Goby Underwater Autonomy Project

User Manual for Version 2.0

<https://launchpad.net/goby>

Contents

1 Introduction

> *Il semble que la perfection soit atteinte non quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retrancher.* (Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.)
>
> Antoine de Saint-Exupéry, *Terre des Hommes* (1939)

1.1 What is Goby?

The Goby Underwater Autonomy Project allows your robotics software to *communicate*:

- between applications on a single robot.
- between robots (over common links, e.g. ethernet or more exotic links, e.g. acoustic modems)
- between software written by collaborators in a different autonomy architecture, like the MOOS [1] or LCM [2].

In Goby, you are free to design your applications as you like from scratch in your choice of programming language, or take advantage of the rich base applications provided in C++.

1.2 Why Goby?

Goby is designed to be easy to approach but not to limit you once you comprehend it.

- It leverages a handful of open source projects to give reliable results when you need them on your expensive robots.
- It gives you the tools to do your work without selling your soul to a particular "right way" of doing things.

1.3 Structure of this Manual

This manual is structured with the beginner material towards the beginning and the advanced material at the back. In the beginning we will guide you but by the end you are free to design your own systems, accepting or rejecting our advice. 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 from the Goby home page at `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].

If you are already familiar with other autonomy architectures and want to see what advantages Goby can add to your project, you may want to skip ahead to Chapter 4 where we explain the workings of Goby from the bottom up.

1.4 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:

- The Goby Wiki: `http://gobysoft.com/wiki`.
- Questions and Answers on Launchpad: `https://answers.launchpad.net/goby`.
- The developers' documentation: `http://gobysoft.com/doc`.
- Email the listserver goby@mit.edu. Please sign up first: `http://mailman.mit.edu/mailman/listinfo/goby`.
- Email the lead developer (T. Schneider): tes@mit.edu.

2 The Hello World example

> *"It's a dangerous business, Frodo, going out of your door," he used to say. "You step into the Road, and if you don't keep your feet, there is no knowing where you might be swept off to."*
>
> J.R.R. Tolkien, *The Fellowship of the Ring* (1954)

As well as defining a wire protocol for information exchange in different marshalling schemes, Goby provides a number of useful classes and applications that are written in C++. We feel that C++ is a good blend of elegance, speed, and expressiveness, and we hope that you will come to agree. For next few chapters we will cover these tools. If you are eager to learn about the Goby wire protocol or wish to use Goby with another programming language, please refer to Chapter 4.

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* [4]. If you are experienced in programming but new to C++, we recommend Stroustrup's *The C++ Programming Language* [5]. 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`. There's really no substitute for trying (and breaking) things yourself.

This example involves passing a single type of message (class HelloWorldMsg) from one Goby application (`hello_world1_g`[1]) to another (`hello_world2_g`). See Fig. 2.1 for a sketch of the components in this example. Since the default configuration for Goby uses multicast communications, there is no daemon or server to concern ourselves with. A good analogy to this multicast group is a meeting amongst peers. Everyone is in the same room, but no one explicitly controls the conversation. People tune in (`subscribe`) for topics (messages) that interest them and ignore those that do not.

`hello_world1_g` and `hello_world2_g` reside on the same platform, which for now we will assume is the same computer. We will learn about inter-platform communication later on.

[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.

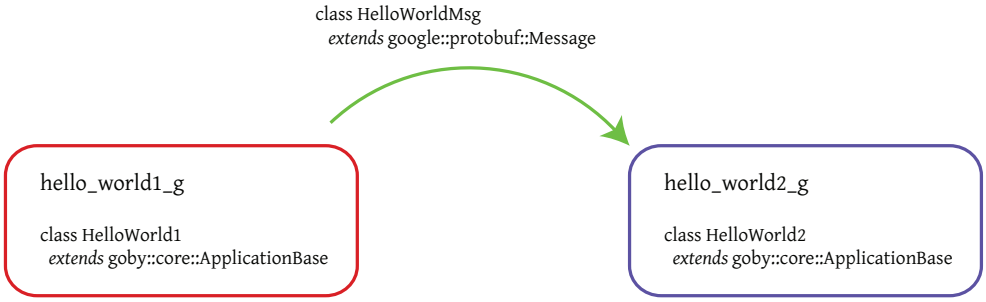

Figure 2.1: In this example `hello_world1_g` publishes messages of type `HelloWorldMsg` to the subscriber `hello_world2_g`.

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 parameters and the configuration file (we will learn about this in Chapter 3) and connects to the multicast group 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.
- a method `publish()` allowing us to publish messages to the multicast group and thereby to any subscribers of that type.
- an object `goby::glog` which acts just like std::cout and lets us write to our choice of debug logs (terminal window / text file) with fine grained control over the verbosity of the output.

2.2 Creating a simple Google Protocol Buffers Message: HelloWorldMsg

Google Protocol Buffers (protobuf) allows us to create custom messages for holding and transmitting data in a structured (object-based) fashion. These protobuf messages are similar to data structures (`structs`) available in many languages. 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 ball might be described by its diameter, color, and weight. A message describing a baseball might be written in protobuf as such:

```
// ball.proto
message Ball
{
   required double diameter = 1;
   enum Color { WHITE = 1; BLACK = 2; SPOTTED = 3 }
   optional Color color = 2;
   optional double weight = 3;
}
```

We will learn the meaning of `required`, `optional` and the sequence numbers (`=1`, `=2`, etc.) shortly.

Similarly, a sample from a CTD[2] sensor can be thought of as an object containing a number of floating point values representing salinity, temperature, pressure, etc. In the protobuf language:

```
import "units_extensions.proto"
// ctd_sample.proto
message CTDSample
{
   required double salinity = 1 [(units)="none"];
   required double temperature = 2 [(units)="degC"];
   required double depth = 3 [(units)="m"];
}
```

Goby (using protobuf objects) allows messages to be formed using this more natural object-based representation.

As you may have noticed from these examples, the protobuf language is simple with a syntax similar to that of C. Protobuf messages are written in .proto files and

[2] Conductivity-Temperature-Depth

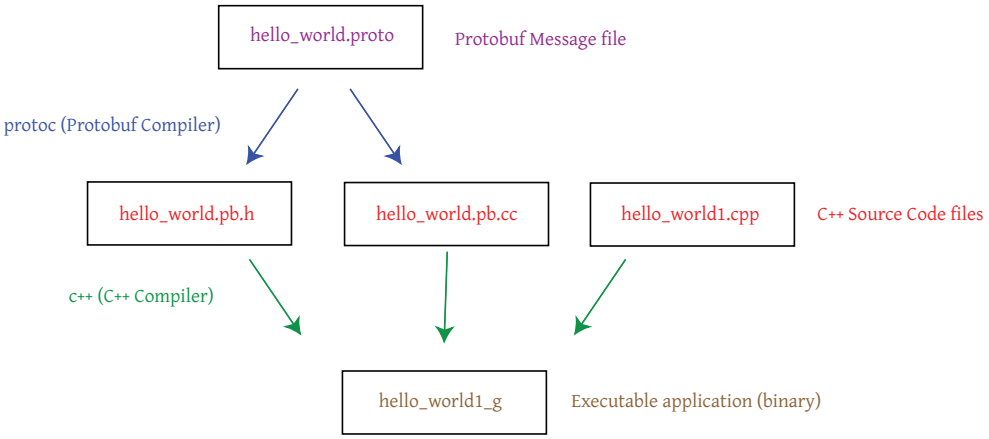

Figure 2.2: The steps of compiling `hello_world1_g`. We write the files `hello_world.proto` and `hello_world1.cpp` and the rest are generated by tools (`protoc` and `c++`.

passed to the protobuf compiler (`protoc`) which generates C++ code to pass to the C++ compiler (`c++`, `gcc` on Linux). See Fig. 2.2 for a graphical representation of this compilation process.

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 excellent Protocol Buffers tutorial [6] to get a feel for the language and usage.

As you become familiar with using protobuf, the language reference [7] will help you in creating .proto files and the generated code reference [8] 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.h"`).
- create an overloaded `loop()` method (which can do nothing).
- 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:

```
1  #include "goby/core.h"
2
3  class DoNothingApplication : public goby::core::ApplicationBase
4  {
5    void loop() {}
6  };
7
8  int main(int argc, char* argv[])
9  {
10     return goby::run<DoNothingApplication>(argc, argv);
11 }
```

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

ApplicationBase provides a pure 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 section 2.2) and publish it to all subscribers (we create a subscriber in section 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     goby::glog << "sending: " << msg << std::endl;
8     publish(msg);
9  }
```

- Line 1: `loop()` takes no arguments and returns nothing (void).
- Line 3: We declare a static integer[4] to keep track of how many times we have looped and thus print an increasing integer value.
- Line 4: We create an instantiation of `HelloWorldMsg` called `msg`.

[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()`.

- Line 5: We set the 'telegram' field of the `HelloWorldMsg` named `msg`
- Line 6: We set the 'count' field of `msg` and increment `i`.
- Line 7: We publish a human debugging log message using `goby::glog` (just like std::cout or other std::ostreams), which will be put to the terminal window in verbose mode[5].
- Line 8: Finally, we publish our message. 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 a message of 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:

```
1 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

[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)`.

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

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

```
1 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 (for another subscribed type) is in the process of being called.

In these cases, `receive_msg()` is called as soon as the blocking method returns. These conditions allow Goby applications to be single threaded. Parallelism is gained via multiple applications that communicate via messages, avoiding the extremely tricky task of data access control (using ugly things like mutexes, semaphores, etc.)

For this example, inside of `receive_msg()` we simply post the message to the debug log (`goby:glog`):

```
1 void receive_msg(const HelloWorldMsg& msg)
2 {
3    goby::glog << "received: " << msg << std::endl;
4 }
```

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

2.5 Compiling our applications using CMake

`CMake` [9], 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 examples configuration (depending on how you installed Goby, `goby/share/examples` or `/usr/share/goby/examples`). 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 pre-compiled 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. The process of steps `CMake` is using to compile our code is illustrated in Fig. 2.2.

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 two terminal windows, one for each of our "hello world" applications. Now we can launch our two applications (we can launch in either order but if we start the publisher `hello_world1_g` after the subscriber `hello_world2g`, we will miss the first few messages.). We add the "-v" flag to indicate we want verbose terminal output and the "–no_db" flag to indicate that we aren't logging data to the `goby_database`. You can always use "-h" to get help on the command line parameters.

```
1 > hello_world2_g -v --no_db
2 > hello_world1_g -v --no_db
```

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

2.6.1 hello_world1_g output

```
1 hello_world1_g (20110421T123431.832807): (Warning): Not using
2 `goby_database`. You will want to ensure you are logging your
```

```
runtime data somehow
hello_world1_g (20110421T123432.000169): sending: ### HelloWorldMsg ###
hello_world1_g (20110421T123432.000309): telegram: "hello world!"
hello_world1_g (20110421T123432.000382): count: 1
hello_world1_g (20110421T123432.000448):
hello_world1_g (20110421T123432.099503): sending: ### HelloWorldMsg ###
hello_world1_g (20110421T123432.099638): telegram: "hello world!"
hello_world1_g (20110421T123432.099707): count: 2
hello_world1_g (20110421T123432.099770):
hello_world1_g (20110421T123432.199695): sending: ### HelloWorldMsg ###
hello_world1_g (20110421T123432.199829): telegram: "hello world!"
hello_world1_g (20110421T123432.199898): count: 3
hello_world1_g (20110421T123432.199960):
```

The warning is emitted because of the "–no_db" flag. That's ok for now.

2.6.2 hello_world2_g output

```
hello_world2_g (20110421T123524.001344): received: ### HelloWorldMsg ###
hello_world2_g (20110421T123524.001899): telegram: "hello world!"
hello_world2_g (20110421T123524.001959): count: 1
hello_world2_g (20110421T123524.002012):
hello_world2_g (20110421T123524.100828): received: ### HelloWorldMsg ###
hello_world2_g (20110421T123524.100916): telegram: "hello world!"
hello_world2_g (20110421T123524.100970): count: 2
hello_world2_g (20110421T123524.101021):
hello_world2_g (20110421T123524.201065): received: ### HelloWorldMsg ###
hello_world2_g (20110421T123524.201161): telegram: "hello world!"
hello_world2_g (20110421T123524.201215): count: 3
hello_world2_g (20110421T123524.201264):
```

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

```

```
4  message HelloWorldMsg
5  {
6    required string telegram = 1;
7    required uint32 count = 2;
8  }
9
```

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

```
1  // for goby::core::ApplicationBase
2  #include "goby/core.h"
3
4  // autogenerated Protocol Buffers header
5  #include "hello_world.pb.h"
6
7  // allows us to directly output protobuf messages to streams
8  using goby::core::operator<<;
9
10 // create our Goby Application with ApplicationBase as a public base
11 class HelloWorld1 : public goby::core::ApplicationBase
12 {
13 private:
14     // loop() is a virtual method of ApplicationBase that is called
15     // at 10 Hz (by default)
16     void loop()
17         {
18             static int i = 0;
19             // create a message of type HelloWorldMsg (defined in
20             // hello_world.proto)
21             HelloWorldMsg msg;
22             // set the fields we need
23             msg.set_telegram("hello world!");
24             msg.set_count(++i);
25
26             goby::glog << "sending: " << msg << std::endl;
27
28             // publish it to `gobyd` who will send to all subscribers
29             publish(msg);
30
31         }
32 };
33
34 int main(int argc, char* argv[])
35 {
```

```
36      // start up our application (ApplicationBase will read argc and
37      // argv for us)
38      return goby::run<HelloWorld1>(argc, argv);
39  }
```

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

```
1   #include "goby/core.h"
2   #include "hello_world.pb.h"
3
4   using goby::core::operator<<;
5
6   class HelloWorld2 : public goby::core::ApplicationBase
7   {
8   public:
9       HelloWorld2()
10          {
11              // subscribe for all messages of type HelloWorldMsg
12              subscribe<HelloWorldMsg>(&HelloWorld2::receive_msg, this);
13          }
14
15  private:
16      void receive_msg(const HelloWorldMsg& msg)
17          {
18              // print to the log the newest received "HelloWorldMsg"
19              goby::glog << "received: " << msg << std::endl;
20          }
21
22      void loop()
23          { }
24  };
25
26  int main(int argc, char* argv[])
27  {
28      return goby::run<HelloWorld2>(argc, argv);
29  }
```

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

```
1   # tells CMake to generate the *.pb.h and *.pb.cc files from the *.proto
2   protobuf_generate_cpp(PROTO_SRCS PROTO_HDRS hello_world.proto)
3   include_directories(${CMAKE_CURRENT_BINARY_DIR})
```

```
4
5
6   # add these executables to the project
7   add_executable(hello_world1_g hello_world1.cpp ${PROTO_SRCS} ${PROTO_HDRS})
8   add_executable(hello_world2_g hello_world2.cpp ${PROTO_SRCS} ${PROTO_HDRS})
9
10  # and link in the goby_core library
11  target_link_libraries(hello_world1_g goby_core)
12  target_link_libraries(hello_world2_g goby_core)
13
```

3 The GPS Driver example

Man is the best computer we can put aboard a spacecraft, and the only one that can be mass produced with unskilled labor.

Werner von Braun

Robots, like people, 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 by making an application `gps_driver_g`.

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 in `depth_simulator_g`.

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` 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 de-

fines 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 (alias) that the application is to use to communicate (if different from the compiled name), and the connection details (IP addresses, ports, etc.). 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:

[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.

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

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

```
> depth_simulator_g -e
```

which gives us a good list of options to choose from. For many of these, the defaults will be fine for now.

```
base {  #  (req)
  ethernet_address: "127.0.0.1"  # primary IP address of the
                                 # interface to multicast over
                                 # (opt) (default="127.0.0.1")
  multicast_address: "239.255.7.15"  # multicast IP address to
                                     # use;
                                     # 239.252.0.0-239.255.255.255
                                     # is recommended (site-local).
                                     # See also
                                     # http://www.iana.org/assignmen
                                     # ts/multicast-addresses/multic
                                     # ast-addresses.xml (opt)
                                     # (default="239.255.7.15")
  ethernet_port: 11142  #  (opt) (default=11142)
  platform_name: "unnamed_goby_platform"  # same as self.name for
                                          # gobyd cfg (opt)
                                          # (default="unnamed_goby_p
                                          # latform")
  app_name: "myapp_g"  # default is compiled name - change this
                       # to run multiple instances (opt)
  verbosity: QUIET  # Terminal verbosity (QUIET, WARN, VERBOSE,
                    # GUI, DEBUG1, DEBUG2, DEBUG3) (opt)
                    # (default=QUIET)
  using_database: true  # True if using goby_database, false if
                        # no database is to be run (opt)
                        # (default=true)
  database_address: "127.0.0.1"  # TCP address to send database
                                 # requests on. If omitted and
                                 # using_database==true,
```

```
                                # `ethernet_address` is used (opt)
  database_port: 11142  # TCP port to send database requests on.
                        # If omitted and using_database==true,
                        # `ethernet_port` is used (opt)
  loop_freq: 10  # the frequency (Hz) used to run loop() (opt)
                 # (default=10)
}
depth:   #  (req)
```

Similarly, to see the allowed command line parameters we can run it with the `-h` (or equivalently, `--help`) flag:

```
> depth_simulator_g --help
```

which should provides output[2]:

```
Allowed options:

Typically given in depth_simulator_g configuration file,
but may be specified on the command line:
  --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
  -a [ --app_name ] arg    name to use while communicating in goby (default:
                           ./depth_simulator_g)
  -e [ --example_config ]  writes an example .pb.cfg file
  -v [ --verbose ] arg     output useful information to std::cout. -v is
                           verbosity: verbose, -vv is verbosity: debug1, -vvv
                           is verbosity: debug2, -vvvv is verbosity: debug3
  -n [ --ncurses ]         output useful information to an NCurses GUI instead
                           of stdout. If set, this parameter overrides
                           --verbose settings.
  -d [ --no_db ]           disables the check for goby_database before
                           publishing. You must set this if not running the
                           goby_database.
```

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

[2] Some of the options are removed for brevity

```
# 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":

```
1  void loop()
2      {
3          DepthReading reading;
4          // just post the depth given in the configuration file plus a small random offset
5          reading.set_depth(cfg_.depth() + (rand() % 10) / 10.0);
6
7          glogger() << reading << std::flush;
8          publish(reading);
9      }
```

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

```
1  import "goby/core/proto/header.proto";
2
3  message DepthReading
4  {
5    // time is in header
6    required Header header = 1;
7    required double depth = 2;
8  }
```

'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 the 'time' is not set, `goby::core::ApplicationBase` will set the time based on the time `publish()` was called. However 'time' should be set if the calling application has a better time stamp for the message than the publish time (for example, the time a sensor's sample was taken).

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.

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. 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
./gps_driver_g gps_driver_g.cfg -v
./depth_simulator_g depth_simulator_g.cfg -v
./node_reporter_g node_reporter_g.cfg -v
goby_database goby_database.cfg -vvv

```

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 [10] Structured Query Language (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.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);

            goby::glog << 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
  optional 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;
```

```
46
47    // time lags (in seconds) from the message Header time
48    optional double lat_time_lag = 9;
49    optional double lon_time_lag = 10;
50    optional double depth_time_lag = 11;
51    optional double altitude_time_lag = 12;
52  }
53
54  // computed from GeodeticCoordinate
55  message CartesianCoordinate
56  {
57    required double x = 1;
58    required double y = 2;
59    optional double z = 3 [default = 0]; // negative of "depth"
60  }
61
62  // all in degrees
63  message EulerAngles
64  {
65    optional double roll = 1;
66    optional double pitch = 2;
67    optional double yaw = 3; // also known as "heading"
68
69    optional SourceSensor roll_source = 4;
70    optional SourceSensor pitch_source = 5;
71    optional SourceSensor yaw_source = 6;
72
73    // time lags (in seconds) from the message Header time
74    optional double roll_time_lag = 7;
75    optional double pitch_time_lag = 8;
76    optional double yaw_time_lag = 9;
77  }
78
```

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

```
1  #ifndef NODEREPORTER20101225H
2  #define NODEREPORTER20101225H
3
4  #include "goby/core.h"
5  #include "config.pb.h"
6
7  #include "gps_nmea.pb.h"
8  #include "depth_reading.pb.h"
```

```
9
10  class NodeReporter : public goby::core::ApplicationBase
11  {
12  public:
13      NodeReporter();
14      ~NodeReporter();
15
16
17  private:
18      void create_node_report(const GPSSentenceGGA& gga,
19                              const DepthReading& depth);
20
21      void handle_depth(const DepthReading& reading)
22      {
23          create_node_report(newest<GPSSentenceGGA>(), reading);
24      }
25
26      void handle_gps(const GPSSentenceGGA& gga)
27      {
28          create_node_report(gga, newest<DepthReading>());
29      }
30
31      void loop() {}
32
33      static NodeReporterConfig cfg_;
34  };
35
36  #endif
```

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

```
1
2   #include "node_reporter.h"
3
4   #include "node_report.pb.h"
5
6   using goby::core::operator<<;
7
8   NodeReporterConfig NodeReporter::cfg_;
9
10  int main(int argc, char* argv[])
11  {
12      return goby::run<NodeReporter>(argc, argv);
13  }
```

```
14
15  NodeReporter::NodeReporter()
16      : goby::core::ApplicationBase(&cfg_)
17  {
18      // from Pressure Sensor Simulator
19      subscribe<DepthReading>(&NodeReporter::handle_depth, this);
20
21      // from GPS Driver
22      subscribe<GPSSentenceGGA>(&NodeReporter::handle_gps, this);
23  }
24
25  NodeReporter::~NodeReporter()
26  { }
27
28
29  void NodeReporter::create_node_report(const GPSSentenceGGA& gga,
30                                        const DepthReading& depth_reading)
31  {
32      if(!(gga.IsInitialized() && depth_reading.IsInitialized()))
33      {
34          goby::glog << warn << "need both GPSSentenceGGA and DepthReading "
35                     << "message to proceed" << std::endl;
36          return;
37      }
38
39      goby::glog << gga << "\n" << depth_reading << std::endl;
40
41
42      // make an abstracted position and pose aggregate from the newest
43      // readings for consumption by other processes
44      NodeReport report;
45
46      // use the time from the GGA message as the base message time
47      report.mutable_header()->set_time(gga.header().time());
48      report.set_name(cfg_.base().platform_name());
49  //    report.set_type(global_cfg().self().type());
50
51      GeodeticCoordinate* global_fix = report.mutable_global_fix();
52      global_fix->set_lat(gga.lat());
53      global_fix->set_lon(gga.lon());
54
55      // we set message time from GPS GGA, so no lag
56      global_fix->set_lat_time_lag(0);
57      global_fix->set_lon_time_lag(0);
58
59      global_fix->set_lat_source(GPS);
```

```
60      global_fix->set_lon_source(GPS);
61
62      // set the depth sensor data
63      global_fix->set_depth(depth_reading.depth());
64      global_fix->set_depth_source(SIMULATION);
65
66      using namespace boost::posix_time;
67      using goby::util::as;
68      time_duration lag = as<ptime>(gga.header().time())-as<ptime>(depth_reading.header().time());
69
70      global_fix->set_depth_time_lag(lag.total_nanoseconds()/1.0e9);
71
72      // TODO(tes): compute the local coordinates
73
74      // in a better world we would want data for altitude, speed and
75      // Euler angles too!
76      goby::glog << report << std::flush;
77
78      publish(report);
79
80  }
```

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

```
1   import "goby/protobuf/header.proto";
2
3   message NMEASentence
4   {
5     // e.g. "GP"
6     required string talker_id = 1;
7     // e.g. "GGA"
8     required string sentence_id = 2;
9     // e.g. 71
10    required uint32 checksum = 3;
11    // e.g. part[0] = $GPGGA
12    //      part[1] = 123519
13    //      part[2] = 4807.038
14    //      part[3] = N
15    // and so on
16    repeated string part = 4;
17  }
18
19  message GPSSentenceGGA
20  {
```

```
21    // time is in header
22    required Header header = 1;
23    required NMEASentence nmea = 2;
24
25    // decimal degrees
26    required double lat = 3;
27    required double lon = 4;
28
29    enum FixQuality
30    {
31      INVALID = 0;
32      GPS_FIX = 1;
33      DGPS_FIX = 2;
34      PPS_FIX = 3;
35      REAL_TIME_KINEMATIC = 4;
36      FLOAT_RTK = 5;
37      ESTIMATED = 6;
38      MANUAL_MODE = 7;
39      SIMULATION_MODE = 8;
40    }
41    required FixQuality fix_quality = 5;
42    required uint32 num_satellites = 6;
43    required float horiz_dilution = 7;
44    required double altitude = 8;
45    required double geoid_height = 9;
46  }
```

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

```
1   #ifndef GPSDRIVER20101014H
2   #define GPSDRIVER20101014H
3
4   #include "goby/core.h"
5   #include "goby/util/time.h"
6   // for serial driver
7   #include "goby/util/linebasedcomms.h"
8   #include "config.pb.h"
9
10  // forward declare (from gps_nmea.proto)
11  class NMEASentence;
12  class GPSSentenceGGA;
13
14  class GPSDriver : public goby::core::ApplicationBase
15  {
```

```
16  public:
17      GPSDriver();
18      ~GPSDriver();
19
20
21  private:
22      void loop();
23      boost::posix_time::ptime nmea_time2ptime(const std::string& nmea_time);
24      void string2nmea_sentence(std::string in, NMEASentence* out);
25      void set_gga_specific_fields(GPSSentenceGGA* gga);
26
27      goby::util::SerialClient serial_;
28      static GPSDriverConfig cfg_;
29  };
30
31  // very simple exception classes
32  class bad_nmea_sentence : public std::runtime_error
33  {
34    public:
35    bad_nmea_sentence(const std::string& s)
36        : std::runtime_error(s)
37      { }
38  };
39
40  class bad_gga_sentence : public std::runtime_error
41  {
42    public:
43    bad_gga_sentence(const std::string& s)
44        : std::runtime_error(s)
45      { }
46  };
47
48
49  #endif
```

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

```
1   #include "gps_driver.h"
2
3   #include "gps_nmea.pb.h"
4
5   #include "goby/util/binary.h" // for goby::util::hex_string2number
6   #include "goby/util/string.h" // for goby::util::as
7
```

```
8  using goby::core::operator<<;
9
10 GPSDriverConfig GPSDriver::cfg_;
11
12 int main(int argc, char* argv[])
13 {
14     return goby::run<GPSDriver>(argc, argv);
15 }
16
17 GPSDriver::GPSDriver()
18     : goby::core::ApplicationBase(&cfg_),
19       serial_(cfg_.serial_port(), cfg_.serial_baud(), cfg_.end_line())
20 {
21     serial_.start();
22 }
23
24 GPSDriver::~GPSDriver()
25 {
26     serial_.close();
27 }
28
29 void GPSDriver::loop()
30 {
31     std::string in;
32     while(serial_.readline(&in))
33     {
34         goby::glog << "raw NMEA: " << in << std::flush;
35
36         // parse
37         NMEASentence nmea;
38         try
39         {
40             string2nmea_sentence(in, &nmea);
41         }
42         catch (bad_nmea_sentence& e)
43         {
44             goby::glog << warn << "bad NMEA sentence: " << e.what()
45                        << std::endl;
46         }
47
48         if(nmea.sentence_id() == "GGA")
49         {
50             goby::glog << "This is a GGA type message." << std::endl;
51
52             // create the message we send on the wire
53             GPSSentenceGGA gga;
```

```
54              // copy the raw message (in case later users want to do their
55              // own parsing)
56              gga.mutable_nmea()->CopyFrom(nmea);
57
58              try
59              {
60                  set_gga_specific_fields(&gga);
61
62                  // parse the time stamp
63                  using goby::util::as;
64                  gga.mutable_header()->set_time(as<std::string>(nmea_time2ptime(nmea.part(1))));
65                  goby::glog << gga << std::flush;
66
67                  publish(gga);
68              }
69              catch(bad_gga_sentence& e)
70              {
71                  goby::glog << warn << "bad GGA sentence: " << e.what()
72                             << std::endl;
73              }
74
75          }
76
77      }
78  }
79
80
81  // from http://www.gpsinformation.org/dale/nmea.htm#GGA
82  // GGA - essential fix data which provide 3D location and accuracy data.
83  // $GPGGA,123519,4807.038,N,01131.000,E,1,08,0.9,545.4,M,46.9,M,,*47
84  // Where:
85  //      GGA          Global Positioning System Fix Data
86  //      123519       Fix taken at 12:35:19 UTC
87  //      4807.038,N   Latitude 48 deg 07.038' N
88  //      01131.000,E  Longitude 11 deg 31.000' E
89  //      1            Fix quality: 0 = invalid
90  //                                1 = GPS fix (SPS)
91  //                                2 = DGPS fix
92  //                                3 = PPS fix
93  //            4 = Real Time Kinematic
94  //            5 = Float RTK
95  //                                6 = estimated (dead reckoning)
96  //                                    (2.3 feature)
97  //            7 = Manual input mode
98  //            8 = Simulation mode
99  //      08           Number of satellites being tracked
```

```
100 //      0.9          Horizontal dilution of position
101 //      545.4,M      Altitude, Meters, above mean sea level
102 //      46.9,M       Height of geoid (mean sea level) above WGS84
103 //                       ellipsoid
104 //      (empty field) time in seconds since last DGPS update
105 //      (empty field) DGPS station ID number
106 //      *47          the checksum data, always begins with *
107 // If the height of geoid is missing then the altitude should be suspect.
108 // Some non-standard implementations report altitude with respect to the
109 // ellipsoid rather than geoid altitude. Some units do not report negative
110 // altitudes at all. This is the only sentence that reports altitude.
111
112 void GPSDriver::set_gga_specific_fields(GPSSentenceGGA* gga)
113 {
114     using goby::util::as;
115     const NMEASentence& nmea = gga->nmea();
116
117     const std::string& lat_string = nmea.part(2);
118
119     if(lat_string.length() > 2)
120     {
121         double lat_deg = as<double>(lat_string.substr(0, 2));
122         double lat_min = as<double>(lat_string.substr(2, lat_string.size()));
123         double lat = lat_deg + lat_min / 60;
124         gga->set_lat((nmea.part(3) == "S") ? -lat : lat);
125     }
126     else
127     {
128         throw(bad_gga_sentence("invalid latitude field"));
129     }
130
131     const std::string& lon_string = nmea.part(4);
132     if(lon_string.length() > 2)
133     {
134         double lon_deg = as<double>(lon_string.substr(0, 3));
135         double lon_min = as<double>(lon_string.substr(3, nmea.part(4).size()));
136         double lon = lon_deg + lon_min / 60;
137         gga->set_lon((nmea.part(5) == "W") ? -lon : lon);
138     }
139     else
140         throw(bad_gga_sentence("invalid longitude field: " + nmea.part(4)));
141
142     switch(goby::util::as<int>(nmea.part(6)))
143     {
144         default:
145         case 0: gga->set_fix_quality(GPSSentenceGGA::INVALID); break;
```

```
146         case 1: gga->set_fix_quality(GPSSentenceGGA::GPS_FIX); break;
147         case 2: gga->set_fix_quality(GPSSentenceGGA::DGPS_FIX); break;
148         case 3: gga->set_fix_quality(GPSSentenceGGA::PPS_FIX); break;
149         case 4: gga->set_fix_quality(GPSSentenceGGA::REAL_TIME_KINEMATIC);
150             break;
151         case 5: gga->set_fix_quality(GPSSentenceGGA::FLOAT_RTK); break;
152         case 6: gga->set_fix_quality(GPSSentenceGGA::ESTIMATED); break;
153         case 7: gga->set_fix_quality(GPSSentenceGGA::MANUAL_MODE); break;
154         case 8: gga->set_fix_quality(GPSSentenceGGA::SIMULATION_MODE); break;
155     }
156
157     gga->set_num_satellites(goby::util::as<int>(nmea.part(7)));
158     gga->set_horiz_dilution(goby::util::as<float>(nmea.part(8)));
159     gga->set_altitude(goby::util::as<double>(nmea.part(9)));
160     gga->set_geoid_height(goby::util::as<double>(nmea.part(11)));
161 }
162
163 // converts a NMEA0183 sentence string into a class representation
164 void GPSDriver::string2nmea_sentence(std::string in, NMEASentence* out)
165 {
166
167     // Silently drop leading/trailing whitespace if present.
168     boost::trim(in);
169
170     // Basic error checks ($, empty)
171     if (in.empty())
172       throw bad_nmea_sentence("message provided.");
173     if (in[0] != '$')
174         throw bad_nmea_sentence("no $: '" + in + "'.");
175     // Check if the checksum exists and is correctly placed, and strip it.
176     // If it's not correctly placed, we'll interpret it as part of message.
177     // NMEA spec doesn't seem to say that * is forbidden elsewhere?
178     // (should be)
179     if (in.size() > 3 && in.at(in.size()-3) == '*') {
180       std::string hex_csum = in.substr(in.size()-2);
181       int cs;
182       if(goby::util::hex_string2number(hex_csum, cs))
183           out->set_checksum(cs);
184       in = in.substr(0, in.size()-3);
185     }
186
187     // Split string into parts.
188     size_t comma_pos = 0, last_comma_pos = 0;
189     while((comma_pos = in.find(",", last_comma_pos)) != std::string::npos)
190     {
191         out->add_part(in.substr(last_comma_pos, comma_pos-last_comma_pos));
```

```
192
193         // +1 moves us past the comma
194         last_comma_pos = comma_pos + 1;
195     }
196     out->add_part(in.substr(last_comma_pos));
197
198     // Validate talker size.
199     if (out->part(0).size() != 6)
200         throw bad_nmea_sentence("bad talker length '" + in + "'.");
201
202     // GP
203     out->set_talker_id(out->part(0).substr(1, 2));
204     // GGA
205     out->set_sentence_id(out->part(0).substr(3));
206
207 }
208
209
210 // converts the time stamp used by GPS messages of the format HHMMSS.SSS
211 // for arbitrary precision fractional
212 // seconds into a boost::ptime object (much more usable class
213 // representation of for dates and times)
214 // *CAVEAT* this assumes that the message was received "today" for the
215 // date part of the returned ptime.
216 boost::posix_time::ptime GPSDriver::nmea_time2ptime(const std::string& mt)
217 {
218     using namespace boost::posix_time;
219     using namespace boost::gregorian;
220
221     // must be at least HHMMSS
222     if(mt.length() < 6)
223         return ptime(not_a_date_time);
224     else
225     {
226         std::string s_hour = mt.substr(0,2), s_min = mt.substr(2,2),
227             s_sec = mt.substr(4,2), s_fs = "0";
228
229         // has some fractional seconds
230         if(mt.length() > 7)
231             s_fs = mt.substr(7); // everything after the "."
232
233         try
234         {
235             int hour = boost::lexical_cast<int>(s_hour);
236             int min = boost::lexical_cast<int>(s_min);
237             int sec = boost::lexical_cast<int>(s_sec);
```

```
238             int micro_sec = boost::lexical_cast<int>(s_fs)*
239                 pow(10, 6-s_fs.size());
240
241             return (ptime(date(day_clock::universal_day()),
242                           time_duration(hour, min, sec, 0)) +
243                     microseconds(micro_sec));
244         }
245         catch (boost::bad_lexical_cast&)
246         {
247             return ptime(not_a_date_time);
248         }
249     }
250 }
251
```

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

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

```
1 base
2 {
3     platform_name: "AUV-1"
4     loop_freq: 1
5 }
6
7 depth: 10
```

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

```
1 base
2 {
3     platform_name: "AUV-1"
4     loop_freq: 1
5 }
6 serial_port: "/tmp/ttyFAKE"
```

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

```
1 base
2 {
```

```
    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 *Goby Underpinnings*

The previous chapters talked about the C++ applications and classes included with Goby to make our lives easy. However, none of these are required to use Goby. In this chapter we will throw away these tools for now and work from the ground up.

4.1 Design Considerations

5 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` [11] based configuration, launch, and runtime manager.

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

Glossary

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 [?]: " 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.". 11

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, 3

base class also known as subclass or child class. 5

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.. 4

derived class also known as superclass or parent class. 5

multicast A communications scheme where one application sends messages to a group of applications. Multicast is designed such that the sender only sends once and the network topology is responsible for replicating it as necessary. In general, multicast refers to IP (internet protocol) multicast. In Goby, we use encapsulated Pragmatic General Multicast (PGM), which provides a reliability layer to UDP multicast.. 4, 5

platform Used to refer to a physical robotic entity, such as an AUV, a topside computer on board a ship, or a buoy.. 4

protobuf From [?]: "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.". 6, 7, 17, 23

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"`. 25

synchronous From [?]: "recurring or operating at exactly the same period.". 9

virtual A member of a base class than can be redefined in a derived class. See also `http://www.cplusplus.com/doc/tutorial/polymorphism/`. 9

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

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

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

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

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

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

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

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