proto/app_base_config.proto.

Specifically, for our `DepthSimulator`, we only have one other configuration parameter, a double called ‘depth’. It is required, so our application will fail to run without a depth provided.

To use the Goby configuration reader, we create an instantation of our `DepthSimulatorConfig`

```
class DepthSimulator : public goby::core::ApplicationBase
{
...
    static DepthSimulatorConfig cfg_;
};
```

which must either be a global object or a static member of our class[1].

Then, all we must do is pass a pointer to that object to the constructor of the base class:

```
    DepthSimulator()
        : goby::core::ApplicationBase(&cfg_)
```

`goby::core::ApplicationBase` will take of the rest. To see what configuration values can be used in our compiled `depth_simulator_g`, we can run it with the `-h` (or equivalently, `--help`) flag:

[1]The configuration object must be a static member so that it is instantiated *before* the `goby::core::ApplicationBase` since normal members of our `DepthSimulator` class would be instantiated *after* `ApplicationBase`, which would lead to trouble when `ApplicationBase` tried to use the object

```
> depth_simulator_g --help
```

which should provides output

```
Allowed options:

Typically given in depth_simulator_g configuration file,
but may be specified on the command line:
  --base arg             (req)
                          platform_name: "AUV-23"  same as self.name for
                                                   gobyd cfg (req)
                          app_name: "myapp_g"  default is compiled name -
                                               change this to run multiple
                                               instances (opt)
                          verbosity: QUIET  Terminal verbosity
                                            (opt) (default)
                          loop_freq: 10  the frequency (Hz) used to run
                                         loop() (opt) (default)
  --depth arg            (req)

Given on command line only:
  -c [ --cfg_path ] arg      path to depth_simulator_g configuration file
                             (typically depth_simulator_g.cfg)
  -h [ --help ]              writes this help message
  -p [ --platform_name ] arg name of this platform (same as gobyd configuration
                             value `self.name`)
  -a [ --app_name ] arg      name to use when connecting to gobyd (default:
                             depth_simulator_g)
  -v [ --verbose ] arg       output useful information to std::cout. -v is
                             verbosity: verbose, -vv is verbosity: debug, -vvv
                             is verbosity: gui
```

Thus, to configure `depth_simulator_g` I could create a text file (let's say depth_simulator.cfg) with values like

```
# depth_simulator.cfg
base
{
    platform_name: "AUV-1"
    loop_freq: 1
}

depth: 10.4
```

Then, when we run `depth_simulator_g` we pass the path to the configuration file as the first command line option:

```
> depth_simulator_g depth_simulator.cfg
```

If we didn't want to use a configuration file, we could pass the same contents of the depth_simulator.cfg file given above on the command line instead:

```
> depth_simulator_g --base 'platform_name: "AUV-1" loop_freq: 1' --depth 10.4
```

If the same configuration values are provided in both the configuration file and on the command line, they are merged for "repeat" fields. For "required" or "optional" fields, the command line value overwrites the configuration file value.

Thus, if we run

```
> depth_simulator_g depth_simulator.cfg --depth 20.5
```

`cfg_.depth()` is 20.5 since the command line provided value takes precedence.

Some commonly used configuration values have shortcuts for the command line. For example, the following two commands are equivalent ways to set the platform name:

```
> depth_simulator_g --base 'platform_name: "AUV-1"'
> depth_simulator_g -p "AUV-1"
```

Other than reading a configuration file, all `DepthSimulator` does is repeatedly write a message of type `DepthReading` (see section 3.6.2) based off a random offset to the configuration value "depth":

```
void loop()
    {
       DepthReading reading;
       // just post the depth given in the configuration file plus a small random offset
       reading.set_depth(cfg_.depth() + (rand() % 10) / 10.0);

       glogger() << reading << std::flush;
       publish(reading);
    }
```

You will note that depth_reading.proto contains an import command and a field of type 'Header':

```
import "goby/core/proto/header.proto";

message DepthReading
{
  // time is in header
  required Header header = 1;
  required double depth = 2;
}
```

'Header' (defined in goby/src/core/proto/header.proto) provides commonly used fields such as time and source / destination addressing. It is highly recommended to include this in messages sent through Goby, but not required. `goby::core::ApplicationBase` will populate any required fields in 'Header' not given by DepthSimulator. For example, if neither time field ('unix_time' nor 'iso_time') is set, `goby::core::ApplicationBase` will set the time based on the time `publish()` was called. However either 'unix_time' or 'iso_time' should be set if the calling application has a better time stamp for the message than the publish time.

3.2 Our first *useful* application: `GPSDriver`

`GPSDriver` doesn't introduce any new features of Goby, but it attempts to be the first non-trivial application we have seen thus far. `GPSDriver` connects to a NMEA-0183 compatible GPS receiver over a serial port, reads all the messages and parses the GGA sentence into a useful protobuf message for posting to the database (via `gobyd`).

3.2.1 Configuration

The configuration needed for `GPSDriver` all pertains to how the serial GPS receiver is connected and how it communicates:

```
message GPSDriverConfig
{
  required AppBaseConfig base = 1;

  required string serial_port = 2;
  optional uint32 serial_baud = 3 [default = 4800];
  optional string end_line = 4 [default = "\r\n"];
}
```

Note the use of defaults when they are meaningful (the NMEA-0183 specification requires carriage return (`\r`) and new line (`\n`) to signify the end of a line so this default will likely often be precisely what our users want, saving them the effort of specifying it every time).

3.2.2 Protobuf Messages

GPSDriver uses two protobuf messages both defined in gps_nmea.proto (see section 3.6.7). The first (`NMEASentence`) is a parsed version of a generic NMEA-0183 message. The second (`GPSSentenceGGA`) contains a `NMEASentence` but also the parsed fields of the GGA position message. Providing the `GPSSentenceGGA` gives all subscribers of this message rapid access to useful data without parsing the original NMEA string again.

3.2.3 Body

`GPSDriver` should be straightforward to understand given what we have learned to this point. It makes use of some utilities in the goby::util libraries, especially the `goby::util::SerialClient` used for reading the serial port. These utilities are documented along with all the other Goby classes at `http://gobysoft.com/doc`.

Goby makes heavy use of the Boost libraries (`http://www.boost.org`). While you are not required to use any of Boost when developing Goby applications, it would be worth your while becoming acquainted with them. For example, the Boost Date-Time library gives a handy object oriented way to handle dates and times that far exceeds the abilities of ctime (i.e. time.h).

3.3 Subscribing for *multiple types*: NodeReporter

`NodeReporter` subscribes to both the output of `DepthSimulator` (`DepthReading`) and `GPSDriver` (`GPSSentenceGGA`). Whenever either is published, a new `NodeReport` message is created as the aggregate of pieces of both messages. The `NodeReport` (defined in node_report.proto in section 3.6.4) is a useful summation of the status of a given node (synonomously, platform). Because `DepthReading` and `GPSSentenceGGA` are published asynchronously, we also keep track of the delays between different parts of the NodeReport message (the `*_lag` fields).

The `NodeReport` provides

1. Name of the platform

2. Type of the platform (e.g. AUV, buoy)

3. The global position of the vehicle in geodetic coordinates (latitude, longitude, depth)
4. The local position of the vehicle in a local cartesian coordinate system (x, y, z) based off the datum defined in the configuration for `gobyd`. This is generally more useful for vehicle operators than the global fix.
5. The Euler angles of the current vehicle pose: roll, pitch, yaw (heading).
6. The speed of the vehicle.

In this example, we only set the first three fields given above. The others would require further sensing capability than we have in this example.

3.4 Putting it all together

First, we either need a real GPS unit or simulate one somehow. If you have a real NMEA-0183 GPS handy, by all means use it. Otherwise, I've made a fake GPS using `socat` and a log file of a real GPS (nmea.txt). This fake GPS can be run using

```
./fake_gps.sh nmea.txt
```

which writes a line from nmea.txt every second to the fake serial port `/tmp/ttyFAKE`. This should be good enough for us here. If you don't have `socat`, you should be able to find it in the package manager for your Linux distribution (`sudo apt-get install socat` in Debian or Ubuntu).

Next we need to launch everything. The list is beginning to grow

```
./fake_gps.sh nmea.txt
gobyd gobyd.cfg -v
./gps_driver_g gps_driver_g.cfg -v
./depth_simulator_g depth_simulator_g.cfg -v
./node_reporter_g node_reporter_g.cfg -v
```

but fortunately we've provided a script that launches everything for you in separate terminal windows. So all you need to do is type

```
./launch.sh
```

and enjoy the magic unfold. Should you wish to modify how things are launched, just edit launch_list.txt in goby/share/examples/core/ex2_gps_driver.

3.5 Reading the log files (SQLite3)

You may have noticed that everytime you run `gobyd` it creates a log file called `AUV-1_YYYYMMDDTHHMMSS_goby.db`. This is an SQLite3 [5] SQL database. Every variable published in Goby is written to this database. To read it, you need a tool capable of reading SQLite3 databases. One candidate is the `sqlite3` command line tool. The following will dump to your screen all the DepthReading values recorded. Using the interactive mode:

```
sqlite3 AUV-1_20110304T212549_goby.db
sqlite> .mode column
sqlite> .headers ON
sqlite> SELECT * FROM DepthReading;
```

or similarly on the command line only

```
sqlite3 -header -column AUV-1_20110304T212549_goby.db "SELECT * FROM DepthReading"
```

If a Graphical User Interface (GUI) is more your style, http://www.sqlite.org/cvstrac/wiki?p=ManagementTools has a whole list. My preference is Sqliteman, accessible in Ubuntu with `sudo apt-get install sqliteman`. Then it's just a matter of loading up the database and away you go:

```
sqliteman AUV-1_20110304T212549_goby.db
```

3.6 Code

This entire example can be browsed online at http://bazaar.launchpad.net/~goby-dev/goby/1.0/files/head:/share/examples/core/ex2_gps_driver.

3.6.1 goby/share/examples/core/ex2_gps_driver/config.proto

```
import "goby/protobuf/option_extensions.proto";
import "goby/protobuf/app_base_config.proto";

message GPSDriverConfig
{
  required AppBaseConfig base = 1;
```

```

  required string serial_port = 2;
  optional uint32 serial_baud = 3 [default = 4800];
  optional string end_line = 4 [default = "\r\n"];
}

message NodeReporterConfig
{
  required AppBaseConfig base = 1;
}

message DepthSimulatorConfig
{
  required AppBaseConfig base = 1;
  required double depth = 2;
}
```

3.6.2 goby/share/examples/core/ex2_gps_driver/depth_reading.proto

```
import "goby/protobuf/header.proto";

message DepthReading
{
  // time is in header
  required Header header = 1;
  required double depth = 2;
}
```

3.6.3 goby/share/examples/core/ex2_gps_driver/depth_simulator.cpp

```
#include <cstdlib> // for rand

#include "goby/core/core.h"

#include "config.pb.h"
#include "depth_reading.pb.h"

using goby::core::operator<<;

class DepthSimulator : public goby::core::ApplicationBase
{
public:
```

```
    DepthSimulator()
        : goby::core::ApplicationBase(&cfg_)
        { }

    void loop()
        {
            DepthReading reading;
            // just post the depth given in the configuration file plus a
            // small random offset
            reading.set_depth(cfg_.depth() + (rand() % 10) / 10.0);

            glogger() << reading << std::flush;
            publish(reading);
        }

    static DepthSimulatorConfig cfg_;
};

DepthSimulatorConfig DepthSimulator::cfg_;

int main(int argc, char* argv[])
{
    return goby::run<DepthSimulator>(argc, argv);
}

```

3.6.4 goby/share/examples/core/ex2_gps_driver/node_report.proto

```
import "goby/protobuf/header.proto";
import "goby/protobuf/app_base_config.proto";
import "goby/protobuf/config.proto";

message NodeReport
{
  required Header header = 1;
  required string name = 2;

  // defined in goby/core/proto/config.proto
  required goby.core.proto.VehicleType type = 3;

  // lat, lon, depth
  required GeodeticCoordinate global_fix = 4;
  // x, y, z on local cartesian grid
  optional CartesianCoordinate local_fix = 5;
```

```

  // roll, pitch, yaw
  optional EulerAngles pose = 7;

  // speed over ground (not relative to water or surface)
  optional double speed = 8;
  optional SourceSensor speed_source = 9;
  optional double speed_time_lag = 11;

}

enum SourceSensor { GPS = 1;
                    DEAD_RECKONING = 2;
                    INERTIAL_MEASUREMENT_UNIT = 3;
                    PRESSURE_SENSOR = 4;
                    COMPASS = 5;
                    SIMULATION = 6;}

message GeodeticCoordinate
{
  required double lat = 1;
  required double lon = 2;
  optional double depth = 3 [default = 0]; // negative of "height"
  optional double altitude = 4;

  optional SourceSensor lat_source = 5;
  optional SourceSensor lon_source = 6;
  optional SourceSensor depth_source = 7;
  optional SourceSensor altitude_source = 8;

  // time lags (in seconds) from the message Header time
  optional double lat_time_lag = 9;
  optional double lon_time_lag = 10;
  optional double depth_time_lag = 11;
  optional double altitude_time_lag = 12;
}

// computed from GeodeticCoordinate
message CartesianCoordinate
{
  required double x = 1;
  required double y = 2;
  optional double z = 3 [default = 0]; // negative of "depth"
}

// all in degrees
```

```
message EulerAngles
{
  optional double roll = 1;
  optional double pitch = 2;
  optional double yaw = 3; // also known as "heading"

  optional SourceSensor roll_source = 4;
  optional SourceSensor pitch_source = 5;
  optional SourceSensor yaw_source = 6;

  // time lags (in seconds) from the message Header time
  optional double roll_time_lag = 7;
  optional double pitch_time_lag = 8;
  optional double yaw_time_lag = 9;
}

```

3.6.5 goby/share/examples/core/ex2_gps_driver/node_reporter.h

```
#ifndef NODEREPORTER20101225H
#define NODEREPORTER20101225H

#include "goby/core/core.h"
#include "config.pb.h"

#include "gps_nmea.pb.h"
#include "depth_reading.pb.h"

class NodeReporter : public goby::core::ApplicationBase
{
public:
    NodeReporter();
    ~NodeReporter();


private:
    void create_node_report(const GPSSentenceGGA& gga,
                            const DepthReading& depth);

    void handle_depth(const DepthReading& reading)
    {
        create_node_report(newest<GPSSentenceGGA>(), reading);
    }

```

```
    void handle_gps(const GPSSentenceGGA& gga)
    {
        create_node_report(gga, newest<DepthReading>());
    }

    static NodeReporterConfig cfg_;
};

#endif
```

3.6.6 goby/share/examples/core/ex2_gps_driver/node_reporter.cpp

```

#include "node_reporter.h"

#include "node_report.pb.h"

using goby::core::operator<<;

NodeReporterConfig NodeReporter::cfg_;

int main(int argc, char* argv[])
{
    return goby::run<NodeReporter>(argc, argv);
}

NodeReporter::NodeReporter()
    : goby::core::ApplicationBase(&cfg_)
{
    // from Pressure Sensor Simulator
    subscribe<DepthReading>(&NodeReporter::handle_depth, this);

    // from GPS Driver
    subscribe<GPSSentenceGGA>(&NodeReporter::handle_gps, this);
}

NodeReporter::~NodeReporter()
{ }


void NodeReporter::create_node_report(const GPSSentenceGGA& gga,
                                      const DepthReading& depth_reading)
{
    if(!(gga.IsInitialized() && depth_reading.IsInitialized()))
```

```
    {
        glogger() << warn << "need both GPSSentenceGGA and DepthReading "
                  << "message to proceed" << std::endl;
        return;
    }

    glogger() << gga << depth_reading << std::flush;


    // make an abstracted position and pose aggregate from the newest
    // readings for consumption by other processes
    NodeReport report;

    // use the time from the GGA message as the base message time
    report.mutable_header()->set_iso_time(gga.header().iso_time());
    report.set_name(cfg_.base().platform_name());
    report.set_type(global_cfg().self().type());

    GeodeticCoordinate* global_fix = report.mutable_global_fix();
    global_fix->set_lat(gga.lat());
    global_fix->set_lon(gga.lon());

    // we set message time from GPS GGA, so no lag
    global_fix->set_lat_time_lag(0);
    global_fix->set_lon_time_lag(0);

    global_fix->set_lat_source(GPS);
    global_fix->set_lon_source(GPS);

    // set the depth sensor data
    global_fix->set_depth(depth_reading.depth());
    global_fix->set_depth_source(SIMULATION);
    global_fix->set_depth_time_lag(gga.header().unix_time()
                                   -depth_reading.header().unix_time());

    // TODO(tes): compute the local coordinates

    // in a better world we would want data for altitude, speed and
    // Euler angles too!
    glogger() << report << std::flush;

    publish(report);

}
```

3.6.7 goby/share/examples/core/ex2_gps_driver/gps_nmea.proto

```
import "goby/protobuf/header.proto";

message NMEASentence
{
  // e.g. "GP"
  required string talker_id = 1;
  // e.g. "GGA"
  required string sentence_id = 2;
  // e.g. 71
  required uint32 checksum = 3;
  // e.g. part[0] = $GPGGA
  //      part[1] = 123519
  //      part[2] = 4807.038
  //      part[3] = N
  // and so on
  repeated string part = 4;
}

message GPSSentenceGGA
{
  // time is in header
  required Header header = 1;
  required NMEASentence nmea = 2;

  // decimal degrees
  required double lat = 3;
  required double lon = 4;

  enum FixQuality
  {
    INVALID = 0;
    GPS_FIX = 1;
    DGPS_FIX = 2;
    PPS_FIX = 3;
    REAL_TIME_KINEMATIC = 4;
    FLOAT_RTK = 5;
    ESTIMATED = 6;
    MANUAL_MODE = 7;
    SIMULATION_MODE = 8;
  }
  required FixQuality fix_quality = 5;
  required uint32 num_satellites = 6;
  required float horiz_dilution = 7;
  required double altitude = 8;
  required double geoid_height = 9;
```

```
}
```

3.6.8 goby/share/examples/core/ex2_gps_driver/gps_driver.h

```
#ifndef GPSDRIVER20101014H
#define GPSDRIVER20101014H

#include "goby/core/core.h"
#include "goby/util/time.h"
// for serial driver
#include "goby/util/linebasedcomms.h"
#include "config.pb.h"

// forward declare (from gps_nmea.proto)
class NMEASentence;
class GPSSentenceGGA;

class GPSDriver : public goby::core::ApplicationBase
{
public:
    GPSDriver();
    ~GPSDriver();


private:
    void loop();
    boost::posix_time::ptime nmea_time2ptime(const std::string& nmea_time);
    void string2nmea_sentence(std::string in, NMEASentence* out);
    void set_gga_specific_fields(GPSSentenceGGA* gga);

    goby::util::SerialClient serial_;
    static GPSDriverConfig cfg_;
};

// very simple exception classes
class bad_nmea_sentence : public std::runtime_error
{
  public:
  bad_nmea_sentence(const std::string& s)
      : std::runtime_error(s)
    { }
};

class bad_gga_sentence : public std::runtime_error
```

```
{
  public:
  bad_gga_sentence(const std::string& s)
      : std::runtime_error(s)
    { }
};


#endif
```

3.6.9 goby/share/examples/core/ex2_gps_driver/gps_driver.cpp

```
#include "gps_driver.h"

#include "gps_nmea.pb.h"

#include "goby/util/binary.h" // for goby::util::hex_string2number
#include "goby/util/string.h" // for goby::util::as

using goby::core::operator<<;

GPSDriverConfig GPSDriver::cfg_;

int main(int argc, char* argv[])
{
    return goby::run<GPSDriver>(argc, argv);
}

GPSDriver::GPSDriver()
    : goby::core::ApplicationBase(&cfg_),
      serial_(cfg_.serial_port(), cfg_.serial_baud(), cfg_.end_line())
{
    serial_.start();
}

GPSDriver::~GPSDriver()
{
    serial_.close();
}

void GPSDriver::loop()
{
    std::string in;
    while(serial_.readline(&in))
```

```
    {
        glogger() << "raw NMEA: " << in << std::flush;

        // parse
        NMEASentence nmea;
        try
        {
            string2nmea_sentence(in, &nmea);
        }
        catch (bad_nmea_sentence& e)
        {
            glogger() << warn << "bad NMEA sentence: " << e.what()
                      << std::endl;
        }

        if(nmea.sentence_id() == "GGA")
        {
            glogger() << "This is a GGA type message." << std::endl;

            // create the message we send on the wire
            GPSSentenceGGA gga;
            // copy the raw message (in case later users want to do their
            // own parsing)
            gga.mutable_nmea()->CopyFrom(nmea);

            try
            {
                set_gga_specific_fields(&gga);

                // parse the time stamp
                boost::posix_time::ptime t = nmea_time2ptime(nmea.part(1));
                gga.mutable_header()->set_iso_time(
                    boost::posix_time::to_iso_string(t));

                glogger() << gga << std::flush;

                publish(gga);
            }
            catch(bad_gga_sentence& e)
            {
                glogger() << warn << "bad GGA sentence: " << e.what()
                          << std::endl;
            }

        }

```

```
    }
}


// from http://www.gpsinformation.org/dale/nmea.htm#GGA
// GGA - essential fix data which provide 3D location and accuracy data.
// $GPGGA,123519,4807.038,N,01131.000,E,1,08,0.9,545.4,M,46.9,M,,*47
// Where:
//      GGA          Global Positioning System Fix Data
//      123519       Fix taken at 12:35:19 UTC
//      4807.038,N   Latitude 48 deg 07.038' N
//      01131.000,E  Longitude 11 deg 31.000' E
//      1            Fix quality: 0 = invalid
//                                1 = GPS fix (SPS)
//                                2 = DGPS fix
//                                3 = PPS fix
//            4 = Real Time Kinematic
//            5 = Float RTK
//                                6 = estimated (dead reckoning)
//                                    (2.3 feature)
//            7 = Manual input mode
//            8 = Simulation mode
//      08           Number of satellites being tracked
//      0.9          Horizontal dilution of position
//      545.4,M      Altitude, Meters, above mean sea level
//      46.9,M       Height of geoid (mean sea level) above WGS84
//                       ellipsoid
//      (empty field) time in seconds since last DGPS update
//      (empty field) DGPS station ID number
//      *47          the checksum data, always begins with *
// If the height of geoid is missing then the altitude should be suspect.
// Some non-standard implementations report altitude with respect to the
// ellipsoid rather than geoid altitude. Some units do not report negative
// altitudes at all. This is the only sentence that reports altitude.

void GPSDriver::set_gga_specific_fields(GPSSentenceGGA* gga)
{
    using goby::util::as;
    const NMEASentence& nmea = gga->nmea();

    const std::string& lat_string = nmea.part(2);

    if(lat_string.length() > 2)
    {
        double lat_deg = as<double>(lat_string.substr(0, 2));
        double lat_min = as<double>(lat_string.substr(2, lat_string.size()));
```

```
        double lat = lat_deg + lat_min / 60;
        gga->set_lat((nmea.part(3) == "S") ? -lat : lat);
    }
    else
    {
        throw(bad_gga_sentence("invalid latitude field"));
    }

    const std::string& lon_string = nmea.part(4);
    if(lon_string.length() > 2)
    {
        double lon_deg = as<double>(lon_string.substr(0, 3));
        double lon_min = as<double>(lon_string.substr(3, nmea.part(4).size()));
        double lon = lon_deg + lon_min / 60;
        gga->set_lon((nmea.part(5) == "W") ? -lon : lon);
    }
    else
        throw(bad_gga_sentence("invalid longitude field: " + nmea.part(4)));

    switch(goby::util::as<int>(nmea.part(6)))
    {
        default:
        case 0: gga->set_fix_quality(GPSSentenceGGA::INVALID); break;
        case 1: gga->set_fix_quality(GPSSentenceGGA::GPS_FIX); break;
        case 2: gga->set_fix_quality(GPSSentenceGGA::DGPS_FIX); break;
        case 3: gga->set_fix_quality(GPSSentenceGGA::PPS_FIX); break;
        case 4: gga->set_fix_quality(GPSSentenceGGA::REAL_TIME_KINEMATIC);
            break;
        case 5: gga->set_fix_quality(GPSSentenceGGA::FLOAT_RTK); break;
        case 6: gga->set_fix_quality(GPSSentenceGGA::ESTIMATED); break;
        case 7: gga->set_fix_quality(GPSSentenceGGA::MANUAL_MODE); break;
        case 8: gga->set_fix_quality(GPSSentenceGGA::SIMULATION_MODE); break;
    }

    gga->set_num_satellites(goby::util::as<int>(nmea.part(7)));
    gga->set_horiz_dilution(goby::util::as<float>(nmea.part(8)));
    gga->set_altitude(goby::util::as<double>(nmea.part(9)));
    gga->set_geoid_height(goby::util::as<double>(nmea.part(11)));
}

// converts a NMEA0183 sentence string into a class representation
void GPSDriver::string2nmea_sentence(std::string in, NMEASentence* out)
{

    // Silently drop leading/trailing whitespace if present.
    boost::trim(in);
```

```

    // Basic error checks ($, empty)
    if (in.empty())
      throw bad_nmea_sentence("message provided.");
    if (in[0] != '$')
        throw bad_nmea_sentence("no $: '" + in + "'.");
    // Check if the checksum exists and is correctly placed, and strip it.
    // If it's not correctly placed, we'll interpret it as part of message.
    // NMEA spec doesn't seem to say that * is forbidden elsewhere?
    // (should be)
    if (in.size() > 3 && in.at(in.size()-3) == '*') {
      std::string hex_csum = in.substr(in.size()-2);
      int cs;
      if(goby::util::hex_string2number(hex_csum, cs))
          out->set_checksum(cs);
      in = in.substr(0, in.size()-3);
    }

    // Split string into parts.
    size_t comma_pos = 0, last_comma_pos = 0;
    while((comma_pos = in.find(",", last_comma_pos)) != std::string::npos)
    {
        out->add_part(in.substr(last_comma_pos, comma_pos-last_comma_pos));

        // +1 moves us past the comma
        last_comma_pos = comma_pos + 1;
    }
    out->add_part(in.substr(last_comma_pos));

    // Validate talker size.
    if (out->part(0).size() != 6)
        throw bad_nmea_sentence("bad talker length '" + in + "'.");

    // GP
    out->set_talker_id(out->part(0).substr(1, 2));
    // GGA
    out->set_sentence_id(out->part(0).substr(3));

}


// converts the time stamp used by GPS messages of the format HHMMSS.SSS
// for arbitrary precision fractional
// seconds into a boost::ptime object (much more usable class
// representation of for dates and times)
// *CAVEAT* this assumes that the message was received "today" for the
```

```
// date part of the returned ptime.
boost::posix_time::ptime GPSDriver::nmea_time2ptime(const std::string& mt)
{
    using namespace boost::posix_time;
    using namespace boost::gregorian;

    // must be at least HHMMSS
    if(mt.length() < 6)
        return ptime(not_a_date_time);
    else
    {
        std::string s_hour = mt.substr(0,2), s_min = mt.substr(2,2),
            s_sec = mt.substr(4,2), s_fs = "0";

        // has some fractional seconds
        if(mt.length() > 7)
            s_fs = mt.substr(7); // everything after the "."

        try
        {
            int hour = boost::lexical_cast<int>(s_hour);
            int min = boost::lexical_cast<int>(s_min);
            int sec = boost::lexical_cast<int>(s_sec);
            int micro_sec = boost::lexical_cast<int>(s_fs)*
                pow(10, 6-s_fs.size());

            return (ptime(date(day_clock::universal_day()),
                          time_duration(hour, min, sec, 0)) +
                    microseconds(micro_sec));
        }
        catch (boost::bad_lexical_cast&)
        {
            return ptime(not_a_date_time);
        }
    }
}

```

3.6.10 goby/share/examples/core/ex2_gps_driver/gobyd.cfg

```
self
{
    name: "AUV-1"
    type: AUV
```

```
}
```

3.6.11 goby/share/examples/core/ex2_gps_driver/depth_simulator_g.cfg

```
base
{
    platform_name: "AUV-1"
    loop_freq: 1
}

depth: 10
```

3.6.12 goby/share/examples/core/ex2_gps_driver/gps_driver_g.cfg

```
base
{
    platform_name: "AUV-1"
    loop_freq: 1
}
serial_port: "/tmp/ttyFAKE"
```

3.6.13 goby/share/examples/core/ex2_gps_driver/node_reporter_g.cfg

```
base
{
    platform_name: "AUV-1"
    loop_freq: 0.5
}
```

3.6.14 goby/share/examples/core/ex2_gps_driver/nmea.txt

```
$GPRMC,183729,A,3907.356,N,12102.482,W,000.0,360.0,080301,015.5,E*6F
$GPRMB,A,,,,,,,,,,,,V*71
$GPGGA,183730,3907.356,N,12102.482,W,1,05,1.6,646.4,M,-24.1,M,,*75
$GPGSA,A,3,02,,,07,,09,24,26,,,,,1.6,1.6,1.0*3D
$GPGSV,2,1,08,02,43,088,38,04,42,145,00,05,11,291,00,07,60,043,35*71
$GPGSV,2,2,08,08,02,145,00,09,46,303,47,24,16,178,32,26,18,231,43*77
$PGRME,22.0,M,52.9,M,51.0,M*14
```

```
$GPGLL,3907.360,N,12102.481,W,183730,A*33
$PGRMZ,2062,f,3*2D
$PGRMM,WGS 84*06
$GPBOD,,T,,M,,*47
$GPRTE,1,1,c,0*07
$GPRMC,183731,A,3907.482,N,12102.436,W,000.0,360.0,080301,015.5,E*67
$GPRMB,A,,,,,,,,,,,,V*71
```

4 What's next

That's all for `goby-core` in Release 1.0. There's still a lot to do so keep tuned. If you want the bleeding edge, you can check out the Goby trunk branch with `bzr checkout lp:goby`.

Here's what's on the horizon:

- support for seamless inter-platform communications via acoustics (acomms), serial, wifi, and ethernet. Maybe even two cans and a string.
- a `Wt` [13] based configuration, launch, and runtime manager.

Stay tuned at `https://launchpad.net/goby`. Thanks.

A

Goby MOOS Modules

The acoustic communications portion of Goby was developed originally for the MOOS autonomy architecture. Thus, the relevant MOOS modules `pAcommsHandler` and others are still maintained (in goby/src/moos) for the use of the `MOOS-IvP` community. `MOOS-IvP` is explained in [14] and is available at `http://moos-ivp.org`. The usage of these modules is documented here. See `http://gobysoft.com/doc/#install` for how to install Goby.

The beginning of this appendix motivates the design, followed by a detailed user manual for the individual MOOS processes.

A.1 Unified Command and Control for Subsea Autonomous Sensing Networks

The process of undersea observation, mapping, and monitoring is experiencing a dramatic paradigm shift away from platform-centric, human-controlled sensing, processing and interpretation. Rather, distributed sensing using networks of autonomous platforms is becoming the preferred technique. An optimal platform suite is often highly heterogeneous with large differences in mobility, maneuverability, sensing capability, and communication connectivity. The sensor systems have different constraints on platform mobility and communication capacity, and some network operations require highly coordinated maneuvering of heterogeneous platforms. Unified Command and Control [15] is a new command and control paradigm inherently suited for such heterogeneous networks. Implemented using `MOOS-IvP`, Unified C2 provides the fully integrated sensing, modeling and control that allows each platform, on its own or in collaboration with partners of opportunity, to autonomously detect, classify, localize and track (DCLT) an episodic, natural or human-created event, and subsequently report back to the operators.

A robust undersea communication infrastructure is crucial to the operation of such networks. In contrast to air and land-based equivalents, the extremely limited bandwidth, latency and intermittency of underwater acoustic communication imposes severe requirements to the selectivity of message handling. Thus, contact and track reports for high-priority event, such as a detected chemical plume from a deep ocean vent, which may indicate an imminent volcanic eruption, must be transmitted to the system operators without delay. On the other hand, reports concerning less important events and platform status reports may be delayed without significant effects. Previous message handling systems for underwater communications have only a rigid, hard-coded queuing infrastructure, and do not support such advanced priority-based selectivity, hampering the type and amount of information

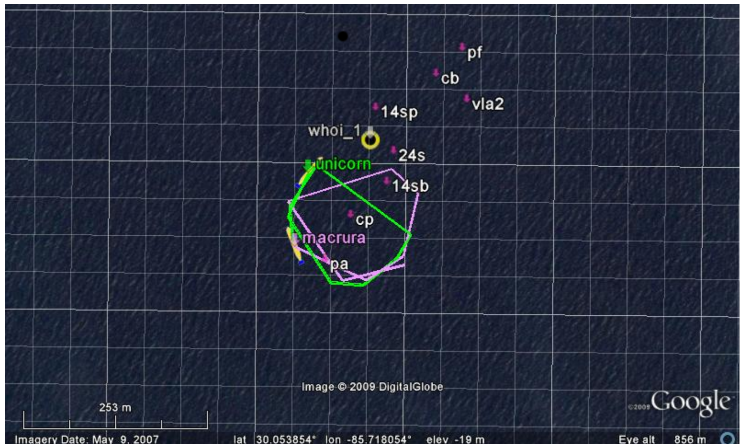

Figure A.1: Collaborative autonomy demonstrated in SWAMSI09 using MIT LAMSS communication stack. The two BF21 AUVs Unicorn and Macrura perform synchronized swimming maintaining a constant bistatic angle of 60° relative to a proud cylindrical target (cp).

that can be passed between cooperating nodes in the network. This severely limits the level of autonomy that can be supported on the network nodes.

In response to this problem, a new `MOOS-IvP` communication software stack was developed at the MIT Laboratory for Autonomous Marine Sensing Systems (LAMSS) [16], in support of autonomous sensing programs such as the ONR ASAP MURI, GOATS, and SWAMSI. This new stack has enabled the operation of a communication infrastructure which provides robust message handling for collaborative autonomous sensing by heterogeneous, undersea autonomous assets, as demonstrated in a handful of major recent field experiments. As an example, Fig. A.1 shows the collaborative, multistatic MCM mission by the Unicorn and Macrura BF21 AUVs during SWAMSI09 in Panama City, FL. The two vehicles are circling a proud cylinder (cp) at a distance of 80 m maintaining a constant bistatic angle of 60 degrees. The collaboration was achieved fully autonomously without any intervention by the operators, with each vehicle adapting its speed based on its current position and the position of the other vehicle extrapolated from the latest status, contact or track report. Such collaborative maneuvers would not be possible using traditional communication schemes, where navigation packets must be rigidly interleaved with messages containg data and command and control sequences. In contrast, the Dynamic Compact Control Language (DCCL) used by the LAMSS communication stack allows for adequate navigation information to be packed with all other required message content.

Being based on the established open source `goby-acomms` libraries of message handling software, the open source architecture of this new MOOS communication stack (embodied primarily in the MOOS application `pAcommsHandler` lends itself directly to a wide range of military and civilian applications. It supports an arbitrary message suite and content without requirement of modifying software. All message encoding and decoding information is specified in a mission-unique configuration file written in the standard XML format. Not only does this ensure maximum flexibility in regard to message design, but it inherently enables arbitrary levels of encryption for LPI/LPD communication networks.

A.2 Overview of the LAMSS Communication Stack

MIT LAMSS [16] has over the last decade focused its research on the development of sensor-adaptive, collaborative, autonomous sensing concepts for the capture of episodic undersea events, including the mapping of coastal fronts, chemical plumes, and natural and man-made underwater acoustic sources. All these applications involve the Detection, Classification, Localization and Tracking (DCLT) of the event. To exploit the benefits of having multiple platforms involved in tracking the event, an underwater robust communication system is obviously a requirement. On the other hand, the communication capacity of such systems is many orders of magnitude below land- and air-based equivalents, requiring a much higher level of data compression and on-board processing and decision-making than is required in air-based systems. Unified C2 [15], developed over the last decade by LAMSS, is an example of such an autonomy-driven undersea sensing concept. Although this concept is based on the philosophy that the system must be able to achieve its mission objective even during periods with no or limited communication, there is obviously still a need for occasional communication, e.g. for reporting detected events of interest.

The new `MOOS-IvP` communication stack alleviates some of the problems and limitations of the existing software stacks in this regard. These software stacks in general were designed to sequentially transmit all messages generated by the autonomy system, with only a rigid, hard-coded priority-based message queuing infrastructure.

In undersea autonomous systems the priorities of information generated by the on-board processing are highly dynamic, depending on the tactical situation and the criticality of the generated information. Thus, for example, a contact report for a target of interest obviously must bypass queued contact reports for less significant targets. Also, in high-clutter environments, the number of contact reports may by far exceed the communication capacity and on-board priority-based filtering is

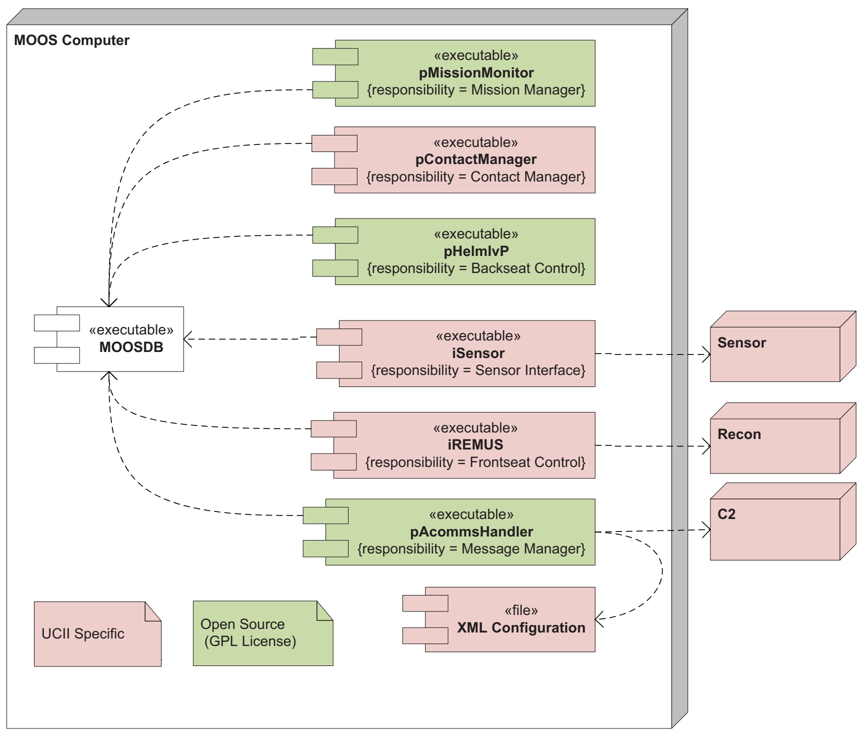

Figure A.2: Incorporation of the open source LAMSS communication stack into a MOOS-IvP DCLT Autonomy System. The green boxes identify the open source modules, including the IvP Helm, the generic mission manager module, and the communication stack. The red modules are project specific, including the frontseat driver module, and the sensor modules. Also the message configuration specifying the message content and the coding, is project specific.

required.

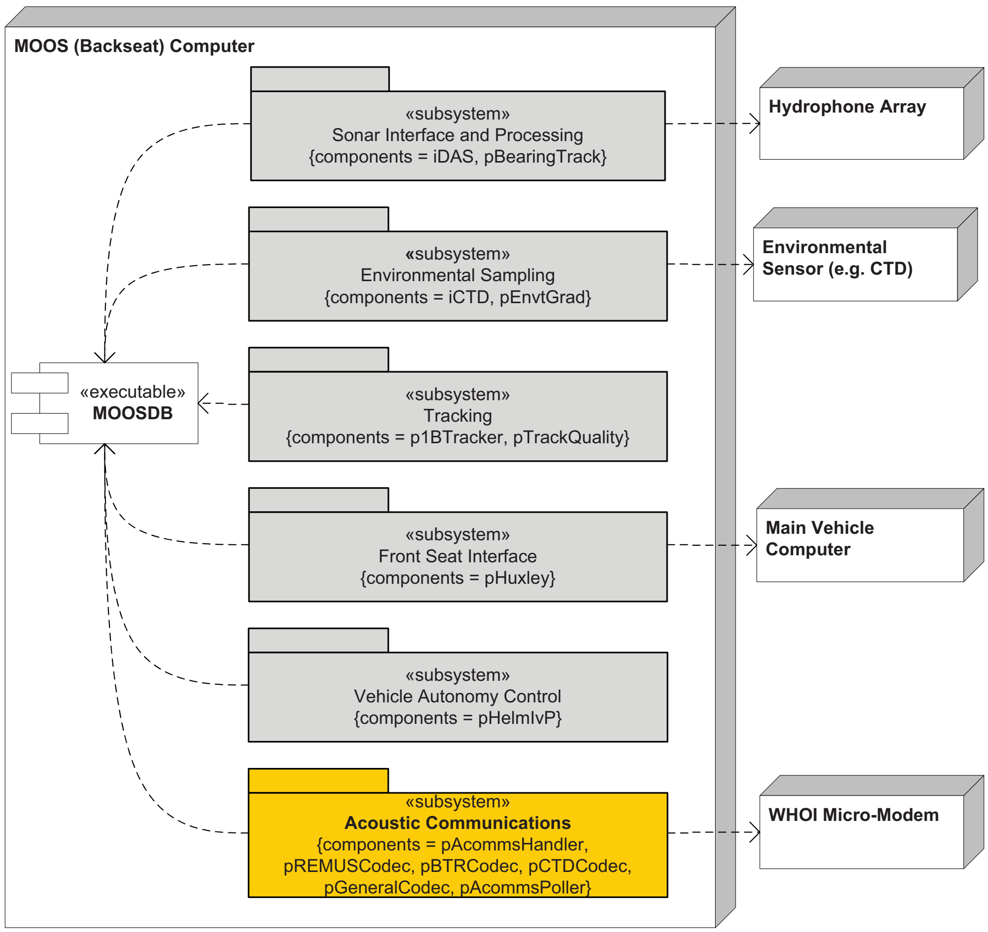

Figure A.3: `MOOS-IvP` community for MIT sonar AUVs, with the autonomous communication, command and control modules highlighted in gold.

The incorporation of the MIT LAMSS communication stack into a MOOS-IvP DCLT Autonomy System is illustrated in Fig. A.2. The green boxes identify the Open Source modules, including the helm `pHelmIvP`, the generic mission manager module `pMissionMonitor`, and the communication stack. The red modules are project-specific, including the frontseat driver module `iRemus`, the sensor modules, and the contact manager process `pContactManager`. Also the message configuration files specifying the message content and the coding specifics, are project-specific and not hard-wired into the communication stack.

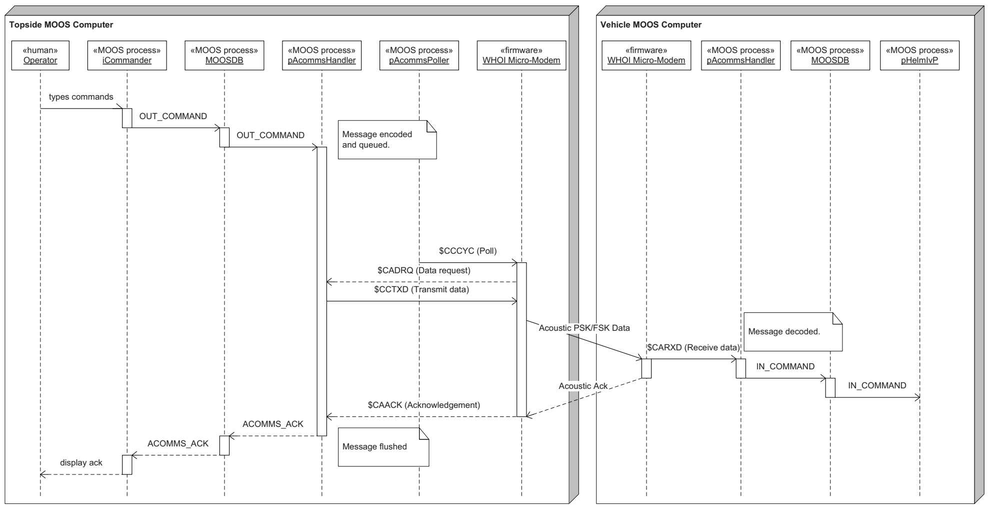

Figure A.4: UML Sequence diagram for sending a command to an AUV via the LAMSS Acoustic Communications Modules.

Figure A.3 shows the communications subsystem as part of the whole MIT LAMSS AUV MOOS community.

Figure A.4 shows the sequence of commands for a single operator command message sent using iCommander.

The structure of the MIT LAMSS communication stack is illustrated in Fig. A.5.

A.3 pAcommsHandler

A.3.1 Problem

Acoustic communications are highly limited in throughput. Thus, it is unreasonable to expect "total throughput" of all communications data. Furthermore, even if total throughput is achievable over time, certain messages have a lower tolerance for delay (e.g. vehicle status) than others (e.g. CTD sample data). Reference `http://acomms.whoi.edu/umodem/documentation.html` for more information on the WHOI Micro-Modem.

Also, in order to make the best use of this available bandwidth, messages need to be compacted to a minimal size before sending (effective encoding). To do this, pAcommsHandler provides an interface to the Dynamic Compact Control Language (DCCL[1].) encoder/decoder. Furthermore, DCCL has powerful parsing abilities ("al-

[1] the name comes from the original CCL written by Roger Stokey for the REMUS AUVs, but with the

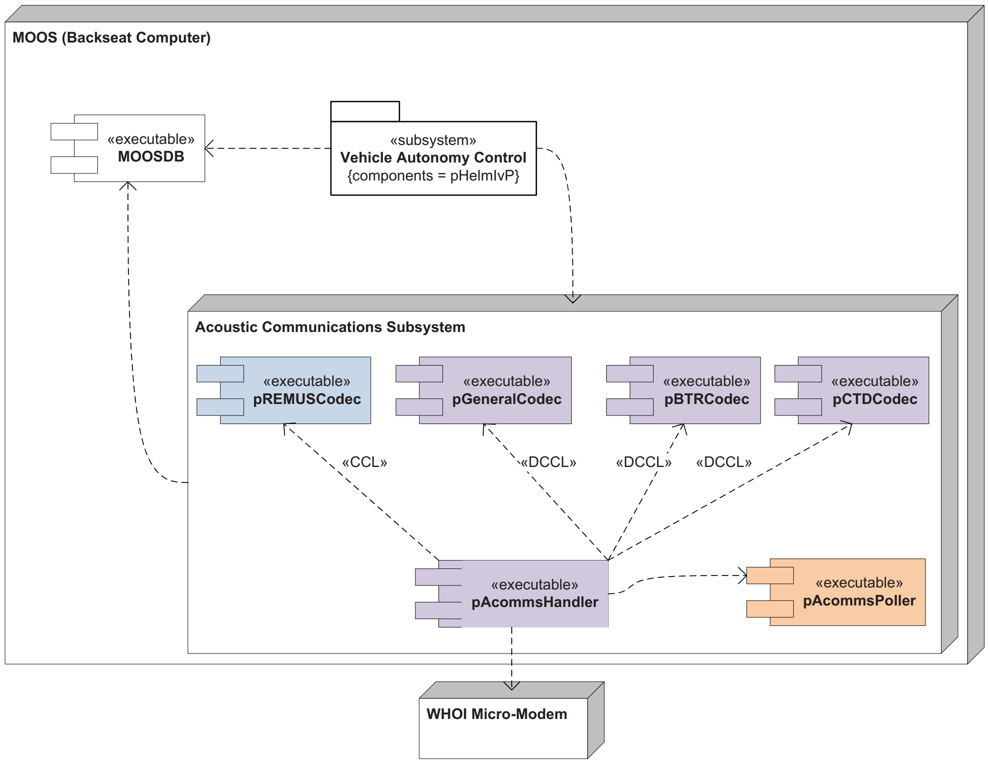

Figure A.5: UML Component Model of the MIT LAMSS communication stack. The principal message handler module is `pAcommsHandler`, which communicates directly with the modem using built-in drivers, and thus not dependent on third-party MOOS modem drivers. It also manages the message stream by a dynamic, priority-based queuing system. The message coding and decoding is performed by `pGeneralCodec` based on the rules set out in the configuration file, and dedicated DCCL codecs for transmitting various data streams.The stack also supports standard fixed Compact Control Language (CCL) messages such as the State message used by the Remus AUV, using dedicated codecs. Dashed line indicate dependencies between components.

gorithms") for both encoding and decoding, including the ability to perform certain geodesic conversions (e.g. latitude, longitude ↔ UTM x,y) and lookups (e.g. *modem_id* ↔ vehicle name) on data.

pAcommsHandler roughly performs the same functions of pFramer, pRouter, pAcommsPoller, and iMicroModem but generalized to handle any number of message queues and extended to give more control over queue parameters. The DCCL encoding is much more flexible and more compact than the CCL encoding used by these older processes.

A.3.2 Solution

pAcommsHandler provides a(n):

1. Encoder/decoder unit (codec): encodes and decodes messages using DCCL (goby-acomms `dccl` library), which reduces the data required to be sent by:
   - Predefined messages: the user must specify a message structure what specifies what fields the message contains and how large each field should be (in an intuitive fashion that DCCL turns into bits). Both the sender and receiver have preshared knowledge of the message structure. From this knowledge, no meta information about the message (beyond an identifier) needs to be sent, simply the data.
   - Custom field sizes: message fields are defined with custom tolerances (ranges and precisions) that are tighter than those given by the IEEE standards for floating point and integer numbers. For example, if a field needs to hold an integer that will never range outside [0, 1000] that field in the message will only be 10 bits long ($\text{ceil}(\log_2 1001)$).
2. Priority Queuing System: maintains an arbitrary number of message queues (each tied to a different MOOS variable) for hexadecimal data strings. (goby-acomms `queue` library)
   - allows configuration of the queue priorities and dynamic growth of the priority over the time since the last sent message.
   - allows management of WHOI CCL message types as well as DCCL queuing.
3. Modem Driver: handles all Micro-Modem serial communications. The driver (goby-acomms `modemdriver` library) can be used with other modems besides

ability to dynamically reconfigure messages based on mission need. DCCL is backwards compatible with a CCL network as it uses CCL message number 32 the WHOI Micro-Modem (see `http://gobysoft.com/doc/acomms__driver.html#acomms_writedriver` for information on writing a new driver).

4. MAC Manager: provides medium access control in the form of a simple slotted time division-multiple access (TDMA) scheme or flexible centralized polling (goby-acomms `amac` library).

A.3.3 Limitations

pAcommsHandler *does not*:

- presently provide any multi-hop routing. The sender and receiver must be directly connected acoustically.
- split user messages into packets. The user must provide data that are small enough to fit into the modem frame desired (32 - 256 bytes for the WHOI Micro-Modem).

A.3.4 Compilation

pAcommsHandler depends on the Goby and MOOS libraries. See goby/DEPENDENCIES for help resolving the dependencies on your system.

A.3.5 Parameters for the pAcommsHandler Configuration Block

Example moos file

You can always get a complete listing of MOOS file parameters with their syntax by running

```
> pAcommsHandler --example_config
```

This is a complete list of all the configuration values pAcommsHandler accepts. Most of the time you will need far fewer configuration options to use it.

Filling out the .moos file

Many of the parameters are sufficiently explained in the above list of configuration parameters. What follows is a detailed explanation of the parameters that need further explanation.

- `common`: Parameters that can be set for any of the Goby MOOS applications. Here you can control logging to a text file, terminal verbosity. You can also initialize a variable in the MOOS database at startup. Many of these parameters will automatically be set to a global MOOS variable (specified outside any ProcessConfig block) if left empty. For example, the global MOOS variable `LatOrigin` will set the pAcommsHandler variable `common::lat_origin`. This allows pAcommsHandler to conform to MOOS *de facto* conventions.

  – `verbosity`: choose `VERBOSITY_VERBOSE` for full text terminal output, `VERBOSITY_WARN` for warnings only, and `VERBOSITY_QUIET` for no terminal output. `VERBOSITY_GUI` opens an NCurses GUI helpful to debugging and visualizing the many data flows of pAcommsHandler.
  – `initializer`: since many times it is useful to have a MOOS variable including in a message that remains static for a given mission (vehicle name, etc), we give the option to publish initial MOOS variables here (for later use in messages [until overwritten, of course]). If `global_cfg_var` is set, pAcommsHandler looks for a global (i.e. specified at the top of the MOOS file or outside any `ProcessConfig` blocks) value in the .moos file with the name to the right of the colon and publishes it to a MOOS variable with the name to the left of the colon. For example:

    ```
    initializer { global_cfg_var: "LatOrigin" moos_var: "LAT_ORIGIN" }
    ```

    looks for a variable in the .moos file called `LatOrigin` and publishes it to the MOOSDB as a double variable `LAT_ORIGIN` with the value given by `LatOrigin`.
  – `log_path`: folder to log all terminal output to for later debugging. Similar to system logs in /var/log.
  – `log`: boolean to indicate whether to log terminal output or not to files in the path by `log_path`.

- `modem_id`: integer that specifies the `modem_id` of this current vehicle / community. For the WHOI Micro-Modem this is the Micro-Modem “SRC” configuration parameter (as set by `\$CCCFG,SRC,#` to check). For the remainder of the document, `modem_id` refers to the value `\$CCCFG,SRC,modem_id`. This configuration parameter will be set on startup. Setting this within the main block for pAcommsHandler sets it for all the modems (`driver_cfg`, `dccl_cfg`, `queue_cfg`, `mac_cfg`)
- `modem_id_lookup_path`: path to a text file giving the mapping between `modem_id` and vehicle name and type for a given experiment. This file should look like:

```
// modem id, vehicle name (should be community name), vehicle type
0, broadcast, broadcast
1, endeavor, ship
3, unicorn, auv
4, macrura, auv
```

Encoding/Decoding (DCCL) Parameters (`dccl_cfg`)

- `modem_id`: Will be set to the same as `ProcessConfig { modem_id: }` . There is no need to set it again here.

- `message_file`: `path` to an XML file containing a message set of one or messages. If you want, you can insert one or more manipulators that change the behavior of pAcommsHandler for messages defined in that file. Allowed manipulators:

  – `NO_MANIP`: blank manipulator (behavior is not modified by this manipulator)

  – `NO_ENCODE`: do not encode this message

  – `NO_DECODE`: do not decode this message

  – `NO_QUEUE`: do not queue this message

  – `LOOPBACK`: decode this message internally immediately following encode. Note that messages addressed to the local vehicle are looped back regardless of the value of this manipulator.

  – `ON_DEMAND`: encode immediately preceding a data request command (use for time sensitive messages like STATUS). This only works if all the message variables are always assumed fresh in the MOOSDB.

- `crypto_password`: optionally provide a password here to encrypt all communications using AES. All receiving nodes must have the same password.

Queuing Parameters (`queue_cfg`) All queue configuration for DCCL messges must be configured within the XML files `<queuing />` tag and included with `message_file: {path: "message.xml"}`. Any `message_file`s specified for `dccl_cfg` are copied to `queue_cfg` and vice-versa, so you don't need to specify them in two places.

CCL messages are configured using the `queue { }` object. The fields for `queue` correspond to the XML `<queuing />` tags:

- `id`: DCCL: a unique ID for this message (in the range 0-511). CCL: The decimal representation of the first byte of the CCL message to be queued.

- `ack`: boolean flag (1=true, 0=false) whether to request an acoustic acknowledgment on all sent messages from this field. If omitted, default of 0 (false, no ack) is used.

- `blackout_time`: time in seconds after sending a message from this queue for which no more messages will be sent. Use this field to stop an always full queue from hogging the channel. If omitted, default of 0 (no blackout) is used.

- `max_queue`: number of messages allowed in the queue before discarding messages. If `newest_first` is set to true, the oldest message in the queue is discarded to make room for the new message. Otherwise, any new messages are disregarded until the space in the queue opens up.

- `newest_first`: boolean flag (1=true=FILO, 0=false=FIFO) whether to send newest messages in the queue first (FILO) or not (FIFO).

- `ttl`: the time (in seconds) the message is allowed to live before being discarded. This also factors into the priority calculation as messages with a lower time-to-live (ttl) grow in priority faster.

- `value_base`: Each queue has a base value (V_{base}) and a time-to-live (ttl) that create the priority ($P(t)$) at any given time (t):

  $$P(t) = V_{base}\frac{(t - t_{last})}{ttl}$$

  where t_{last} is the time of the last send from this queue.

  This means for every queue, the user has control over two variables (V_{base} and ttl). V_{base} is intended to capture how important the message type is in general. Higher base values mean the message is of higher importance. The ttl governs the number of seconds the message lives from creation until it is destroyed by libqueue. The ttl also factors into the priority calculation since all things being equal (same V_{base}), it is preferable to send more time sensitive messages first. So in these two parameters, the user can capture both overall value (i.e. V_{base}) and latency tolerance (ttl) of the message queue.

- `in_pubsub_var`: name of the moos variable that is published for received messages to this queue. Not used for DCCL queuing.

- `out_pubsub_var`: name of the moos variable to subscribe to for messages to add to this queue. Not used for DCCL queuing.

An example queuing block (for DCCL messages):

```
<message_set>
  <message>
    <id>23</id>
    ...
    <queuing>
      <ack>false</ack>
      <blackout_time>0</blackout_time>
      <max_queue>1</max_queue>
      <newest_first>true</newest_first>
      <value_base>4</value_base>
      <ttl>1000</ttl>
    </queuing>
  </message>
  ...
</message_set>
```

Modem Driver Parameters (driver_cfg)

- `driver_type`: The only real driver implemented is the `DRIVER_WHOI_MICROMODEM`. `DRIVER_ABC_EXAMPLE_MODEM` is a simple test "modem". `DRIVER_NONE` disables the modem driver.
- `connection_type`: type of connection to make to the modem (`CONNECTION_SERIAL`, `CONNECTION_TCP_AS_CLIENT`, `CONNECTION_TCP_AS_SERVER`).
- `serial_port`: serial port to which the modem is connected.
- `serial_baud`: baud rate to use. Should be set to 19200 for the WHOI Micro-Modem.
- `tcp_port`: networking port to use.
- `tcp_server`: IPv4 networking address of the server to connect to.

Extensions for the WHOI Micro-Modem

- `[MicroModemConfig.nvram_cfg]`: set some modem NVRAM setting to a value. Set `[MicroModemConfig.reset_nvram]: true` to reset all NVRAM (CFG) parameters on startup (`\$CCCFG,ALL,0`). All the `[MicroModemConfig.nvram_cfg]` values are sent after this reset. You do not need to send SRC as this is set to the `modem_id`.
- `[MicroModemConfig.hydroid_gateway_id]`: Set to the HYDROID gateway id (1 or 2) *only if using a HYDROID gateway buoy*. Omit for a normal WHOI Micro-Modem.

Medium Access Control (MAC) Parameters (mac_cfg)

- `type`: type of Medium Access Control. See `http://gobysoft.com/doc/acomms__mac.html#amac_schemes` for an explanation of the various MAC schemes.

- `slot_seconds`: length, in seconds, of each communication slot for the `type: MAC_AUTO_DECENTRALIZED` MAC option.

- `rate`: rate for the `type: MAC_AUTO_DECENTRALIZED` MAC option. For the WHOI Micro-Modem 0 is a single 32 byte packet (FSK), 2 is three frames of 64 bytes (PSK), 3 is two frames of 256 bytes (PSK), and 5 is eight frames of 256 bytes (PSK)

- `expire_cycles`: number of consecutive cycles in which a vehicle can be silent before being removed from the cycle for the `type: MAC_AUTO_DECENTRALIZED` MAC option.

- `slot`: use this repeated field to specify a manual polling or fixed TDMA cycle for the `type: MAC_FIXED_DECENTRALIZED` and `type: MAC_POLLED`.

  - `src`: The sending `modem_id` for this slot.
  - `dest`: The receiving `modem_id` for this slot.
  - `rate`: Bit-rate code for this slot (0-5).
  - `type`: Type of transaction to occur in this slot. Can be `SLOT_DATA` (send a datagram), `SLOT_PING` (send a ranging two-way ping to another modem), `SLOT_REMUS_LBL` (ping a REMUS LBL network (WHOI Micro-Modem only)).
  - `slot_seconds`: The duration of this slot, in seconds.

A.3.6 MOOS variables subscribed to by pAcommsHandler

Except for DCCL <src_var>s and <trigger_var>s, pAcommsHandler uses the Google Protocol Buffers TextFormat class for parsing from MOOS strings. This saves significant effort in manually parsing strings. You should use these same facilities for creating and reading messages. Two helper functions are provided in goby/moos/libmoos_util/moos_protobuf_helpers will help you serialize and parse these messages. See `http://gobysoft.com/doc/acomms.html#protobuf` for a brief overview of Google Protocol Buffers as used in Goby.

- `DCCL`: Most variables subscribed to by pAcommsHandler are configured in the message XML files and are designated by the tags <src_var> (used to fetch data for a particular `message_var` within a DCCL message) and <trigger_var>

(used to trigger the creatinon of a particular DCCL message and possibly provide some data for that message. See A.3.8 for details on the XML configuration.

- `Queue`:
  - Subscribes to the variables given in `queue_cfg.queue.in_pubsub_var` for CCL queue sending. The contents of this MOOS variable should be a serialized ModemDataTransmission).
  - `ACOMMS_RANGE_COMMAND` (type: ModemRangingRequest): You write this to initiate a ranging request outside the MAC schedule. Note in general it is preferable to use the MAC cycle to coordinate data and ranging.
- `MAC`: `ACOMMS_MAC_CYCLE_UPDATE` (type: MACUpdate) You write this to update the MAC cycle for `MAC_FIXED_DECENTRALIZED` and `MAC_POLLED` modes of operation.

For example, to publish a `ACOMMS_MAC_CYCLE_UPDATE`, you would use code like this:

```
// provides serialize_for_moos
#include <goby/moos/libmoos_util/moos_protobuf_helpers.h>
// provides goby::acomms::protobuf::MACUpdate
#include <goby/protobuf/amac.pb.h>

...

MyMOOSApp::Iterate()
{
  if(do_update_mac)
  {
    using namespace goby::acomms::protobuf;
    MACUpdate mac_update;
    mac_update.set_dest(1); // update for us if modem_id == 1
    // add slot to end of existing cycle
    mac_update.set_update_type(MACUpdate::ADD);
    Slot* new_slot = mac_update.add_slot();
    new_slot->set_src(1);  // send from us
    new_slot->set_dest(3); // send to vehicle 3
    new_slot->set_rate(0);
    new_slot->set_slot_seconds(15);
    new_slot->set_type(SLOT_DATA);

    std::string serialized;
    serialize_for_moos (&serialized, mac_update);
    m_Comms.Notify("ACOMMS_MAC_CYCLE_UPDATE", serialized);
```

```
  }
}
```

A.3.7 MOOS variables published by pAcommsHandler

Except for DCCL <publish_var>s (which use a printf style syntax), pAcommsHandler uses the Google Protocol Buffers TextFormat class for serializing to MOOS strings.

- `DCCL`: Most variables published by pAcommsHandler are configured in the message XML files and are designated by the tags <publish_var> within a <publish> block. See A.3.8 for details on the XML configuration.
- `Queue`:
  - `ACOMMS_INCOMING_DATA` (type: ModemDataTransmission) written for all received messages containing a data payload
  - `ACOMMS_OUTGOING_DATA` (type: ModemDataTransmission) written for all queued messages containing a data payload
  - `ACOMMS_RANGE_RESPONSE` (type: ModemRangingReply) written in response to ranging request (to another modem or LBL beacons)
  - `ACOMMS_ACK` (type: ModemDataAck) written when received data is acknowledged acoustically by a third party. Contains the original message.
  - `ACOMMS_EXPIRE` (type: ModemDataExpire) written when a message expires (time-to-live [ttl] exceeded) from the queue before being sent (ack = false) or acknowledged (ack = true)
  - `ACOMMS_QSIZE` (type: QueueSize) written when a queue changes size (pop or push) with the new size of the queue.
- `MAC`: Does not publish anything.
- `ModemDriver`:
  - `ACOMMS_NMEA_IN` (type: string), ModemMsgBase::raw() for all incoming messages ("$CA..." for WHOI Micro-Modem)
  - `ACOMMS_NMEA_OUT` (type: string), ModemMsgBase::raw() for all outgoing messages ("$CC..." for WHOI Micro-Modem)

For example, to read an `ACOMMS_RANGE_RESPONSE`, you would use code like this:

```
// provides parse_for_moos
#include <goby/moos/libmoos_util/moos_protobuf_helpers.h>
// provides goby::acomms::protobuf::ModemRangeReply
#include <goby/protobuf/modem_message.pb.h>

...

MyMOOSApp::OnNewMail()
{
  ...
  if(moos_msg.GetKey() == "ACOMMS_RANGE_RESPONSE")
  {
    using namespace goby::acomms::protobuf;
    ModemRangeReply range_response;
    parse_for_moos (serialized, &range_response);

    // now do what you want to with the nice `range_response` object
    std::cout << "one way travel time to " << range_response.base().dest()
              << " is " << range_response.one_way_travel_time(0) << std::endl;
  }
}
```

A.3.8 DCCL Encoding/Decoding Unit: Overview

Example message XML file

First, let us give a brief background on XML (eXtensible Markup Language). XML files contain tags (like <name>) that are considered "metadata" and define both the structure of the following data and the contents. Order of the tags does not matter for a given level unless otherwise specified. Text data resides both in the tags (like <name>bob</name> or as attributes of the tag (such as <name id="1245"></name>). XML files can be edited with any text editor. For more information on XML consult any number of books on the subject or browse the internet. XML is a very widely used format for storing data that can be both read by both people and computers. Also see section A.3.9 for further examples. Let's call this file example1.xml, which we will use in two following examples:

```
<?xml version="1.0" encoding="ASCII" standalone="yes"?>
<message_set>
  <message>
    <name>GoToCommand</name>
    <id>1</id>
```

```
      <trigger>publish</trigger>
      <trigger_var mandatory_content="CommandType=GoTo">
        OUTGOING_COMMAND
      </trigger_var>
      <size>32</size>
      <header>
        <dest_id>
          <name>Destination</name>
        </dest_id>
      </header>
      <layout>
        <static>
          <name>type</name>
          <value>goto</value>
        </static>
        <int>
          <name>goto_x</name>
          <max>10000</max>
          <min>0</min>
        </int>
        <int>
          <name>goto_y</name>
          <max>10000</max>
          <min>0</min>
        </int>
        <bool>
          <name>lights_on</name>
        </bool>
        <string>
          <moos_var>SPECIAL_INSTRUCTIONS</moos_var>
          <name>new_instructions</name>
          <max_length>10</max_length>
        </string>
        <float>
          <name>goto_speed</name>
          <max>3</max>
          <min>0</min>
          <precision>2</precision>
        </float>
      </layout>
      <on_receipt>
        <publish>
          <publish_var>INCOMING_COMMAND</publish_var>
          <all />
        </publish>
        <publish>
```

```
        <publish_var>SPECIAL_INSTRUCTIONS</publish_var>
        <format>special_instructions=%1%,lights_on=%2%</format>
        <message_var>new_instructions</message_var>
        <message_var>lights_on</message_var>
      </publish>
    </on_receipt>
  </message>
  <message>
    <name>VehicleStatus</name>
    <id>2</id>
    <trigger>time</trigger>
    <trigger_time>30</trigger_time>
    <size>32</size>
    <layout>
      <float>
        <name>nav_x</name>
        <src_var>NAV_X</src_var>
        <max>1000</max>
        <min>0</min>
        <precision>1</precision>
      </float>
      <float>
        <name>nav_y</name>
        <src_var>NAV_Y</src_var>
        <max>1000</max>
        <min>0</min>
        <precision>1</precision>
      </float>
      <enum>
        <name>health</name>
        <src_var>VEHICLE_HEALTH</src_var>
        <value>good</value>
        <value>low_battery</value>
        <value>abort</value>
      </enum>
    </layout>
    <on_receipt>
      <publish>
        <publish_var>STATUS_SUMMARY</publish_var>
        <all />
      </publish>
    </on_receipt>
  </message>
</message_set>
```

A.3.9 DCCL Encoding/Decoding Unit: Designing Messages

Designing a publish triggered message

We will look at two scenarios and detail how to design a proper message file for each scenario. We will reference the example file given in section A.3.8 for both scenarios.

Scenario: you want to command an surface craft to move to a new location:

1. Identify the data: location (x (`goto_x`) and y (`goto_y`) on a local grid). you also want to specify a speed (`goto_speed`) at which it should transit, whether it should have lights (`lights_on`) on or not, and finally a string (`special_instructions`) with possible special instructions. All these data will come in to a moos variable `OUTGOING_COMMAND` on a string like:

```
OUTGOING_COMMAND: Destination=3,CommandType=GoTo,goto_x=351,goto_y=294,
            lights_on=true,special_instructions=make_toast,goto_speed=2.3
```

2. Type the data (i.e. is it an int, a float, a string?) and give the ranges and precisions needed:

   - `goto_x`: integer (in meters) (`int`) that will operate on a (positive valued) local grid not to exceed 10 km in either dimension.
   - `goto_y`: same as `goto_x`.
   - `goto_speed`: speed in m/s. the vehicle cannot exceed 3 m/s and does not go backwards. we would like to give precise speeds to the hundredths place. thus, we need a `float` ranging from 0 to 3 with precision 2.
   - `lights_on`: simply a flag (boolean value) whether to have our lights on or off. thus, we need a `bool` *message_var*.
   - `special_instructions`: We want a field that can hold any string of characters, but we know it will not exceed ten characters. thus, we need a `string` *message_var*.

3. Putting all this together, we can define the <layout> portion of the first message defined in section A.3.8. We do not need any <src_var> tags within the *message_var*s since all the data are contained in the contents of the trigger variable message (`OUTGOING_COMMAND`). That is, when we leave out the <src_var>, pAcommsHandler will insert <src_var>OUTGOING_COMMAND</src_var>, which is exactly what we want. For example, taking one of the *message_var*s:

```
      <int>
        <name>goto_x</name>
        <max>10000</max>
        <min>0</min>
      </int>
```

is exactly the same as saying

```
      <int>
        <name>goto_x</name>
        <src_var>OUTGOING_COMMAND</src_var>
        <max>10000</max>
        <min>0</min>
      </int>
```

4. Now we can fill out the rest of the tags on the `<message>` level:
   - `<name>GoToCommand</name>`: just a name so we can identify this message quickly when reading through the XML.
   - `<trigger>publish</trigger>`: we are creating this message on a publish (to `OUTGOING_COMMAND`).
   - `<trigger_var mandatory_content="CommandType=GoTo"> OUTGOING_COMMAND </trigger_var>`: `OUTGOING_COMMAND` is the trigger variable and it must contain the substring `CommandType=GoTo`. That is, other commands might be published here (e.g. `CommandType=Loiter`, `CommandType=Track`) and we do not define the message structure of those here (this particular `<message>` is only for a GoTo message). Other messages can be created to encode/decode these other command types.
   - `<size>32</size>`: we want this message to fit in a WHOI micromodem FSK frame (32 bytes).
5. Finally, we fill out the `<publish>` section which indicates where (i.e. what moos variables) and how (what format and which part(s) of the message) pAcommsHandler should publish decoded messages upon receipt of hex from other vehicles. Each `<publish>` indicates a separate action that is taken upon receipt of a message. As many `<publish>` sections as desired may be included for a given message. So, for our example message, we want to replicate the original string (a common practice):

```
INCOMING_COMMAND: CommandType=GoTo,goto_x=351,goto_y=294,
            lights_on=true,special_instructions=make_toast,goto_speed=2.3
```

to do this we fill out a publish <`all`>. This is the simplest form of the <`publish`> section:

```
<on_receipt>
  <publish>
    <publish_var>INCOMING_COMMAND</publish_var>
    <all />
  </publish>
</on_receipt>
```

this says to take every *message_var* and make a “key=value” comma-delimited string from it. the above <`publish`> block is a shortcut for a much longer form:

```
<on_receipt>
  <publish>
    <publish_var>INCOMING_COMMAND</publish_var>
    <format>type=goto,goto_x=%1%,goto_y=%2%,lights_on=%3%,
    special_instructions=%4%,goto_speed=%5%</format>
    <message_var>goto_x</message_var>
    <message_var>goto_y</message_var>
    <message_var>lights_on</message_var>
    <message_var>special_instructions</message_var>
    <message_var>goto_speed</message_var>
  </publish>
</on_receipt>
```

These two blocks are functionally identical.

We may want to also publish the `special_instructions` to another moos variable, so that:

```
SPECIAL_INSTRUCTIONS: special_instructions=make_toast,lights_on=true
```

we can do this with another publish block:

```
<publish>
  <publish_var>SPECIAL_INSTRUCTIONS</publish_var>
  <format>special_instructions=%1%,lights_on=%2%</format>
  <message_var>new_instructions</message_var>
  <message_var>lights_on</message_var>
</publish>
```

in this case the <format> block is necessary because the default would be <format>new_instructions=%1%,lights_on=%2%</format> not <format>special_instructions=%1%,lights_on=%2%</format>.

Those are the basics to designing a publish triggering message.

Designing a time triggered message Scenario: we need a status message that grabs data from various moos variables and publishes them (encoded) on a time interval. We will not go into as much detail here, but rather highlight the changes from the previous scenario.

- you will notice

```
<trigger>time</trigger>
<trigger_time>30</trigger_time>
```

  instead of

```
<trigger>publish</trigger>
<trigger_var mandatory_content="CommandType=GoTo">
  OUTGOING_COMMAND
</trigger_var>
```

  this indicates that a message should be made on a time interval (given by <trigger_time>, which is every 30 seconds here), rather than on a publish to some MOOS variable.

- you will notice that all the *message_var*s have a <src_var> tag, which was omitted in the previous example since we were taking data from the trigger variable. Obviously, there is no trigger variable now so we must specify a location for the data to come from (in the MOOSDB). The newest available value will be used when the message needs to be made. This means there is no guarantee that the data is fresh. Thus, you should use MOOS variables that are often updated for a `<trigger>time</trigger>` message. If this is not the case, a `<trigger>publish</trigger>` message (see previous scenario) may be a better choice.

- the format of the value read from the `<src_var>` can have several options. First, if the *message_var* is of a numeric type (`<int>`, `<float>`, `<bool>`) and the `<moos_var>` is a double, the value of the double is used as is (with appropriate rounding and type casting). If the *message_var* is a string, two options are available. First, pAcommsHandler looks for a substring of the form:

  ```
  name=value
  ```

  within the string and picks out `value` to send for the message. If there is no such `name=` substring, the entire string is converted to the appropriate form. An example: we have a `<float>` called `<name>my_float</name>` that has a tag `<moos_var>SOME_FLOAT_VARIABLE</moos_var>`:

  – if

```
1 (double)SOME_FLOAT_VARIABLE: 3.56
```

  then 3.56 is sent.

  – if instead

```
2 (string)SOME_FLOAT_VARIABLE: "my_float=3.56"
```

  then 3.56 is still sent.

  – if instead

```
3 (string)SOME_FLOAT_VARIABLE: "3.56"
```

  again, 3.56 is sent.

  – Finally, if some other string like

```
4 (string)SOME_FLOAT_VARIABLE: "blah=3.56"
```

  then `blah=3.56` is converted to a float, which will probably be zero or something else undesired. In other words, case `4` is not what you want, whereas `1-3` are fine.

Further examples

- I currently store some example working message files in `goby/xml`. look for .xml files in this directory for further examples.
- Probably the simplest message you can make (for a single string MOOS variable published to `IN_MESSAGE` that gets truncated at 26 chars (need six bytes for the DCCL header) and sent to broadcast):

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <name>Chat</name>
    <id>1</id>
    <size>32</size>
    <queuing>
      <ack>true</ack>
      <newest_first>false</newest_first>
    </queuing>
    <layout>
      <string>
        <name>message</name>
        <max_length>26</max_length>
      </string>
    </layout>

    <!-- only used by pAcommsHandler (publish/subscribe)-->
    <trigger>publish</trigger> <!-- pack -->
    <trigger_var>OUT_MESSAGE</trigger_var>
    <on_receipt> <!-- unpack -->
      <publish>
        <publish_var>IN_MESSAGE</publish_var>
        <message_var>message</message_var>
      </publish>
    </on_receipt>
    <!-- end used by pAcommsHandler -->

  </message>
</message_set>
```

A.3.10 DCCL Encoding/Decoding Unit: XML Tag Reference

The XML tag reference is now part of the Goby Developers documentation (http://gobysoft.com/doc:

- See `http://gobysoft.com/doc/acomms__dccl.html#dccl_tags` for a structure of all the allowed tags.
- Visit `http://gobysoft.com/doc/acomms__dccl.html#dccl_tags_details` for an up-to-date reference of all the DCCL tags with a description of their usage.

Algorithms

You can perform a number of simple algorithms on data either before encoding (specified in the message_var tag (e.g. <string algorithm="">) or after receipt (specified in the <message_var> tag. You can apply more than one algorithm by separating them with commas and they are processed in the order given. The currently implemented algorithms include:

- `to_upper`: converts string, enum, or bool to uppercase
- `to_lower`: converts string, enum, or bool to lowercase
- `angle_0_360`: wraps float or int angle in degrees into the range of [0, 360)
- `angle_-180_180`: wraps float or int angle in degrees into the range of [-180, 180)
- `lon2utm_x`: converts longitude to a local utm coordinate (meters) used by LAMSS[2]. Requires `LatOrigin` and `LongOrigin` to be specified at the top of the moos file. Since a UTM conversion requires a lon/lat pair, you must specify the latitude variable here to pair with by adding a colon after this algorithm followed by the name of the latitude variable. e.g.

  ```
  <message_var algorithm="lon2utm_x:our_lat">our_lon</message_var>
  ```

  converts `our_lon` to a local x (easting) using `our_lat` as the latitude point.
- `lat2utm_y`: similar to `lon2utm_x` but for latitude. e.g.

  ```
  <message_var algorithm="lat2utm_y:our_lon">our_lat</message_var>
  ```

  converts `our_lat` to a local y (northing) using `our_lon` as the longitude point.

[2]we define a latitude/longitude origin near our basis of operations. From this datum we calculate the UTM northings (y) and eastings (x). All further UTM calculations are the offset from this datum point. This offset is what is returned by this algorithm. Contact me if you need more information on this.

- utm_x2lon: the reverse conversion from x to longitude. similarly to the latitude, longitude to x,y conversion you must pair x and y. e.g.,

```
<message_var algorithm="utm_x2lon:our_y">our_x</message_var}
```

- utm_y2lat: example:

```
<message_var algorithm="utm_y2lat:our_x">our_y</message_var}
```

- modem_id2name: converts a WHOI modem_id to a vehicle name. requires a file (path given in the .moos as modem_id_lookup_path: "/path/to/modemidlookup.txt". an example file:

```
// modem_id, vehicle name (should be community name), vehicle type
0, broadcast, broadcast
1, endeavor, ship
3, unicorn, auv
4, macrura, auv
```

  if no match is found, the modem_id is returned as a string (e.g. "10").

- name2modem_id: performs the (case insensitive) reverse lookup on the same file. if no match is found, atoi(name.c_str()) is returned (probably zero unless you passed something like "4" to this function).

- modem_id2type: similar to modem_id2name but returns the type of the vehicle (ship, auv, etc.)

- power_to_dB: takes $10 \log_{10}$ of the value.

- dB_to_power: takes power antilog of the value.

- alg_TSD_to_soundspeed: applied to temperature, with references to salinity and depth, calculates the speed of sound using the Mackenzie equation. For example:

```
<message_var algorithm="alg_TSD_to_soundspeed:sal:depth">temp</message_var>
```

- add: adds the reference <message_var> to the current <message_var>. example: <message_var algorithm="add:b">a</message_var> adds b to a.

- subtract: subtracts the reference <message_var> from the current <message_var>.

A.3.11 DCCL Encoding/Decoding Unit: Under the Hood

See http://gobysoft.com/doc/acomms__dccl.html#dccl_how and [17] for details on how the DCCL encoding is done.

A.3.12 Priority Message Queuing Unit

pAcommsHandler takes all the configured queues and maintains a stack of messages for each queue. when it is prompted by data by the modem, it has a priority "contest" between the queues. the queue with the current highest priority (as determined by the `value_base` and `ttl` fields) is selected. The next message in that queue is then provided to the MicroModem to send. For modem messages with multiple frames per packet, each frame is a separate contest. Thus a single packet may contain frames from different queues (e.g. a rate 5 PSK packet has eight 256 byte frames. frame 1 might grab a STATUS message since that has the current highest queue. then frame 2 may grab a BTR message and frames 3-8 are filled up with CTD messages (e.g. STATUS is in blackout, BTR queue is empty)). See http://gobysoft.com/doc/acomms__queue.html#queue_priority for more

For messages with `ack: true` (acknowledge requested), the last message continues to be re-sent (that is, it is not popped from the message queue) until the ACK is received from the modem (thus blocking the sending of other messages). Messages with `ack: false` are popped and discarded when they are sent (no retries).

If you do not wish for dynamic growth of the priorities, simply set the `ttl` to the special value 0. Then the priorities grow as $P = V\_base$ and messages never expire. Note that this is the same as setting `ttl` = ∞.

Messages not to us are ignored We choose modem id 0 as broadcast. thus messages with the destination field = 0 will always be read by all nodes and reported to the appropriate moos variable. Otherwise, we ignore messages unless they correspond to our modem id. so if you send a message to modem id 10, pAcommsHandler for modem ids $1 \rightarrow 9, 11 \rightarrow N$ will ignore that. This is not the default behavior of the WHOI Micro-Modem, which always reports data, regardless of the sender's ID.

The XML tag reference is now part of the Goby Developers documentation (http://gobysoft.com/doc:

- See http://gobysoft.com/doc/acomms__queue.html#queue_tags for a structure of all the allowed tags.
- http://gobysoft.com/doc/acomms__queue.html#queue_tags_details provides an up-to-date reference of all the Queue tags with a description of their usage.

A.3.13 Modem Driver Unit

The Modem driver unit current supports the WHOI Micro-Modem acoustic modem and is extensible to other acoustic modems. To directly monitor the modem feed, subscribe to ACOMMS_NMEA_IN and ACOMMS_NMEA_OUT. For a complete list of supported commands of the WHOI Micro-Modem, see `http://gobysoft.com/doc/acomms__driver.html#acomms_mmdriver`.

A.3.14 Medium Access Control (MAC) Unit

The MAC unit uses time division (TDMA) to attempt to ensure a collision-free acoustic channel.

pAcommsHandler supports two variants of the TDMA MAC scheme: centralized and decentralized. As the names suggest, Centralized TDMA (`type: MAC_POLLED`) involves control of the entire cycle from a single master node, whereas each node's respective slot is controlled by that node in Decentralized TDMA. Within decentralized TDMA, Goby supports both a fixed (preprogrammed) cycle (`type: MAC_FIXED_DECENTRALIZED`) and an autodiscovery mode (`type: MAC_AUTO_DECENTRALIZED`). To disable the pAcommsHandler MAC, use (`type: MAC_NONE`)

Centralized TDMA (Polling)

Centralized TDMA involves a master node (usually aboard the Research Vessel or on land) which initiates every transmission for the entire communcations cycle (i.e. "polls" each node for data). Thus, the other nodes are not required to maintain synchronized clocks as the timing is all performed on the master node.

This style of MAC has been widely used for small AUV operations using the WHOI Micro-Modem. Its principal advantages are that it has 1) no requirement for synchronized clocks, 2) full control over the communications cycle at runtime (assuming the master is accessible to the vehicle operators, as is usually the case); and 3) a master who can acknowledge "broadcast" messages.

However, centralized TDMA has a number of substantial disadvantages. In order for a third-party master to initiate a transmission, an acoustic packet must be sent for this initialization. This additional "cycle initialization" packet, like any acoustic message, has a high chance of being lost (after which the data are never sent because the sending node did not receive a cycle initialization message), consumes power, and lengthens the time of the communications slot. See Fig. A.6 for the various parts of the communication cycle with (for Centralized TDMA) and without (for Decentralized TDMA) the cycle initialization message. The additional time required

(a) Centralized TDMA

(b) Decentralized TDMA

Figure A.6: Comparison of the time needed for a single slot for the two types of TDMA supported by pAcommsHandler. Eq. A.1 gives the additional length of time required by the Centralized variant.

for each slot of Centralized TDMA is

$$\tau_{ci} + r_{max}/c \tag{A.1}$$

where τ_{ci} is the length (in seconds) of the cycle initalization packet (about one second for the WHOI Micro-Modem), r_{max} is the maximum range of the network (typically of order 1000s of meters), and c is the compressional speed of sound (nominally 1500 m/s).

Decentralized TDMA with passive auto-discovery

Decentralized TDMA removes the cycle initialization packet and thus reduces the length of each slot and the chance of errors. However, it introduces the constraint of synchronized clocks[3] for all nodes, which can be somewhat tricky to maintain underwater.

Decentralized TDMA gives each vehicle a single slot in which it transmits. Each vehicle initiates its own transmission at the start of its slot. Collisions are avoided by each vehicle following the same rules about slot placement within the time window (based on the time of day). All slots are ordered by ascending acoustic MAC address (or "modem identification number"), which is an unsigned integer unique for each network.

During the runtime of the network, it is often desirable to add or remove nodes. Since the MAC is spread throughout the nodes, there is no easy way to change the

[3]the accuracy of the clock synchronization can be low relative to other timing needs such as bistatic sonar. Generally, accuracy better than 0.1 seconds is acceptable; higher inaccuracies can be handled by increasing the guard time on both sides of each slot.

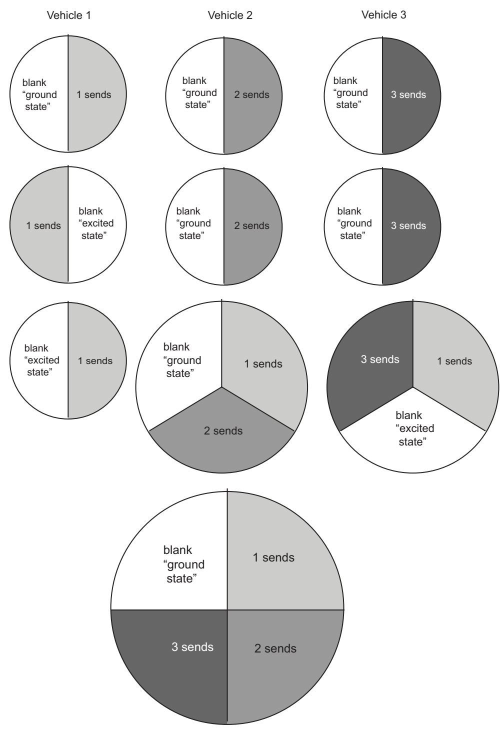

Figure A.7: Graphical example of auto discovery for three nodes launched at the same time. Each circle represents the vehicle's cycle at each time step (represented by horizontal rows) based on the vehicle's current knowledge of the world. In the first row, all vehicles only know of themselves and put the blank slot in the last slot; thus, all communications collide and no discoveries are made. In the second row, vehicle 1's blank is moved (by pseudo-chance) to the penultimate (first) slot, so vehicles 2 and 3 discover 1. Then, in the third row vehicles 2 and 3 are discovered by the others because vehicle 3 moves its blank slot. By the fourth row all vehicles have discovered the others and continue to transmit without collision following the cycle diagrammed on this row.

time	vehicle 1	vehicle 2	result
0	send	send	collision
15	blank	blank	nothing
30	blank	send	success: 1 discovers 2
45	cycle wait	blank	nothing
60	cycle wait	send	success
75	cycle wait	blank	nothing
90	send	blank	success: 2 discovers 1
105	listen for 2	cycle wait	nothing
120	blank	cycle wait	nothing
135	send	listen for 1	success
150	listen for 2	send	success
165	blank	blank	nothing
180	send	listen for 1	success
195	blank	blank	nothing
210	listen for 2	send	success

Table A.1: Example initialization for the Decentralized TDMA with autodiscovery. By 135 seconds, both vehicles have discovered each other and are synchronized. Thus, no more collisions will occur. This scenario assumes that both vehicles always have some data to send during their slot.

cycle during runtime. *libamac* supports passive auto-discovery (and subsequent expiration) of nodes to provide a solution to this problem. This auto-discovery is passive because it requires no control messaging beyond the normal communications between nodes.

Vehicles are discovered by shifting a blank slot in each cycle based on their knowledge of the world and the time of day. If a new vehicle is heard from during the blank, it is added to the listening vehicle's knowledge of the world and hence their cycle. In the simplified situation (which is really a worst case scenario) discovery is defined by a single vehicle transmitting during a cycle and all the others silent (the current slot is not equal to each vehicle's acoustic MAC address).

A.3.15 Simple complete example MOOS files

Example 1: Basic CCL (goby/cfg/MOOS/basic_ccl)

This example sends the bytes `0x020304` from node 1 (`mm1`) to node 2 (`mm2`). It shows use of all the parts of pAcommsHandler except the DCCL encoding / decoding unit.

I use `iModemSim` here to simulate the WHOI Micro-Modem. This process is available in moos-ivp-local (http://oceanai.mit.edu/moos-ivp/pmwiki/pmwiki.php?n=Support.Milocal). You can also easily substitute real modems by removing iModemSim references and changing the `serial_port`.

MOOS file for Node 1: goby/cfg/MOOS/basic_ccl/mm1.moos

```
// t. schneider tes@mit.edu 2.16.11

// bare bones acoustic communications
// stack for topside receiver
// for CCL message

ServerHost = localhost
ServerPort = 9101
Community  = mm1

LatOrigin = 0
LongOrigin = 0

ProcessConfig = ANTLER
{
    MSBetweenLaunches = 10
    Run = MOOSDB @ NewConsole = false

    /////////////////////////////////////
    // acomms related
    /////////////////////////////////////
    // queuing
    Run = pAcommsHandler        @ NewConsole = true
    // modem simulator
    Run = iModemSim             @ NewConsole = true

    // simulate CCL data source
    Run = uTimerScript            @ NewConsole = true
}

ProcessConfig = pAcommsHandler
{
    modem_id: 1

    driver_type: DRIVER_WHOI_MICROMODEM

    driver_cfg
    {
        serial_port: "/dev/ttyLOOPA2"
```

```
    # doesn't work with iModemSim, set to true for real ops
        [MicroModemConfig.reset_nvram]: false
    }

    mac_cfg
    {
        type: MAC_FIXED_DECENTRALIZED
        slot
        {
            src: 1
            dest: 2
            rate: 0
            type: SLOT_DATA
            slot_seconds: 10
        }
    }

    queue_cfg
    {
        queue
        {
            key {
                type: QUEUE_CCL
                id: 2 # decimal CCL id (first byte)
            }
            in_pubsub_var: "IN_TEST_32B"
            out_pubsub_var: "OUT_TEST_32B"
            name: "TEST"
        }
    }
}
// must set serial_loopbacks to use
// as root run the shell script (in moos-ivp-local/scripts)
// > loopbacks
ProcessConfig = iModemSim
{
        AppTick   = 4
CommsTick = 4

Port = /dev/ttyLOOPA1
Speed = 19200

IPPort = 49234
BroadcastAddr = 127.0.0.1

        InputLocType = constant_local
```

```
        ConstantPosX = 0
        ConstantPosY = 0
        ConstantDepth = 0
}



ProcessConfig = uTimerScript
{
    // data is 2 2 3 4 in octal
    EVENT  = var=OUT_TEST_32B, val="data: "\002\002\003\004"", time = 10
    RESET_TIME = end
}
```

MOOS file for Node 2: goby/cfg/MOOS/basic_ccl/mm2.moos

```
// t. schneider tes@mit.edu 4.28.10

// bare bones acoustic communications
// stack for auv
// for CCL message

ServerHost = localhost
ServerPort = 9102
Community  = mm2

LatOrigin = 0
LongOrigin = 0

ProcessConfig = ANTLER
{
    MSBetweenLaunches = 10
    Run = MOOSDB @ NewConsole = false

    ////////////////////////////////////
    // acomms related
    ////////////////////////////////////
    // queuing
    Run = pAcommsHandler        @ NewConsole = true

    Run = iModemSim             @ NewConsole = true
}

ProcessConfig = pAcommsHandler
```

```
{
    modem_id: 2

    driver_type: DRIVER_WHOI_MICROMODEM

    driver_cfg
    {
        serial_port: "/dev/ttyLOOPB2"
    # doesn't work with iModemSim, set to true for real ops
        [MicroModemConfig.reset_nvram]: false
    }

    mac_cfg
    {
        type: MAC_FIXED_DECENTRALIZED
        slot
        {
            src: 1
            dest: 2
            rate: 0
            type: SLOT_DATA
            slot_seconds: 10
        }
    }

    queue_cfg
    {
        queue
        {
            key {
                type: QUEUE_CCL
                id: 2  # decimal CCL id (first byte)
            }
            in_pubsub_var: "IN_TEST_32B"
            out_pubsub_var: "OUT_TEST_32B"
            name: "TEST"
        }
    }
}


// must set serial_loopbacks to use
// as root run the shell script (in moos-ivp-local/src/bin)
// > loopbacks
ProcessConfig = iModemSim
{
```

```
AppTick   = 4
CommsTick = 4

Port = /dev/ttyLOOPB1
Speed = 19200

IPPort = 49234
BroadcastAddr = 127.0.0.1

        InputLocType = constant_local
        ConstantPosX = 0
        ConstantPosY = 0
        ConstantDepth = 0
}
```

Example 2: DCCL and CCL (goby/cfg/MOOS/ccl_and_dccl)

This example sends the DCCL "Simple Status" messsage from node 1 (`mm1`) to node 2 (`mm2`). `mm2` sends the REMUS CCL State message to `mm1`. It thus uses all the components of pAcommsHandler. As in the previous example, you can use real modems by removing iModemSim and changing the `serial_port` to the proper real serial port.

MOOS file for Node 1: goby/cfg/MOOS/ccl_and_dccl/mm1.moos

```
// t. schneider tes@mit.edu 3.2.11

// bare bones acoustic communications
// stack for topside receiver

ServerHost = localhost
ServerPort = 9101
Community  = mm1

LatOrigin =  42.35
LongOrigin = -70.95

NoNetwork = true
modem_id_lookup_path = modemidlookup.txt


ProcessConfig = ANTLER
{
    MSBetweenLaunches = 10
```

```
    Run = MOOSDB @ NewConsole = false

    Run = pREMUSCodec           @ NewConsole = true, XConfig=1
    Run = pAcommsHandler        @ NewConsole = true, XConfig=2
    Run = iModemSim             @ NewConsole = true, XConfig=3

    1 = -geometry,80x15+0+0
    2 = -geometry,80x100+0+230
    3 = -geometry,80x15+0+570
}

ProcessConfig = pREMUSCodec
{
  mdat_state_var: "IN_REMUS_STATUS"
  mdat_state_out: "OUT_REMUS_STATUS"
  create_status: false
}


ProcessConfig = pAcommsHandler
{
    common
    {
      verbosity: VERBOSITY_GUI
        initializer { type: INI_DOUBLE  global_cfg_var: "LatOrigin"  moos_var: "LAT_ORIGIN" }
        initializer { type: INI_DOUBLE  global_cfg_var: "LongOrigin" moos_var: "LONG_ORIGIN" }
        initializer { type: INI_STRING  moos_var: "VEHICLE_TYPE" sval: "topside" }
        initializer { type: INI_STRING  moos_var: "VEHICLE_NAME" sval: "mm1" }
        initializer { type: INI_DOUBLE  moos_var: "NAV_X"  dval: 100 }
        initializer { type: INI_DOUBLE  moos_var: "NAV_Y"  dval: 300 }
        initializer { type: INI_DOUBLE  moos_var: "NAV_HEADING" dval: 150 }
        initializer { type: INI_DOUBLE  moos_var: "NAV_SPEED" dval: 0 }
        initializer { type: INI_DOUBLE  moos_var: "NAV_DEPTH" dval: 0 }
    }

    modem_id: 1

    driver_type: DRIVER_WHOI_MICROMODEM
    driver_cfg
    {
      serial_port: "/tmp/ttyLOOPA2"
# doesn't work with iModemSim, set to true for real ops
      [MicroModemConfig.reset_nvram]: false
    }

    mac_cfg
```

```
    {
        type: MAC_FIXED_DECENTRALIZED
        slot { src: 1  dest: 2  rate: 0  type: SLOT_DATA  slot_seconds: 10 } # downlink
        slot { src: 2  dest: 1  rate: 0  type: SLOT_DATA  slot_seconds: 10 } # uplink
    }

    queue_cfg
    {
        queue
        {
            key {
                type: QUEUE_CCL
                id: 14 # decimal CCL id (first byte)
            }
            in_pubsub_var: "IN_REMUS_STATUS"
            out_pubsub_var: "OUT_REMUS_STATUS"
            name: "Remus_State"
        }
    }

    dccl_cfg
    {
        message_file { path: "/home/toby/goby/share/xml/simple_status.xml" }
    }
}

// must set serial_loopbacks to use
// as root run the shell script (in moos-ivp-local/src/bin)
// > loopbacks
ProcessConfig = iModemSim
{
AppTick   = 4
CommsTick = 4

Port = /tmp/ttyLOOPA1
Speed = 19200

IPPort = 49234
BroadcastAddr = 127.0.0.1

        InputLocType = constant_local
        ConstantPosX = 0
        ConstantPosY = 0
        ConstantDepth = 0
}

```

MOOS file for Node 2: goby/cfg/MOOS/ccl_and_dccl/mm2.moos

```
// t. schneider tes@mit.edu 3.2.11

// bare bones acoustic communications
// stack for auv

ServerHost = localhost
ServerPort = 9102
Community  = mm2

LatOrigin =  42.35
LongOrigin = -70.95

modem_id_lookup_path = modemidlookup.txt
modem_id = 2

NoNetwork = true

ProcessConfig = ANTLER
{
    MSBetweenLaunches = 10

    Run = MOOSDB @ NewConsole = false

    Run = pREMUSCodec           @ NewConsole = true, XConfig=1
    Run = pAcommsHandler        @ NewConsole = true, XConfig=2
    Run = iModemSim             @ NewConsole = true, XConfig=3

    1 = -geometry,80x15-0+0
    2 = -geometry,80x100-0+230
    3 = -geometry,80x15-0+570
}

ProcessConfig = pREMUSCodec
{
  create_status: true

  mdat_state_var: "IN_REMUS_STATUS"
  mdat_state_out: "OUT_REMUS_STATUS"
  modem_id_lookup_path: "modemidlookup.txt"
}

ProcessConfig = pAcommsHandler
{
    common
    {
```

```
      verbosity: VERBOSITY_GUI
        initializer { type: INI_DOUBLE  global_cfg_var: "LatOrigin"  moos_var: "LAT_ORIGIN" }
        initializer { type: INI_DOUBLE  global_cfg_var: "LongOrigin" moos_var: "LONG_ORIGIN" }
        initializer { type: INI_STRING  moos_var: "VEHICLE_TYPE"  sval: "auv" }
        initializer { type: INI_STRING  moos_var: "VEHICLE_NAME"  sval: "mm2" }
        initializer { type: INI_DOUBLE  moos_var: "NAV_X"  dval: 123 }
        initializer { type: INI_DOUBLE  moos_var: "NAV_Y"  dval: 321 }
        initializer { type: INI_DOUBLE  moos_var: "NAV_HEADING" dval: 45 }
        initializer { type: INI_DOUBLE  moos_var: "NAV_SPEED" dval: 1.2 }
        initializer { type: INI_DOUBLE  moos_var: "NAV_DEPTH" dval: 111 }
    }

    modem_id: 2
    modem_id_lookup_path: "modemidlookup.txt"

    driver_type: DRIVER_WHOI_MICROMODEM
    driver_cfg
    {
        serial_port: "/tmp/ttyLOOPB2"
    # doesn't work with iModemSim, set to true for real ops
        [MicroModemConfig.reset_nvram]: false
    }

    mac_cfg
    {
        type: MAC_FIXED_DECENTRALIZED
        slot { src: 1  dest: 2  rate: 0  type: SLOT_DATA  slot_seconds: 10 } # downlink
        slot { src: 2  dest: 1  rate: 0  type: SLOT_DATA  slot_seconds: 10 } # uplink
    }

    queue_cfg
    {
        queue
        {
            key { type: QUEUE_CCL id: 14 }
            in_pubsub_var: "IN_REMUS_STATUS"
            out_pubsub_var: "OUT_REMUS_STATUS"
            name: "Remus_State"
        }
    }

    dccl_cfg
    {
        message_file { path: "/home/toby/goby/share/xml/simple_status.xml"
                       manipulator: NO_ENCODE }
    }
```

```
}

// must set serial_loopbacks to use
// as root run the shell script (in moos-ivp-local/src/bin)
// > loopbacks
ProcessConfig = iModemSim
{
AppTick   = 4
CommsTick = 4

Port = /tmp/ttyLOOPB1
Speed = 19200

IPPort = 49234
BroadcastAddr = 127.0.0.1

        InputLocType = constant_local
        ConstantPosX = 0
        ConstantPosY = 0
        ConstantDepth = 0
}
```

XML definition of Simple Status: goby/xml/simple_status.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
  <message>
    <name>SIMPLE_STATUS</name>
    <trigger>time</trigger>
    <trigger_time>5</trigger_time>
    <size>32</size>
    <header>
      <id>20</id>
      <time>
        <name>Timestamp</name>
      </time>
      <src_id algorithm="to_lower,name2modem_id">
        <name>Node</name>
        <moos_var>VEHICLE_NAME</moos_var>
      </src_id>
    </header>
    <layout>
      <static>
        <name>MessageType</name>
```

```
        <value>LAMSS_STATUS</value>
      </static>
      <float>
        <name>nav_x</name>
        <moos_var>NAV_X</moos_var>
        <max>100000</max>
        <min>-100000</min>
        <precision>0</precision>
      </float>
      <float>
        <name>nav_y</name>
        <moos_var>NAV_Y</moos_var>
        <max>100000</max>
        <min>-100000</min>
        <precision>0</precision>
      </float>
      <float>
        <name>Speed</name>
        <moos_var>NAV_SPEED</moos_var>
        <max>20</max>
        <min>-2</min>
        <precision>1</precision>
      </float>
      <float algorithm="angle_0_360">
        <name>Heading</name>
        <moos_var>NAV_HEADING</moos_var>
        <max>360</max>
        <min>0</min>
        <precision>2</precision>
      </float>
      <float>
        <name>Depth</name>
        <moos_var>NAV_DEPTH</moos_var>
        <max>6400</max>
        <min>0</min>
        <precision>1</precision>
      </float>
    </layout>

    <!-- decoding -->
    <on_receipt>
      <publish>
        <moos_var>STATUS_REPORT_IN</moos_var>
        <all />
      </publish>
      <publish>
```

```
        <moos_var>NODE_REPORT</moos_var>
        <format>NAME=%1%,TYPE=%2%,UTC_TIME=%3$.0lf,X=%4%,Y=%5%,LAT=%6$lf,LON=%7$lf,SPD=%8%,HDG=%9%,DEP
        <message_var algorithm="modem_id2name">Node</message_var>
<message_var algorithm="modem_id2type">Node</message_var>
        <message_var>Timestamp</message_var>
        <message_var>nav_x</message_var>
        <message_var>nav_y</message_var>
        <message_var algorithm="utm_y2lat:nav_x">nav_y</message_var>
        <message_var algorithm="utm_x2lon:nav_y">nav_x</message_var>
        <message_var>Speed</message_var>
        <message_var>Heading</message_var>
        <message_var>Depth</message_var>
      </publish>
    </on_receipt>
    <queuing>
      <ack>false</ack>
      <blackout_time>10</blackout_time>
      <ttl>300</ttl>
      <value_base>1.5</value_base>
    </queuing>
  </message>
</message_set>

```

Modem Lookup Table: goby/cfg/MOOS/ccl_and_dccl/modemidlookup.txt

```
1,mm1,topside
2,mm2,auv
```

A.4 iCommander

`iCommander` is a topside command and control (C2) tool which provides a simple console for issuing commands through the acoustic network. By sharing DCCL message configuration (XML) files with `pAcommsHandler` it automatically adapts to the current message set, without any need to change code.

Parameters for the iCommander Configuration Block

Example .moos file The moos file is simple since the bulk of the configuration is stored in separate XML files (see section A.3.8 for the configuration of these files):

As with pAcommsHandler, the above configuration file can be generated at any time with the command:

```
iCommander --example_config
```

Filling out the .moos file Some of the DCCL configuration (`dccl_cfg`) parameters are not used, such as the `crypto_passphrase`.

- `common`: See section A.3.5.
- `dccl_cfg.message_file`: path to an XML file containing a message set of one or messages. These are the DCCL messages. You can also load messages XML files through the Main Menu in the program.
- `load`: path to a file of iCommander saved message(s) to load automatically on startup. You can also load messages through the Main Menu in the program.

Reference Sheet

Main Menu

```
 ---------------------------------------------------------------
|            iCommander: Vehicle Command Message Sender         |
|                        2 messages loaded.                     |
|    Main Menu:                                                 |
|    > Return to active message                                 |
|    > Select Message                                           |
|    > Load                                                     |
|    > Save                                                     |
|    > Import Message File                                      |
|    > Exit                                                     |
|_______________________________________________________________|
```

- *Return to active message* - only available if you have actively edited a message this session. Choose to return to the editing screen of the last message you were editing.
- *Select Message* - pick a message type to edit. All messages are read from DCCL (dynamic compact control language) XML message files.
- *Load* - load a saved message parameters file. This allows you to save values for message fields from session to session.
- *Save* - saves all open messages to a single file for later use. These files are plain text for easy use outside iCommander.

- *Import Message File* - import another DCCL XML file for use.
- *Exit* - quit cleanly.

Editing screen

```

  __________________________________________________
 |                                                  |
 |Editing message variable 1 of 22: MessageType    |
 |(static) you cannot change the value of this field|
 |__________________________________________________|

 ___________________________________________________
|                                                   |
|Message (Type: SENSOR_PROSECUTE)                   |
|22 entries total                                   |
|        {Enter} for options                        |
|        {Up/Down} for more message variables       |
|                                                   |
|                                 _________________ |
|                                |                 ||
|1. MessageType (static)         |SENSOR_PROSECUTE ||
|                                |_________________||
|                                 _________________ |
|                                |                 ||
|2. SensorCommandType (int)      |1                ||
|                                |_________________||
|                                 _________________ |
|                                |                 ||
|3. SourcePlatformId (int)       |0                ||
|                                |_________________||
|                                 _________________ |
|                                |                 ||
|4. DestinationPlatformId (int)  |3                ||
|                                |_________________||
|___________________________________________________|
```

Scroll to select the box to edit. Note that you will need to scroll up or down off the screen to see all the fields at once. The information box at the top will tell you how large the field can be based on the DCCL settings. You cannot enter a value outside these ranges. Hit enter to get the editing menu.

Editing menu

```
    ----------------------------------------------------------
   |                                                          |
   |                     Choose an action                     |
   |> Return to message                                       |
   |> Send                                                    |
   |> Preview                                                 |
   |> Quick switch to another open message                    |
   |> Insert special: current time                            |
   |> Insert special: local X,Y to longitude,latitude         |
   |> Insert special: community                               |
   |> Insert special: modem id                                |
   |> Clear message                                           |
   |> Main Menu                                               |
   |                                                          |
   |                                                          |
   |__________________________________________________________|
```

- *Return to message*
- *Send* - publish the variables for use by pAcommsHandler
- *Preview* - preview the message to be sent in exact syntactical form
- *Quick switch to another open message* - switch to another message with information (either edited this session or loaded)
- *Insert special: current time* - insert a placeholder ("_time") that will be replaced with the current UNIX time when message is sent (e.g. 1236053988). Shortcut: type 't' directly into the field and bypass this menu.
- *Insert special: local X,Y to longitude,latitude* - insert a placeholder designator to do a UTM local grid to latitude / longitude conversion. first the latitude (Y or northings) is entered ("y(lat)1:"), then you choose where to put the longitude (X or eastings) ("x(lon)1:"). after the colon enter the desired value in meters that will be converted to latitude/longitude based in the LatOrigin/LongOrigin set in the top of the MOOS file. Note that you may have more than one pair of x/y. This is the reason for the number following "y(lat)"/"x(lon)". "y(lat)1" is paired with "x(lon)1", "y(lat)2" is paired with "x(lon)2", etc. Shortcut: type 'y' or 'x' respectively directly into the fields and bypass this menu.
- *Insert special: community* - insert the name of this MOOS community.
- *Insert special: modem id* - choose a modem id from a list of names. This is based off the modem id lookup table used by pAcommsHandler.

- Clear message
- Main Menu

Acknowledgments If you are using pAcommsHandler with the ACK field set to 1 (true), all acoustic message acknowledgments are displayed at the top of the screen. For example, the ack of a `LAMSS_DEPLOY` message would look like this:

```
 ____________________________________________
|                                            |
|Message acknowledged from queue: LAMSS_DEPLOY|
| for destination: 5                         |
| at time: 2011-Mar-03 22:38:12              |
|____________________________________________|
```

Similarly, expired messages (messages that exceed their *ttl* without being sent) are shown as well:

```
 ____________________________________________
|                                            |
|Message expired from queue: LAMSS_DEPLOY    |
| for destination: 5                         |
| at time: 2011-Mar-03 22:38:12              |
|____________________________________________|
```

A.5 pREMUSCodec

Example .moos file As with pAcommsHandler, the above configuration file can be generated at any time with the command:

```
pREMUSCodec --example_config
```

This codec handles several of the standard REMUS CCL messages. It can be configured to generate CCL State messages at regular intervals, and it will translate incoming CCL State messages into the standard `NODE_REPORT` format used internally in the LAMSS autonomy systems. This codec allows a MOOS vehicle to perform collaborative behaviors, such as collision avoidance, with a non-MOOS, standard CCL vehicle. See section A.3.15 for an example of using pREMUSCodec.

A.6 iMOOS2SQL

This is a transponder process, which translates Status, Contact, and Track Reports into a format for interfacing the MOOS C2 with the generic Google Earth-based (geov) topside display, e.g. as shown in Fig. A.1. This module is available in moos-ivp-local (`http://oceanai.mit.edu/moos-ivp/pmwiki/pmwiki.php?n=Support.Milocal`).

A.7 pGeneralCodec

Deprecated. Do not use, rather use pAcommsHandler with no driver, no MAC, and no queueing if only encoding/decoding is desired.

A.8 pBTRCodec

Deprecated. Do not use, rather use the <`array_length`> feature of pAcommsHandler which provides the same functionality.

A.9 pCTDCodec

Deprecated. Do not use, rather use the <`max_delta`> feature of pAcommsHandler which provides all the same functionality but with much more generality.

A.10 pAcommsPoller

Deprecated. Use the MAC built into pAcommsHandler.

Glossary

acoustic networking a way of connecting underwater vehicles and other nodes wirelessly using sound waves (since light is rapidly attenuated in sea water). See also `http://gobysoft.com/doc/acomms`. 2

application a collection of code that compiles to a single exectuable unit on your operating system. synonymously (and more precise): processes or binaries. 2

asynchronous From [18]: " of, used in, or being digital communication (as between computers) in which there is no timing requirement for transmission and in which the start of each character is individually signaled by the transmitting device.". 10

autonomy architecture lossly defined, a collection of software applications and libraries that facilitate communications, decision making, timing, and other utilties needed for making robots function. Another common term for this is autonomy "middleware". 2

base class also known as subclass or child class. 4, 90

daemon an application on a Linux/UNIX machine that runs continuously in the background. the `gobyd` is a server and the Goby applications are clients.. 2, 4

derived class also known as superclass or parent class. 4, 90

LAMSS A multidiscplinary research group at the Center for Ocean Engineering (Dept. of Mechanical Engineering) at Massachusetts Institute of Technology. LAMSS focuses on collaborative marine robotics for a variety of acoustic and non acoustic sensing tasks. See `http://lamss.mit.edu`.. 40, 41, 44

protobuf From [4]: "Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data – think XML, but smaller, faster, and simpler. You define how you want your data to be structured once, then you can use special generated source code to easily write and read your structured data to and from a variety of data streams and using a variety of languages – Java, C++, or Python.". 2, 4, 6, 14, 19

publish/subscribe a method of communication between processes that is roughly analogous to authors and customers of a newspaper or newsletter. Certain people (applications) publish stories (data) that other people (applications) subscribe for and read in the newsletter. Typically applications perform both tasks, subscribing for some data and publishing others. See also `http://en.wikipedia.org/wiki/Publish/subscribe`. 2

SQL a language (in the sense of a programming language) that allows querying or accessing data from a database. For example, if I wanted to know the best baseball players in history and I had a database of players' stats, I could write in SQL the following query that would provide the data I need: `"SELECT * FROM baseball_players WHERE batting_average > 0.300 ORDER BY batting_average DESC"`. 2, 21

star topology all communications pass through a central mediator (in this case, `gobyd`) and not directly from any Goby application to another. 4

synchronous From [19]: "recurring or operating at exactly the same period.". 8

virtual A member of a base class than can be redefined in a derived class. See also `http://www.cplusplus.com/doc/tutorial/polymorphism/`. 8

Bibliography

[1] P. Newman, "The MOOS: Cross platform software for robotics research." [Online]. Available: http://www.robots.ox.ac.uk/~mobile/MOOS/wiki/pmwiki.php

[2] A. S. Huang, E. Olson, and D. C. Moore, "Lightweight communications and marshalling." [Online]. Available: http://code.google.com/p/lcm/

[3] Goby Developers, "Goby underwater autonomy project documentation." [Online]. Available: http://gobysoft.com/doc

[4] Google, "Protocol buffers." [Online]. Available: http://code.google.com/apis/protocolbuffers/

[5] SQLite Developers, "Sqlite." [Online]. Available: http://www.sqlite.org/

[6] PostgreSQL Global Development Group, "Postgresql." [Online]. Available: http://www.postgresql.org/

[7] S. Prata, *C++ Primer Plus (Fourth Edition)*, 4th ed. Indianapolis, IN, USA: Sams, 2001.

[8] B. Stroustrup, *The C++ Programming Language*, 3rd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[9] Google, "Protocol buffer basics: C++." [Online]. Available: http://code.google.com/apis/protocolbuffers/docs/cpptutorial.html

[10] ——, "Language guide." [Online]. Available: http://code.google.com/apis/protocolbuffers/docs/proto.html

[11] ——, "C++ generated code." [Online]. Available: http://code.google.com/apis/protocolbuffers/docs/reference/cpp-generated.html

[12] Kitware, "CMake." [Online]. Available: http://www.cmake.org/

[13] Emweb, "Wt, a C++ web toolkit." [Online]. Available: http://www.webtoolkit.eu/wt

[14] M. R. Benjamin, H. Schmidt, P. M. Newman, and J. J. Leonard, "Nested autonomy for unmanned marine vehicles with MOOS-IvP," *Journal of Field Robotics*, vol. 27, no. 6, pp. 834–875, 2010. [Online]. Available: http://dx.doi.org/10.1002/rob.20370

[15] T. Schneider and H. Schmidt, "Unified command and control for heterogeneous marine sensing networks," *Journal of Field Robotics*, vol. 27, no. 6, pp. 876–889, 2010. [Online]. Available: http://dx.doi.org/10.1002/rob.20346

[16] "The laboratory for autonomous marine sensing systems (LAMSS)." [Online]. Available: http://lamss.mit.edu/

[17] T. Schneider and H. Schmidt, "The Dynamic Compact Control Language: A compact marshalling scheme for acoustic communications," in *Proceedings of the IEEE Oceans Conference 2010*, Sydney, Australia, 2010.

[18] Merriam-Webster Online Dictionary, "asynchronous," 2011. [Online]. Available: http://www.merriam-webster.com/dictionary/asynchronous

[19] ——, "synchronous," 2011. [Online]. Available: http://www.merriam-webster.com/dictionary/synchronous