Goby Underwater Autonomy Project

User Manual for Version 2.1.11.

Released on 2018.11.08.
<https://launchpad.net/goby>

Contents

Contents 1

1 Introduction 3
- 1.1 What is Goby? . . . 3
- 1.2 Structure of this Manual . . . 4
- 1.3 Prerequisites . . . 4
- 1.4 Getting the Code . . . 4
- 1.5 Changes / incompatiblities with version 1 . . . 5
- 1.6 How to get help . . . 6

2 Goby-Acomms 7
- 2.1 Introduction . . . 7
- 2.2 Dynamic Compact Control Language: DCCL . . . 9
- 2.3 Time dependent priority queuing: Queue . . . 12
- 2.4 Time Division Multiple Access (TDMA) Medium Access Control (MAC): AMAC . . . 16
- 2.5 Abstract Acoustic (or other slow link) Modem Driver: ModemDriver 20

3 Goby Common Modules 25
3.1 Goby Common Applications 25
3.2 Liaison 27
3.3 Gateway Applications 29

4 Goby MOOS Modules 30
4.1 Goby MOOS Applications 30
4.2 pTranslator 32
4.3 Translator techniques 36
4.4 pAcommsHandler 37
4.5 MOOS Plugins for Goby Liaison 41
4.6 Migrating from Version 1 to Version 2 49
4.7 iFrontSeat 50
4.8 iCommander 60
4.9 pREMUSCodec 60

5 What's next 61

Glossary 62

Bibliography 63

1 Introduction

1.1 What is Goby?

The Goby Underwater Autonomy Project is an autonomy architecture tailored for marine robotics with a focus on intervehicle communication.

Currently, Goby provides several libraries, with a primary focus on Goby-Acomms:

- Goby-Acomms: The Goby Acoustic Communications library (`goby-acomms`) has been provided since Version 1.0. See the Developers' documentation for details on these library and the various modules it contains at [1]. Users of the MOOS application `pAcommsHandler` should see Chapter 4.
- Goby-Common: A library providing tools for the rest of Goby to use. For release 2.0, Goby-Common provides a debug logging tool (`goby::glog`), various utilities (e.g. time functions), and the groundwork for an autonomy architecture. The Goby-Common architecture that ties together various marshalling schemes (Google Protocol Buffers, MOOS, LCM, etc.) and provides a message passing middleware based on ZeroMQ (for ethernet) and Goby-DCCL (for acoustic communications and other "slow links"). Goby-Common will be provided in a more complete (and documented) form in release version 3.0.
- Goby-Util: A utility library that provide functions for dealing with type conversions (`goby::util::as<>()`), binary conversions, etc. This library is intended to be small, as Goby makes use of the C++ Standard Library and Boost for most utility tasks.
- Goby-PB: The Google Protocol Buffers / C++ implementation of Goby-Common. Like much of Goby-Common, this will be finalized in release 3.0, but is preliminarily provided in release 2.0 to support tools such as `goby_liaison`.
- Goby-MOOS: The MOOS [2] / C++ implementation of Goby-Common. This library provides translator tools from MOOS messages (CMOOSMsg) to and from the Google Protobuf messages used internally. It provides a Goby-Acomms modem driver for the MOOS-IvP uField toolbox [3], allowing multivehicle network simulation without acoustic modem hardware. See also [4] for more on MOOS-IvP.

1.2 Structure of this Manual

This manual covers general use of the Goby libraries and the applications provided with them. If you are interested in a complete API and further details, please read the online Developers' documentation at [1] In fact, you may want to go download and install Goby now before reading further: https://launchpad.net/goby.

1.3 Prerequisites

Goby (for both DCCL and the various external APIs) makes significant use of the Google Protocol Buffers (protobuf) mechanism for serializing structured data. This library is very well documented and is widely adopted in numerous open source projects. Please take a few moments to familiarize yourself with the project here: https://developers.google.com/protocol-buffers/docs/overview.

1.4 Getting the Code

By far the easiest way to get Goby is to use any currently supported Ubuntu distribution (see http://en.wikipedia.org/wiki/List_of_Ubuntu_releases#Version_timeline), and install it using apt-get:

```
sudo apt-add-repository ppa:goby-dev/ppa
sudo apt-get update
sudo apt-get install libgoby2
```

and then, optionally, install one or more additional packages:

```
# for the core applications
sudo apt-get install goby2-apps
# for the MOOS applications
sudo apt-get install goby2-moos
# for the developer header files
sudo apt-get install libgoby2-dev
# for the documentation
sudo apt-get install goby2-doc
# for the unit tests
sudo apt-get install goby2-test
```

You can also compile Goby from source using the bazaar version control software:

```
bzr co lp:goby/2.0
```

The dependencies for Goby are minimally

- Google Protocol Buffers (see https://code.google.com/p/protobuf/)
- Boost (see http://www.boost.org/).

Certain optional libraries and/or functionality require additional dependencies:

- ZeroMQ for the communications applications: `goby_modemdriver`, `goby_bridge`, `goby_file_transfer`, `goby_store_server`, `goby_rudics_shore`.
- MOOS or MOOS 10 (see http://themoos.org/) and PROJ.4 for the Goby MOOS library and applications (see Chapter 4).
- Wt for web-browser based GUI applications (`goby_liaison`)
- Crypto++ for encrypting DCCL messages.
- GMP for the Iridium driver.
- NCurses for the debugging terminal GUI.

1.5 Changes / incompatiblities with version 1

Goby version 2 has been significantly reworked to based on the valuable feedback from users of version 1 and our experience in numerous field trials. The major changes include:

- Goby-Acomms:
  - DCCL has been rewritten to be based on Google Protocol Buffers (no more XML). This means much richer type support and cleaner code. Also, any field or message can be encoded using a user-defined codec for that particular job.
  - WHOI Micro-Modem driver (MMDriver) supports all the modem's major functionality: ping, LBL ranging, data, communications statistics, user mini-packet. The DriverBase interface for writing custom modem drivers has been streamlined.
  - AMAC is simpler and more intuitive: now it is basically a std::list plus a timer.

  - Many fewer dependencies: only required are Boost and Google Protocol Buffers (which compile nicely on nearly all platforms).

Because of these substantial changes, full backwards compatibility support is provided for users of MOOS (pAcommsHandler) since that community was the primary user base of that release. Other users must migrate code from version 1 before using version 2. Help on migrating from release 1 is given in Chapter 4.6.

1.6 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. The Wiki: `http://gobysoft.com/wiki`.
3. Questions and Answers on Launchpad: `https://answers.launchpad.net/goby`.
4. The developers' documentation: `http://gobysoft.com/doc/2.0`.
5. Email the listserver goby@mit.edu. Please sign up first: `http://mailman.mit.edu/mailman/listinfo/goby`.
6. Email the lead developer (T. Schneider): tes@mit.edu.

2 Goby-Acomms

2.1 Introduction

2.1.1 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).

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, Goby-Acomms provides an interface to the Dynamic Compact Control Language (DCCL[1]) encoder/decoder.

For the interested reader, the publications listed in the Developers' documentation [1] give a more in-depth look at the problem.

2.1.2 Goby contributions to the solution

Goby is hardly a complete solution to this problem, but it's a start. It provides four key components (listed in order from closest to the application to closest to the physical link) intended to address the limits of traditional networking systems in light of the extreme bandwidth and latency constraints of underwater links:

1. The Dynamic Compact Control Language (DCCL) (section 2.2) is a marshalling (or synonymously serialization) scheme that creates highly compressed small messages suitable for sending over links with very low maximum transmission units (order of 10s to 100s of bytes) such as typical underwater acoustic modems. DCCL provides greater efficiency (i.e. smaller messages) than existing marine (CCL, Inter-Module Communication) and non-marine (Google Protobuf, ASN.1, boost::serialization, etc.) techniques by pre-sharing all structural information and bounding message fields to minimum and maximum values (which then create messages of any bit size, not limited by integer multiples of octets such as int16, int32, etc.). The

[1] the name comes from the original CCL written by Roger Stokey for the REMUS AUVs, but with the ability to dynamically reconfigure messages based on mission need. If desired, DCCL can be configured to be backwards compatible with a CCL network using CCL message number 32

DCCL structure language is independent of a given programming language and provides compile-time type safety and syntax checking, both of which are important for fielding complex robotic systems. Finally, DCCL is extensible to allow user-provided source encoders for any given field or message type.

2. The transport layer of Goby-Acomms provides time dynamic priority queuing (Goby-Queue, section 2.3). In our experience, acoustic links on fielded vehicles have been typically run at over-capacity; that is, there are more data to send than will ever be send over the link. Thus, the data that are to be sent must be chosen in some fashion. Historically, priority queues are widely used to send more valuable data first. However, different types of data also have different time sensitivities, which Goby-Queue recognizes via the use of a (clock time based) time-to-live parameter. Finally, the demand for a given type of data can increase over time since last receiving a message of that type. Goby-Queue extends the traditional priority queue concept to balance these various demands and send the most valuable data under this set of metrics.

3. Acoustic modems such as the WHOI Micro-Modem do not provide any shared access of the acoustic channel. Coordinating shared access can be accomplished by assigning slots of time in which each vehicle can transmit, which is the time-division multiple access (TDMA) flavor of medium access control (MAC). The Goby-Acomms acoustic MAC (AMAC), section 2.4 extends the basic TDMA idea to include passive (i.e. no data overhead) auto-discovery of vehicles in a small, equally time-shared network. Thus, AMAC simplifies the amount of pre-deployment configuration required to configure small networks of AUVs.

4. The Goby ModemDriver (section 2.5) provides an abstract interface for acoustic modems (and other "slow link" devices, such as satellite modems), as there is no standard for interfacing to such devices. Many acoustic modems provide functionality beyond the strict definition of a modem (which is defined as sending data from one point to another). Examples of these extra features include navigation (long base line or LBL, ultra-short base line or USBL) and ranging measurements ("pings"). Goby ModemDriver allows an application intent only on transmitting data to operate on any implemented modem without concerning itself with the details of that device. On the other hand, if the application needs to use some of the extra features, it can do so via a set of well-defined extensions.

These components are loosely coupled. For example, it is possible to use the ModemDriver (with or without AMAC) to send encoded messages of any origin. You can also use DCCL without any of the other components. However, Queue requires DCCL (it does not queue other types of messages). Thus, you can design systems using only one, several, or all of the components of Goby, as you need and see fit.

2.2 Dynamic Compact Control Language: DCCL

DCCL allows you to take object based "messages" (similar to C structs) defined in the Google Protocol Buffers language and extend them to be more strictly bounded. It provides a set of default encoders for these bounded Protocol Buffers messages (now called DCCL messages) to provide a more minimal encoding than the default Protocol Buffers encoding (which is reasonably decent already, but still has too much overhead for extremely slow links).

Thus, broadly speaking, DCCL provides an alternative (more compact and extensible) encoding scheme for Google Protocol Buffers, at some cost of additional development time and the requirement that the sender and receiver share the exact .proto definition file (which the normal protobuf encoder does not require). In our experience, this extra effort is worth it for acoustic (and other "very slow link" networks, such as satellite).

2.2.1 Configuration: DCCLConfig

Configuration of individual DCCL messages (the vast majority of DCCL configuration) is done within the .proto definition. All the non-message specific available configuration for `goby::acomms::DCCLCodec` is given in its TextFormat form as:

```
crypto_passphrase: "twinkletoes%24"
```

- `crypto_passphrase`: If provided, this preshared key is used to encrypt the body of all messages using AES (Rijndael) encryption. Omit this field to turn off encryption. Note that the contents of messages received by nodes with the wrong encryption key are undefined, and such failure is not currently detected.

2.2.2 Configuration: Designing DCCL messages using Protocol Buffers Extensions

A full guide to designing DCCL messages is given at http://gobysoft.com/doc/2.0/acomms_dccl.html along with a full list of the DCCL extensions to the Google Protobuf `MessageOptions` (i.e. `(dccl.msg).*`) and `FieldOptions` (i.e. `(dccl.field).*`). Therefore, we will not replicate that information here. However, we will give a broad overview of the DCCL configuration.

DCCL messages are protobuf messages with "invisible" extensions. By "invisible," we mean that DCCL messages can be compiled by the standard protobuf compiler (`protoc`) without requiring the Goby-Acomms library. This allows DCCL messages to be shared with users that do not need the functionality of DCCL (e.g. are only using traditional IP networks), but need to communicate with groups that need the additional compression afforded by DCCL. The goal is to break down the barriers for using acoustic links on robotic systems, while still maintaining the efficiency necessary for effective use of these highly restricted links.

A simple, but realistic, protobuf message might look like this:

```
1  package example;
2
3  message MinimalStatus
4  {
5
6    required double time = 1;
7    required int32 source = 2;
8    required int32 dest = 3;
9    required double x = 4;
10   required double y = 5;
11   required double depth = 6;
12 }
```

The field numbers (e.g. `1` in `time = 1`) are used by the default protobuf encoding (but not by DCCL) to allow backwards compatibility of messages. DCCL requires that both sender and receiver have the identical message definition (.proto file), so for our purpose you just need to make sure no two fields share the same field number. These numbers have no effect in the DCCL encoding.

A priori, we know certain physical bounds on the message fields. These can be conservative (if a field goes out-of-bounds, the receiver sees it as empty; that is,

`.has_field() == false`), but even conservative bounds will often make a field consume far fewer bytes than the system equivalent.

Integer (`(u)int32`, `(u)int64`) fields take a `max` and `min` value[2], and DCCL creates the smallest (bit-sized) integer than can hold that value. For reals (`float` and `double`), an additional `precision` value is provided: this represents the number of decimal digits of precision to preserve (negative values are also allowed). Thus, `precision=1` means round to the nearest tenth, `precision=-2` means round to the nearest hundred.

Booleans (`bool`) and enumerations (`enum`) are automatically bounded their nature, and require no additional configuration. Strings (`string`) (which are generally discouraged on an acoustic link since they tend to be sparse) are bounded by a maximum length. Similarly, `bytes` are pre-encoded data that are passed through unmodified in DCCL. Applying these bounds to the example message above (along with the required .proto file imports) yields:

```
1  import "dccl/protobuf/option_extensions.proto";
2
3  package example;
4
5  message MinimalStatus
6  {
7    option (dccl.msg).id = 21;
8    option (dccl.msg).max_bytes = 10;
9
10   required double time = 1 [(dccl.field).codec="_time",
11                             (dccl.field).in_head=true];
12
13   required int32 source = 2 [(dccl.field).max=31,
14                              (dccl.field).min=0,
15                              (dccl.field).in_head=true];
16
17   required int32 dest = 3 [(dccl.field).max=31,
18                            (dccl.field).min=0,
19                            (dccl.field).in_head=true];
20
21   required double x = 4 [(dccl.field).max=10000,
22                          (dccl.field).min=-10000,
23                          (dccl.field).precision=1];
24
25   required double y = 5 [(dccl.field).max=10000,
```

[2] the DCCL bounds must be a subset of the system type's bounds

```
                          (dccl.field).min=-10000,
                          (dccl.field).precision=1];

   required double depth = 6 [(dccl.field).max=6400,
                              (dccl.field).min=0,
                              (dccl.field).precision=-1];
}
```

The option "in_head" tags the field as belonging in the user header. The only distinction between the header and body of a DCCL message is for encryption: the body is encrypted but the header is not (it is used as the nonce). The option codec allows a different DCCL codec to be used than the default for that field type (_time is a codec that encodes time of day to the nearest second assuming that messages are received within 12 hours of transmission). If you wish to write custom encoders, see the DCCLTypedFixedFieldCodec class in the Developers' documentation.

2.3 Time dependent priority queuing: Queue

Goby-Queue manages a queue for each DCCL message. 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 modem 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/2.0/acomms_queue.html for more information.

2.3.1 Configuration: QueueManagerConfig

The configuration options for goby::acomms::QueueManager are:

```
modem_id: 1
message_entry {
```

```
    protobuf_name: ""
    ack: true
    blackout_time: 0
    max_queue: 100
    newest_first: true
    ttl: 1800
    value_base: 1
    manipulator:
    role {
      type:
      setting: FIELD_VALUE
      field: ""
      static_value:
    }
  }
  on_demand_skew_seconds: 1
  minimum_ack_wait_seconds: 0
```

- `modem_id`: A unique integer value for this particular vehicle (like a MAC address). Should be as small as possible for optimal bounding of the source and destination fields of the message. `0` is reserved for broadcast (analogous to 255.255.255.255 for IPv4).
- `message_entry`: Configures the QueueManager to queue this DCCL type:
  - `protobuf_name`: String representing the DCCL message to manipulate. Messages are named the same as `google::protobuf::Descriptor::full_name()`, which is the `package` followed by the message name, separated by dots: e.g. "example.MinimalStatus" for the message shown in Section 2.2.
  - `ack`: Whether an acoustic acknowledgment should be requested for messages sent from this queue. If `ack` is true, messages will not be dequeued until a positive `ack` is received (or it expires due to exceeding the `ttl`).
  - `blackout_time`: Minimum number of seconds allowed between sending messages from this queue.
  - `max_queue`: Allowed size of the queue before overflow. If `newest_first` is true, the oldest elements are removed upon overflow, otherwise the newest elements are. `0` is a special value signifying infinity (no maximum).

- `newest_first`: true (true=FILO, false=FIFO) whether to send newest messages in the queue first (FILO) or not (FIFO).
- `ttl`: the time in seconds a message lives after its creation before being discarded. This time-to-live also factors into the growth in priority of a queue. see `value_base` for the main discussion on this. 0 is a special value indicating infinite life (i.e. `ttl` = 0 is effectively the same as `ttl` = ∞)
- `value_base`: base priority value for this message queue. priorities are calculated on a request for data by the modem (to send a message). The queue with the highest priority (and isn't in blackout) is chosen. The actual priority (P) is calculated by $P(t) = V_{base}\frac{(t-t_{last})}{ttl}$ where V_{base} is the value set here, t is the current time (in seconds), t_{last} is the time of the last send from this queue, and ttl is the `ttl` option. Essentially, a message with low `ttl` will become effective quickly again after a sent message (the priority line grows faster).
- `manipulator`: One or more manipulators to apply to the queuing of this message.
  * `NO_MANIP`: A do nothing (noop) manipulator. Same as omitting this field.
  * `NO_QUEUE`: Do not queue this message when generated on this node (but messages will still be received (dequeued).
  * `NO_DEQUEUE`: Do not dequeue (receive) this message on this node (but messages will be queued). When both `NO_QUEUE` and `NO_DEQUEUE` are set, there isn't much point to having the message loaded at all.
  * `LOOPBACK`: Dequeue all instances of this message immediately upon queuing. The message is still queued and sent to its addressed destination. Often used with `PROMISCUOUS`.
  * `ON_DEMAND`: A special (advanced) feature where QueueManager assumes this queue is always full and asks for data immediately from the application upon request from the modem side. Useful for ensuring time sensitive data does not get stale.
  * `LOOPBACK_AS_SENT`: Like loopback, but rather than dequeuing upon queuing, this manipulator dequeues a copy locally upon a data request from the modem. Often used with `PROMISCUOUS`.
  * `PROMISCUOUS`: Dequeue all messages of this type even if this `modem_id` does not match the destination address.
  * `NO_ENCODE`: Same as `NO_QUEUE`, provided for backwards compatibility with Goby v1.

        * `NO_DECODE`: Same as `NO_DEQUEUE`, provided for backwards compatibility with Goby v1.
    - role: allows the assignment of a field in the DCCL message to a particular role. This takes the place of a fixed header that strictly hierarchical protocols might use.
        * `type`: the type of this role. Valid values are `SOURCE_ID` (which represents the source address of this message), `DESTINATION_ID` (the destination address of this message), `TIMESTAMP` (the time this message was created: used for the `ttl` calculation).
        * `setting`: how is the value of this role obtained: `FIELD_VALUE` (read this role's value from the message field given by `field`) or `STATIC` (read this value from this configuration's `static_value` field).
        * `field`: If `setting == FIELD_VALUE`, the field name (e.g. `dest`) in the message whose contents should be used for in this role. Do not set this if using `setting == STATIC`
        * `static_value`: The static value to use for `setting == STATIC`. Has no effect if `setting == FIELD_VALUE`.

- `on_demand_skew_seconds`: (Advanced) this sets the number of seconds before data encoded on demand are considering stale and thus must be demanded again with the signal `QueueManager::signal_data_on_demand`. Setting this to 0 is unadvisable as it will cause many calls to `QueueManager::signal_data_on_demand` and thus waste CPU cycles needlessly encoding.

- `minimum_ack_wait_seconds`: (Advanced) how long to wait for an acknowledgment before resending the same data.

For example, to queue the message given in Section 2.2, the following snippet could suffice:

```
message_entry {
  protobuf_name: "example.MinimalStatus"
  ack: false
  blackout_time: 30
  max_queue: 1
  newest_first: true
  ttl: 900
```

```
    value_base: 0.5
    role { type: DESTINATION_ID  field: "dest"   }
    role { type: SOURCE_ID       field: "source" }
    role { type: TIMESTAMP       field: "time"   }
  }
```

2.4 Time Division Multiple Access (TDMA) Medium Access Control (MAC): AMAC

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

AMAC 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 a fixed (preprogrammed) cycle (`type: MAC_FIXED_DECENTRALIZED`) that can be updated by the application. The autodiscovery mode (`type: MAC_AUTO_DECENTRALIZED`) supported in version 1 is no longer provided in version 2. To disable the AMAC, use (`type: MAC_NONE`). See http://gobysoft.com/doc/2.0/acomms_mac.html for more details.

2.4.1 Configuration: MACConfig

The `goby::acomms::MACManager` is basically a `std::list<ModemTransmission>`. Thus, its configuration is primarily such an initial list of these `slot`s. Since `ModemTransmission` is extensible to handle different modem drivers, the AMAC configuration is also automatically extended. Some fields in `ModemTransmission` do not make sense to configure `goby::acomms::MACManager` with, so these are omitted here:

```
modem_id: 1
type: MAC_NONE
slot {
  src: -1
  dest: -1
  rate: 0
  type: UNKNOWN
  ack_requested: true
```

```
    slot_seconds: 10
    unique_id: 0
    [micromodem.protobuf.type]: BASE_TYPE
    [micromodem.protobuf.narrowband_lbl] {
      transmit_freq:
      transmit_ping_ms:
      receive_freq:
      receive_ping_ms:
      turnaround_ms:
      transmit_flag: true
      lbl_max_range: 2000
    }
    [micromodem.protobuf.remus_lbl] {
      enable_beacons: 15
      turnaround_ms: 50
      lbl_max_range: 1000
    }
    [goby.moos.protobuf.type]: BASE_TYPE
    [PBDriverTransmission.type]: BASE_TYPE
  }
  start_cycle_in_middle: true
```

Further details on these configuration fields:

- `type`: type of Medium Access Control. See http://gobysoft.com/doc/2.0/acomms_mac.html#amac_schemes for an explanation of the various MAC schemes.
- `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. Setting both src and dest to 0 causes AMAC to ignore this slot (which can be used to provide a blank slot).
  - `dest`: The receiving `modem_id` for this slot. Omit or set to -1 to allow next datagram to set the destination.
  - `rate`: Bit-rate code for this slot (0-5). 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).
  - `type`: Type of transaction to occur in this slot. If `DRIVER_SPECIFIC`, the specific hardware driver governs the type of this slot (e.g. `[micromodem.protobuf.type]: MICROMODEM_MINI_DATA`.

  - `slot_seconds`: The duration of this slot, in seconds.
  - `unique_id`: Integer field that can optionally be used to identify certain types of slots. For example, this allows integration of an in-band (but otherwise unrelated) sonar with the modem MAC cycle.

Relevant extensions of `goby::acomms::protobuf::ModemTransmission` for the WHOI Micro-Modem driver (`DRIVER_WHOI_MICROMODEM`):

- `slot`
  - `[micromodem.protobuf.type]`: Type of transaction to occur in this slot. This value is only used if `type == DRIVER_SPECIFIC`. Valid values include: `BASE_TYPE` (use the type given in `type` above), `MICROMODEM_TWO_WAY_PING` (`$CCMPC`, `MICROMODEM_REMUS_LBL_RANGING` (`$CCPDT`), `MICROMODEM_NARROWBAND_LBL_RANGING` (`$CCPNT`), `MICROMODEM_MINI_DATA` (`$CCMUC`).
  - `[micromodem.protobuf.narrowband_lbl]`: Narrowband long-baseline configuration. These are merged with any global settings given in the ModemDriver configuration (Section 2.5), with the values set here taking precedence.
  - `[micromodem.protobuf.remus_lbl]`: REMUS long-baseline configuration. These are merged with any global settings given in the ModemDriver configuration (Section 2.5), with the values set here taking precedence.

Several examples:

- Continous uplink from node 2 to node 1 with a 15 second pause between datagrams (this is node 1's configuration; it is the same for node 2 except for `modem_id = 2`):

```
modem_id: 1
type: MAC_FIXED_DECENTRALIZED
slot { src: 2  dest: 1  type: DATA  slot_seconds: 15 }
```

- Equal sharing for three vehicles (destination governed by next data packet):

```
modem_id: 1 # 2 or 3 for other vehicles
type: MAC_FIXED_DECENTRALIZED
slot { src: 1  type: DATA  slot_seconds: 15 }
slot { src: 2  type: DATA  slot_seconds: 15 }
slot { src: 3  type: DATA  slot_seconds: 15 }
```

- Three vehicles with both data and WHOI Micro-Modem two-way ranging (ping):

```
modem_id: 1 # 2 or 3 for other vehicles
type: MAC_FIXED_DECENTRALIZED
slot { src: 1  type: DATA  slot_seconds: 15 }
slot {
  src: 1
  dest: 2
  type: DRIVER_SPECIFIC
  [micromodem.protobuf.type]: MICROMODEM_TWO_WAY_PING
  slot_seconds: 5
}
slot {
  src: 1
  dest: 3
  type: DRIVER_SPECIFIC
  [micromodem.protobuf.type]: MICROMODEM_TWO_WAY_PING
  slot_seconds: 5
}
slot { src: 2  type: DATA  slot_seconds: 15 }
slot { src: 3  type: DATA  slot_seconds: 15 }
```

- One vehicle interleaving data and REMUS long-base-line (LBL) navigation pings:

```
modem_id: 1
type: MAC_FIXED_DECENTRALIZED
slot { src: 1  type: DATA  slot_seconds: 15 }
slot {
  src: 1
  dest: 2
  type: DRIVER_SPECIFIC
  [micromodem.protobuf.type]: MICROMODEM_REMUS_LBL_RANGING
```

```
    [micromodem.protobuf.remus_lbl] {
      enable_beacons: 0xf   # enable all four: b1111
      turnaround_ms: 50
      lbl_max_range: 500 # meters
    }
    slot_seconds: 5
  }
```

2.5 Abstract Acoustic (or other slow link) Modem Driver: ModemDriver

The ModemDriver unit provides a common interface to any modem capable of sending datagrams. It currently supports the WHOI Micro-Modem acoustic modem, UDP over the Internet, and is extensible to other acoustic (or slow link) modems. More details on the ModemDriver are available here: http://gobysoft.com/doc/2.0/acomms_driver.html.

2.5.1 Configuration: DriverConfig

Base driver configuration:

```
modem_id: 1
connection_type: CONNECTION_SERIAL
line_delimiter: "\r\n"
serial_port: "/dev/ttyS0"
serial_baud: 19200
tcp_server: "192.168.1.111"
tcp_port: 50010
```

- `modem_id`: A unique integer value for this particular vehicle (like a MAC address). Should be as small as possible for optimal bounding of the source and destination fields of the message. 0 is reserved for broadcast (analogous to 255.255.255.255 for IPv4).
- `connection_type`: How the modem is attached to this computer. Some of the drivers do not use this connection. Valid options: `CONNECTION_SERIAL` (uses a serial connection, e.g. `/dev/ttyS0`), `CONNECTION_TCP_AS_CLIENT` (connect using TCP where this application is a client, and the modem is a server),

CONNECTION_TCP_AS_SERVER (connect using TCP where this application is a server, and the modem is a client).

- line_delimiter: A string representing the "end-of-line" of each message from the modem.
- serial_port: Only for CONNECTION_SERIAL, the name of the serial port on this machine.
- serial_baud: Only for CONNECTION_SERIAL, the baud rate to use when talking to the modem.
- tcp_server: Only for CONNECTION_TCP_AS_CLIENT, the IP address or domain name of the modem TCP server.
- tcp_port: For CONNECTION_TCP_AS_CLIENT, the port to connect to on tcp_server; for CONNECTION_TCP_AS_SERVER, the port to bind on.

Extensions for the WHOI Micro-Modem (DRIVER_WHOI_MICROMODEM):

```
[micromodem.protobuf.Config.reset_nvram]: false
[micromodem.protobuf.Config.nvram_cfg]: ""
[micromodem.protobuf.Config.hydroid_gateway_id]: 0
[micromodem.protobuf.Config.narrowband_lbl] {
  transmit_freq:
  transmit_ping_ms:
  receive_freq:
  receive_ping_ms:
  turnaround_ms:
  transmit_flag: true
  lbl_max_range: 2000
}
[micromodem.protobuf.Config.remus_lbl] {
  enable_beacons: 15
  turnaround_ms: 50
  lbl_max_range: 1000
}
[micromodem.protobuf.Config.mm_version]: 1
```

- [micromodem.protobuf.Config.reset_nvram]: If true, reset all the modem's configuration settings at startup (before applying those specified in nvram_cfg). In general, it is a good idea to set this to true so that the modem's NVRAM (configuration) state is known.

- `[micromodem.protobuf.Config.nvram_cfg]`: This repeated field specifies an NVRAM configuration sentence to send. For example, to set `$CCCFG,DTO,10`, use "DTO,10" as the value for this field.
- `[micromodem.protobuf.Config.nvram_cfg]`: You must omit this in all cases except when using a Hydroid Buoy which uses a modified talker to communicate with the Micro-Modem. In that case, set this to the Buoy identification number.
- `[micromodem.protobuf.Config.narrowband_lbl]`: Default configuration used for each `MICROMODEM_NARROWBAND_LBL_RANGING` transmission. Overwritten by any settings also specified in the AMAC configuration. See http://gobysoft.com/doc/2.0/acomms_driver.html for the details of these fields.
- `[micromodem.protobuf.Config.remus_lbl]`: Default configuration used for each `MICROMODEM_REMUS_LBL_RANGING` transmission. Overwritten by any settings also specified in the AMAC configuration. See http://gobysoft.com/doc/2.0/acomms_driver.html for the details of these fields.
- `[micromodem.protobuf.Config.mm_version]`: Micro-Modem major version. Only Micro-Modem 1 is currently supported (and Micro-Modem 2 in backwards-compatible mode). Thus, currently, this field should always be 1.

Extensions for the example driver (`DRIVER_ABC_EXAMPLE_MODEM`):

```
[ABCDriverConfig.enable_foo]: true
[ABCDriverConfig.enable_bar]: false
```

This "modem" is simply an example on how to write drivers. See http://gobysoft.com/doc/2.0/acomms_driver.html#acomms_writedriver. Do not use this for real work.

Extensions for the MOOS uField driver (`DRIVER_UFIELD_SIM_DRIVER`) that uses the MOOS-IvP uField toolbox [3] as the transport:

```
```

```
[goby.moos.protobuf.Config.moos_server]: "localhost"
[goby.moos.protobuf.Config.moos_port]: 9000
[goby.moos.protobuf.Config.incoming_moos_var]: "ACOMMS_UFIELD_DRIVER_IN"
[goby.moos.protobuf.Config.outgoing_moos_var]: "ACOMMS_UFIELD_DRIVER_OUT"
[goby.moos.protobuf.Config.ufield_outgoing_moos_var]: "NODE_MESSAGE_LOCAL"
[goby.moos.protobuf.Config.rate_to_bytes]:
[goby.moos.protobuf.Config.modem_id_lookup_path]: ""
```

- `[goby.moos.protobuf.Config.moos_server]`: Address for the `MOOSDB`.
- `[goby.moos.protobuf.Config.moos_port]`: Port for the `MOOSDB`.
- `[goby.moos.protobuf.Config.incoming_moos_var]`: MOOS variable to use for incoming messages.
- `[goby.moos.protobuf.Config.outgoing_moos_var]`: MOOS variable to use for outgoing messages.
- `[goby.moos.protobuf.Config.ufield_outgoing_moos_var]`: The MOOS variable uField uses for relaying messages.
- `[goby.moos.protobuf.Config.rate_to_bytes]`: This repeated field is the size in bytes of the given rate. The order these are defined in the configuration file maps onto the rate. The first is rate 0, the second is rate 1, and so on. To emulate the WHOI Micro-Modem, use:

```
[goby.moos.protobuf.Config.rate_to_bytes]: 32
[goby.moos.protobuf.Config.rate_to_bytes]: 192
[goby.moos.protobuf.Config.rate_to_bytes]: 192
[goby.moos.protobuf.Config.rate_to_bytes]: 512
[goby.moos.protobuf.Config.rate_to_bytes]: 512
[goby.moos.protobuf.Config.rate_to_bytes]: 2048
```

- `[goby.moos.protobuf.Config.modem_id_lookup_path]`: Path to a file containing the mapping of MOOS Community names to modem IDs. 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
```

Extensions for the UDP driver (DRIVER_UDP), a basic driver for Goby that sends packets using UDP over IP:

```
[UDPDriverConfig.local] {
  ip: "127.0.0.1"
  port:
}
[UDPDriverConfig.remote] {
  ip: "127.0.0.1"
  port:
}
[UDPDriverConfig.max_frame_size]: 65536
```

- `[UDPDriverConfig.local]`: Source port of the local machine (`ip` is not used). This can be omitted, and then a dynamic port is used.
- `[UDPDriverConfig.remote]`: Address and port to send messages to.
- `[UDPDriverConfig.max_frame_size]`: Maximum UDP frame to send (in bytes).

Extensions for the ZeroMQ/Protobuf storage driver (DRIVER_PB_STORE_SERVER):

```
[PBDriverConfig.request_socket] {
  socket_type:
  socket_id: 0
  transport: EPGM
  connect_or_bind: CONNECT
  ethernet_address: "127.0.0.1"
  multicast_address: "239.255.7.15"
  ethernet_port: 11142
  socket_name: ""
}
[PBDriverConfig.query_interval_seconds]: 1
[PBDriverConfig.max_frame_size]: 65536
[PBDriverConfig.reset_interval_seconds]: 120
[PBDriverConfig.rate_to_bytes]:
```

This driver is still under development and thus is not for general use at the moment.

3 Goby Common Modules

3.1 Goby Common Applications

The Goby Common applications use a validating configuration reader based on the Google Protocol Buffers TextFormat class. The configuration of any given application is available by passing the `--example_config` flag (or `-e` for short) to that application. Additionally, any of the configuration that may be given in a file is also available as command line options. Provide `--help` (or `-h`) to see the command line options.

They all share a common subset of the configuration (`base`):

```
base {
  app_name: "myapp_g"
  loop_freq: 10
  platform_name: "unnamed_goby_platform"
  pubsub_config {
    publish_socket {
      socket_type:
      socket_id: 0
      transport: EPGM
      connect_or_bind: CONNECT
      ethernet_address: "127.0.0.1"
      multicast_address: "239.255.7.15"
      ethernet_port: 11142
      socket_name: ""
    }
    subscribe_socket {
      socket_type:
      socket_id: 0
      transport: EPGM
      connect_or_bind: CONNECT
      ethernet_address: "127.0.0.1"
      multicast_address: "239.255.7.15"
      ethernet_port: 11142
      socket_name: ""
    }
  }
  additional_socket_config {
    socket {
      socket_type:
      socket_id: 0
```

```
31        transport: EPGM
32        connect_or_bind: CONNECT
33        ethernet_address: "127.0.0.1"
34        multicast_address: "239.255.7.15"
35        ethernet_port: 11142
36        socket_name: ""
37      }
38    }
39    glog_config {
40      tty_verbosity: QUIET
41      show_gui: false
42      file_log {
43        file_name: ""
44        verbosity: VERBOSE
45      }
46    }
47  }
```

- `app_name`: Name of the application (defaults to binary name, i.e. oart of argv[0] after last `/`).
- `loop_freq`: How often to run the synchronous `loop` method.
- `platform_name`: Name of the node or platform this is running on .
- `pubsub_config`: Socket configuration for the publish-subscribe part of Goby-Common. If omitted, no connections or bindings will be made (if an application is standalone).
  - `publish_socket`: The socket used for publishing messages.
    * `socket_type`: Must always be `PUBLISH` (you can safely omit the field here).
    * `socket_id`: Generally a unique id, unless you want several sockets of the same type to send and receive together. You can safely omit this field; it defaults to 103999.
    * `transport`: `IPC` (UNIX sockets), `TCP`, `PGM` (Pragmatic General Multicast), `EPGM` (PGM encasulated in UDP). In generally, you will use `TCP` or `IPC`.
    * `connect_or_bind`: `CONNECT` is used on the client side, `BIND` is used on the server side. Generally, you will `BIND` the side on a well-known location, and `CONNECT` the sides that may be more dynamic.

        * `ethernet_address`: For `TCP`, `PGM` and `EPGM`, the ethernet address to use.
        * `multicast_address`: For `PGM` and `EPGM`, the multicast address of the group to join.
        * `ethernet_port`: The network port to connect or bind to.
        * `socket_name`: For `IPC`, the name (path) of the UNIX socket to create or connect to.
    - `subscribe_socket`: The socket used for received subscribed messages. Except where noted, the fields are the same as for `publish_socket`.
        * `socket_type`: Must always be `SUBSCRIBE` (you can safely omit the field here).
        * `socket_id`: Generally a unique id, unless you want several sockets of the same type to send and receive together. You can safely omit this field; it defaults to 103998.

- `additional_socket_config`: (Advanced) Used to add additional ZeroMQ connections or bindings.

- `glog_config`: Configure the `goby::glog` logging utility.

    - `tty_verbosity`: Verbosity of the debug logging to standard output in the controlling terminal. Choose `DEBUG1`-`DEBUG3` for various levels of debugging output, `VERBOSE` for some text terminal output, `WARN` for warnings only, and `QUIET` for no terminal output.
    - `file_log`: A repeated field to log the debugging output to one or more files. If omitted, no files are logged.
        * `file_name`: Path to file to log. The symbol `%1%` (if present) will be replaced by the current UTC date and time at application launch.
        * `verbosity`: Verbosity of this file log. Same enumeration options as `tty_verbosity`.

3.2 Liaison

Goby Liaison (`goby_liaison`) is an extensible web-browser based GUI for managing various aspects of Goby. It is written using the Wt [5] library and allows users to manage their Goby systems from any machine (GNU/Linux, Windows, Mac OS X) running a modern web browser (e.g. Firefox, Chrome).

The majority of Liaison is provided by plugin shared libraries that are loaded at runtime using the environmental variable `GOBY_LIAISON_PLUGINS`, which is a colon separated list of libraries (either absolute paths or in paths known to `ld`, such as `/usr/lib`).

The core of Goby Liaison is a server that allows connections from one or more clients through any major modern web browser. The core configuration options are given by:

```
base {
  ...
}
http_address: "localhost"
http_port: 54321
docroot: "/usr/share/goby/liaison"
additional_wt_http_params: "--accesslog=/tmp/access.log"
update_freq: 5
load_shared_library: ""
load_proto_file: ""
load_proto_dir: ""
start_paused: false
```

- `base`: Shared configuration for all `goby_common` applications. See section 3.1.
- `http_address`: IP address or domain name for the interface to bind on. Use `0.0.0.0` to bind on all interfaces. Use `localhost` to allow connections only from the local machine for security.
- `http_port`: TCP port to bind on.
- `docroot`: Path to the Wt `docroot`, where various resources are found (e.g. CSS, images, etc.). The default is usually correct for your installation.
- `additional_wt_http_params`: Additional command line parameters (separated by spaces) to pass to the Wt server. See http://www.webtoolkit.eu/wt/doc/reference/html/overview.html#config_wthttpd.
- `update_freq`: How often to update elements that require data from the server side without client input.
- `load_shared_library`: Load a shared library (probably containing Google Protobuf messages) for use.

- `load_proto_file`: Load a `.proto` file directly and compile it at runtime for use. When possible, use `load_shared_library`.
- `load_proto_dir`: Path to a directory containing `.proto` files. All the `.proto` files in this directory will be loaded and compiled for use.
- `start_paused`: For modules that require server side updates without client input, setting this true will start up Liaison with these modules paused. This prevents any server side initiated data from being pushed to the client. Set true for use on low-throughput links (e.g. wireless at sea).

Additional configuration may be available from the loaded plugins. For example, see the MOOS plugins in section 4.5.

To connect to a server using the default configuration, simply type `http://localhost:54321` into the address bar of your favorite web browser.

3.3 Gateway Applications

Goby, which uses ZeroMQ as a transport layer, sometimes also needs to talk to other systems using incompatible transport mechanisms. To do this, "gateway" applications can be developed that pass packets between the ZeroMQ (Goby) "world" and the other system's world. Thus far, one gateway has been written, the `moos_gateway_g` (see section 4.5.1) for interfacing with the MOOS middleware.

4 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 [4] and is available at http://moos-ivp.org. The usage of these modules is documented here. See http://gobysoft.org/wiki/InstallingGoby for how to install Goby.

4.1 Goby MOOS Applications

The Goby MOOS applications share a common subclass of CMOOSApp that provides a validating configuration reader based on the Google Protocol Buffers TextFormat class. The configuration is still embedded within the .moos file, but the syntax is somewhat different. Here you can control logging to a text file and 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 Goby MOOS configuration variable `common::lat_origin`. This allows Goby MOOS applications to conform to MOOS *de facto* conventions.

Any Goby MOOS application will give all its valid configuration parameters with

```
> pGobyApp --example_config
```

```
ProcessConfig = pGobyApp
{
  common {
    log: true
    log_path: "./"
    log_verbosity: DEBUG2
    community: "AUV23"
    lat_origin: 42.5
    lon_origin: 10.9
    time_warp_multiplier: 1
    app_tick: 10
    comm_tick: 10
    verbosity: VERBOSE
    show_gui: false
```

```
15      initializer {
16        type: INI_DOUBLE
17        moos_var: "SOME_MOOS_VAR"
18        global_cfg_var: "LatOrigin"
19        dval: 3.454
20        sval: "a string"
21        trim: true
22      }
23    }
24  }
```

Some details about the configuration values:

- `log`: boolean to indicate whether to log terminal output or not to files in the path by `log_path`.
- `log_path`: folder to log all terminal output to for later debugging. Similar to system logs in /var/log.
- `log_verbosity`: verbosity of the log file. See `verbosity` for the various settings.
- `community`: the name of the current vehicle community. If omitted, read from the `Community=` global MOOS configuration field.
- `lat_origin`: a decimal degrees latitude indicating the local cartesian origin. If omitted, read from the `LatOrigin=` global MOOS configuration field.
- `lon_origin`: a decimal degrees longitude indicating the local cartesian origin. If omitted, read from the `LongOrigin=` global MOOS configuration field.
- `app_tick`: same as AppTick.
- `comm_tick`: same as CommsTick.
- `verbosity`: choose `DEBUG1-DEBUG3` for various levels of debugging output, `VERBOSE` for some text terminal output, `WARN` for warnings only, and `QUIET` for no terminal output.
- `show_gui`: if true, the running terminal opens an NCurses GUI helpful to debugging and visualizing the many data flows of pAcommsHandler. The verbosity in this GUI is governed by `verbosity`.

- `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`.

4.2 pTranslator

`pTranslator` is a translator between MOOS types (strings and doubles) and Google Protocol Buffers messages (which includes DCCL messages). All of the functionality of `pTranslator` is also present in `pAcommsHandler`, but `pTranslator` is provided as a standalone application for cases when Goby-Acomms is not needed, but the translation functionality is. Also, pTranslator loops back all created messages and immediately publishes them, whereas pAcommsHandler publishes messages received acoustically, and creates messages to be transmitted.

The configuration for `pTranslator` is as follows:

```
ProcessConfig = pTranslator
{
  common {
  ...
  }
  load_shared_library: ""
  load_proto_file: ""
  translator_entry {
    protobuf_name: ""
    trigger {
      type: TRIGGER_PUBLISH
      moos_var: ""
      period:
      mandatory_content: ""
    }
```

```
    create {
      technique: TECHNIQUE_PROTOBUF_TEXT_FORMAT
      moos_var: ""
      format: ""
      repeated_delimiter: ","
      algorithm {
        name: ""
        primary_field:
      }
    }
    publish {
      technique: TECHNIQUE_PROTOBUF_TEXT_FORMAT
      moos_var: ""
      format: ""
      repeated_delimiter: ","
      algorithm {
        name: ""
        output_virtual_field:
        primary_field:
        reference_field:
      }
    }
    use_short_enum: false
  }
  modem_id_lookup_path: ""
  multiplex_create_moos_var: ""
}
```

- `common`: Parameters that can be set for any of the Goby MOOS applications. See section 4.1.
- `load_shared_library`: Repeated string, each with a path to a shared library containing compiled DCCL (Google Protocol Buffers) messages.
- `load_proto_file`: Repeated string, each with one path to a .proto file containing compiled DCCL (Google Protocol Buffers) messages. These will be compiled at runtime and loaded. It is preferable to use `load_shared_library` when possible, as syntactical and type mistakes in the DCCL messages will be caught at compile-time rather than delayed to runtime.
- `translator_entry`: Repeated entry: there should be one `translator_entry` defined for each Google Protobuf message type that you wish to translate to or from.

- `protobuf_name`: Fully qualified name (packages separated by `.`, e.g. `example.MinimalStatus`) to the Protobuf message that this translator should use. This message must be loaded either by `load_shared_library` or `load_proto_file`.
- `trigger`: The event that causes this translation to occur.
  * `type`: Either `TRIGGER_PUBLISH` (do a translation every time a given MOOS variable is published to) or `TRIGGER_TIME` (do a translation on a regular frequency).
  * `moos_var`: For `TRIGGER_PUBLISH`, the MOOS variable that causes the translation to occur.
  * `period`: For `TRIGGER_TIME`, the period (in seconds) between translations.
  * `mandatory_content`: For `TRIGGER_PUBLISH`, if this is defined, the `moos_var` must contain this substring in order to trigger this translation. Use of this field allows a single MOOS variable to trigger several different translations.
- `create`: Upon triggering, this defines how the Protobuf message is created from one or more MOOS variables. Repeat this field for multiple MOOS variables. The `create` directives are processed in the order they are defined and thus later `create`s that write the same fields will overwrite earlier ones.
  * `technique`: The parsing technique to use. See section 4.3.
  * `moos_var`: The MOOS variable to use for this `create`.
  * `format`: For `TECHNIQUE_FORMAT`, the format string to use. This is similar to scanf, but instead of type specifiers, numerical specifiers are used, surrounded by `%` on both sides. For example, if the `format` value is `foo=%1%`, this `create` will parse a `moos_var` containing `foo=5` and put the value `5` into field `1` of the Protobuf message given by `protobuf_name`.
  * `repeated_delimiter`: When parsing for `repeated` Protobuf fields, this is the string that delimits fields. For example, if `foo=%1%`, field 1 is `repeated int32 field_name = 1`, and the value to parse is `foo=10;12;13;14`, `repeated_delimiter` should be ";" in order to parse these four numbers into a "vector" of values in that field.
  * `algorithm`: An algorithm to modify the parsed field before placing it in the Protobuf message. These are largely provided for backwards compatibility for Goby v1, and are not necessarily encouraged for new use. See

http://gobysoft.com/dl/goby1-user-manual.pdf for a detailing of the available algorithms. Several algorithms can be chained (processed in the order they are defined) by repeated this `algorithm` field with the same `primary_field`.

- `name`: Name of the algorithm, e.g. `to_upper`.
- `primary_field`: The field number to apply this algorithm to.

– `publish`: Upon receipt of a Protobuf message, how to publish it back to one or more MOOS variable(s). Several `publish` entries should be specified to publish to several MOOS variables.

* `technique`: The serialization technique to use. See section 4.3.
* `moos_var`: The MOOS variable to write to for this `publish`.
* `format`: For `TECHNIQUE_FORMAT`, the format string to use. This is similar to printf, but instead of type specifiers, numerical specifiers are used, surrounded by `%` on both sides. For example, if the `format` value is `foo=%1%`, this `publish` will write a `moos_var` containing `foo=5` if field 1 in the Protobuf message was `5`.
* `repeated_delimiter`: When writing `repeated` Protobuf fields, this is the string that is used to delimit fields.
* `algorithm`: Several algorithms can be chained (processed in the order they are defined) by repeated this `algorithm` field with the same `primary_field`.
  - `name`: Name of the algorithm, e.g. `to_upper`.
  - `primary_field`: The field number to apply this algorithm to.
  - `output_virtual_field`: A "virtual" field number (one that doesn't exist in the actual Protobuf message) that is used to specify the output of this algorithm. This virtual field can then be used in the `format` string like a real field.
  - `reference_field`: The field(s) required by the algorithm as references, if the algorithm requires them (e.g. `utm_x2lon`).

– `use_short_enum`: If true, the front of the enumeration value is removed if it matches the field name plus a `_`. For example, if the enum field is `foo`, and the enumerations are `FOO_OPTION1`, `FOO_OPTION2`, then `OPTION1` and `OPTION2` are published. If false (the default), the enumeration values are published as defined. This is mostly here for backwards compatibility with Goby 1.

4.3 Translator techniques

There are three broad categories of translator techniques: 1) those that use the Google Protocol Buffers tools (`TECHNIQUE_PREFIXED_PROTOBUF_TEXT_FORMAT`, `TECHNIQUE_PROTOBUF_TEXT_FORMAT`, `TECHNIQUE_PROTOBUF_NATIVE_ENCODED`), 2) one that uses the *de facto* MOOS convention of `key=value` pairs delimited by commas (`TECHNIQUE_COMMA_SEPARATED_KEY_EQUALS_VALUE_PAIRS`), and 3) one that is based roughly on printf/scanf (`TECHNIQUE_FORMAT`).

More details on each translator type:

- `TECHNIQUE_PROTOBUF_TEXT_FORMAT`: exactly the same as if you used the Google TextFormat class: https://developers.google.com/protocol-buffers/docs/reference/cpp/google.protobuf.text_format.
- `TECHNIQUE_PREFIXED_PROTOBUF_TEXT_FORMAT` (recommendated for most uses). Same as `TECHNIQUE_PROTOBUF_TEXT_FORMAT` but prefixed with `@PB[TypeName]` , so that you can put multiple Protobuf Types in a single MOOS Variable (if you really need to). It's also quite human readable and allows for programs to read / write generic Protobuf messages. This technique is useful enough, there are two shortcut functions for use in your C++ MOOS code (`#include "goby/moos/moos_protobuf_helpers.h"`): `serialize_for_moos` and `parse_for_moos`.
- `TECHNIQUE_PROTOBUF_NATIVE_ENCODED`: exactly the same as if you used the default binary Google encoding (binary), represented as a byte string. This tends to break the MOOS tools that assume strings are ASCII / UTF-8.
- `TECHNIQUE_COMMA_SEPARATED_KEY_EQUALS_VALUE_PAIRS`: all fields represented as `key1=value1,key2=value2,...` Messages with submessages are flattened and the keys assembled by concatenation separated with `_`. This is similar to the existing `NODE_REPORT` variable used in MOOS-IvP.
- `TECHNIQUE_FORMAT`: sort of like `printf` / `scanf`, except instead of typed directives (e.g. `%d`), Goby uses numeric directives that correspond the protobuf message field id (e.g. `foobar=%2%`). Submessages can be referenced using ":" (e.g. `%5:1%`, where field 5 is a Message), repeated fields can be referenced using "." (e.g. `%7.1%`, where field 7 is repeated). Note the ending `%` on each directive, which is different than `printf`.

4.4 pAcommsHandler

pAcommsHandler provides a:

1. MOOS Application wrapper for the Goby-Acomms communication library.
2. set of translation tools for converting the DCCL messages (written as an extension of Google Protocol Buffers) to MOOS types (strings and doubles) and vice-versa.
3. full backwards-compatibility support module for version 1 XML messages.

This section describes only the parts relevant for interface to MOOS (variables and translator entries that allow you to read and write to and from DCCL (Protobuf) messages). You should read Chapter 2 before starting this section and reference it as necessary.

4.4.1 Parameters for the pAcommsHandler Configuration Block

Example moos file

pAcommsHandler has a large number of configuration options, many of which you will never use or leave as default. You can always get a complete listing of MOOS file parameters with their syntax by running

```
> pAcommsHandler --example_config
```

These configuration values are provided here (with . . . where the relevant configuration is provided elsewhere in this document):

```
ProcessConfig = pAcommsHandler
{
  common {
  ...
  }
  modem_id: 1
  driver_type: DRIVER_NONE
  driver_cfg {
  ...
  }
  mac_cfg {
```

```
    ...
    }
    queue_cfg {
    ...
    }
    dccl_cfg {
    ...
    }
    route_cfg {
    ...
    }
    moos_var {
      prefix: "ACOMMS_"
      driver_raw_in: "NMEA_IN"
      driver_raw_out: "NMEA_OUT"
      driver_raw_msg_in: "RAW_INCOMING"
      driver_raw_msg_out: "RAW_OUTGOING"
      driver_receive: "MODEM_RECEIVE"
      driver_transmit: "MODEM_TRANSMIT"
      queue_receive: "QUEUE_RECEIVE"
      queue_transmit: "QUEUE_TRANSMIT"
      queue_ack_transmission: "ACK"
      queue_ack_original_msg: "ACK_ORIGINAL"
      queue_expire: "EXPIRE"
      queue_size: "QSIZE"
      queue_flush: "FLUSH_QUEUE"
      mac_cycle_update: "MAC_CYCLE_UPDATE"
      mac_initiate_transmission: "MAC_INITIATE_TRANSMISSION"
    }
    load_shared_library: "/usr/lib/libmy_dccl_messages.so"
    load_proto_file: "/usr/include/mylib/message.proto"
    translator_entry {
    ...
    }
    multiplex_create_moos_var: "LIAISON_COMMANDER_OUT"
    modem_id_lookup_path: ""
    transitional_cfg {
      modem_id: 1
      message_file {
        path: "/home/toby/goby/src/acomms/examples/chat/chat.xml"
        manipulator: NO_MANIP
      }
      generated_proto_dir: "/tmp"
    }
  }
```

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. See section 4.1.
- `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 `$CCCFG,SRC,#`). For the remainder of the document, `modem_id` refers to the value `$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 modules (`driver_cfg`, `queue_cfg`, `mac_cfg`)
- `driver_type`:
  - `DRIVER_WHOI_MICROMODEM` is a driver for the WHOI Micro-Modem.
  - `DRIVER_ABC_EXAMPLE_MODEM` is a simple test "modem". Do not use this for real work, but rather for learning how to write new drivers for Goby.
  - `DRIVER_UFIELD_SIM_DRIVER` is a driver for the MOOS-IvP uField toolbox.
  - `DRIVER_PB_STORE_SERVER` is a ZeroMQ (TCP, UNIX sockets) driver for the `goby_store_server` database.
  - `DRIVER_UDP` is a user datagram protocol (UDP) driver. This is probably the easiest driver to start with for learning pAcommsHandler.
  - `DRIVER_NONE` disables the modem driver.
- `driver_cfg`: Configures the base driver and the specific driver selected. See section 2.5.
- `mac_cfg`: Configures the acoustic Medium Access Control. See section 2.4.
- `queue_cfg`: Configures the Priority Queuing layer. See section 2.3.
- `dccl_cfg`: Configures the Dynamic Compact Control Language. See section 2.2.
- `route_cfg`: Configures a basic static routing module. This is experimental and subject to change.

- `moos_var`: Rename any or all of the MOOS variables published by pAcommsHandler.
- `load_shared_library`: Repeated string, each with a path to a shared library containing compiled DCCL (Google Protocol Buffers) messages.
- `load_proto_file`: Repeated string, each with one path to a .proto file containing compiled DCCL (Google Protocol Buffers) messages. These will be compiled at runtime and loaded. It is preferable to use `load_shared_library` when possible, as syntactical and type mistakes in the DCCL messages will be caught at compile-time rather than delayed to runtime.
- `translator_entry`: List of entries indicating when to make (*trigger*) and how to *create* outgoing DCCL messages, and how to *publish* incoming DCCL messages. This can be thought of as providing a generic interface between MOOS strings and Google Protocol Buffers messages. See section 4.2 for a full explanation on how to configure this translation.
- `multiplex_create_moos_var`: Used by `goby_liaison` to publish multiple commands (outgoing messages) on a single MOOS variable.
- `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
```

- `transitional_cfg`: Provides the same functionality as `dccl_cfg` does in pAcommsHandler from version 1 of Goby. Behind the scenes, XML messages are read, translated to version 2 Protobuf DCCL messages, and written to the `generated_proto_dir`, and subsequently loaded using `load_proto_file`. The appropriate `translator_entry`s are also created from these messages. Do not use this configuration or the XML representation of DCCL messages for any new projects. See the version 1 documentation (http://gobysoft.org/doc/1.1/) for more details on the XML representation of DCCL messages.

4.5 MOOS Plugins for Goby Liaison

Goby2 provides two applications ("tabs") inside Liaison (see section 3.2) that can launched by setting the environmental variable `GOBY_LIAISON_PLUGINS` to include the library `libliaison_plugins_goby_moos.so`. For example, in `bash`:

```
export GOBY_LIAISON_PLUGINS=/usr/lib/libliaison_plugins_goby_moos.so
goby_liaison
```

Please note that multiple plugin libraries can be loaded by separating the library paths with colons. The MOOS Plugins are: 1) MOOSCommander, an application that allows operator entry of any Protobuf message, and 2) MOOSScope, a tool that allows examining some subset of the current MOOSDB variables.

4.5.1 Connecting the Goby Liaison to the MOOSDB using moos_gateway_g

Liaison does not use the standard CMOOSCommClient TCP transport that MOOS uses, but rather ZeroMQ [6]. However, the Goby application `moos_gateway_g` was written to allow messages to pass between these two different "worlds." The `moos_gateway_g` makes a ZeroMQ publish/subscribe connection on one side, and a CMOOSCommClient connection to a `MOOSDB` on the other. Messages are passed between the two worlds using a set of configured filters.

The configuration available is (`moos_gateway_g --example_config`):

```
base {
  ...
}
moos_server_host: "localhost"
moos_server_port: 9000
moos_comm_tick: 5
moos_subscribe_filter: ""
goby_subscribe_filter: ""
```

- `base`: Shared configuration for all `goby_common` applications. See section 3.1. This includes the publish-subscribe configuration required to connect to the Goby ZeroMQ side of the gateway.
- `moos_server_host`: IP address or domain name for the `MOOSDB`
- `moos_server_port`: Port to connect to the `MOOSDB`

- `moos_comm_tick`: Frequency to call into the `MOOSDB` to retrieve mail.
- `moos_subscribe_filter`: A repeated string containing a substring to subscribe for in the MOOSDB. That is, `"NAV_"` will subscribe to `NAV_X`, `NAV_Y`, `NAV_DEPTH`, etc. The empty string (`""`) subscribes to everything.
- `goby_subscribe_filter`: Same as the `moos_subscribe_filter` but for the Goby side.

4.5.2 MOOS Commander GUI (Liaison)

The MOOS Commander tab is shown (with annotations) in Fig. 4.1. This is tool that allows a human operator to send DCCL messages (typically commands) to one or more robots. It supports sending any DCCL message (and any regular Protobuf message) known either at compile time (`load_shared_library`) or runtime (`load_proto_file`).

When the MOOS plugins are loaded, the Commander tab can be configured with the following settings with the file passed to Liaison:

```
[goby.common.protobuf.pb_commander_config] {
  load_protobuf_name: ""
  value_width_pixels: 500
  modify_width_pixels: 100
  sqlite3_database: "/tmp/liaison_commander_autosave.db"
  database_pool_size: 10
  database_view_height: 400
  database_width {
    comment_width: 200
    name_width: 200
    ip_width: 150
    time_width: 150
  }
  modal_dimensions {
    width: 800
    height: 200
  }
  subscription: ""
  time_source_var: ""
}
```

- `load_protobuf_name`: Repeated field, each with the full name of a Protobuf message as given by the `google::protobuf::Descriptor::full_name()`. This is

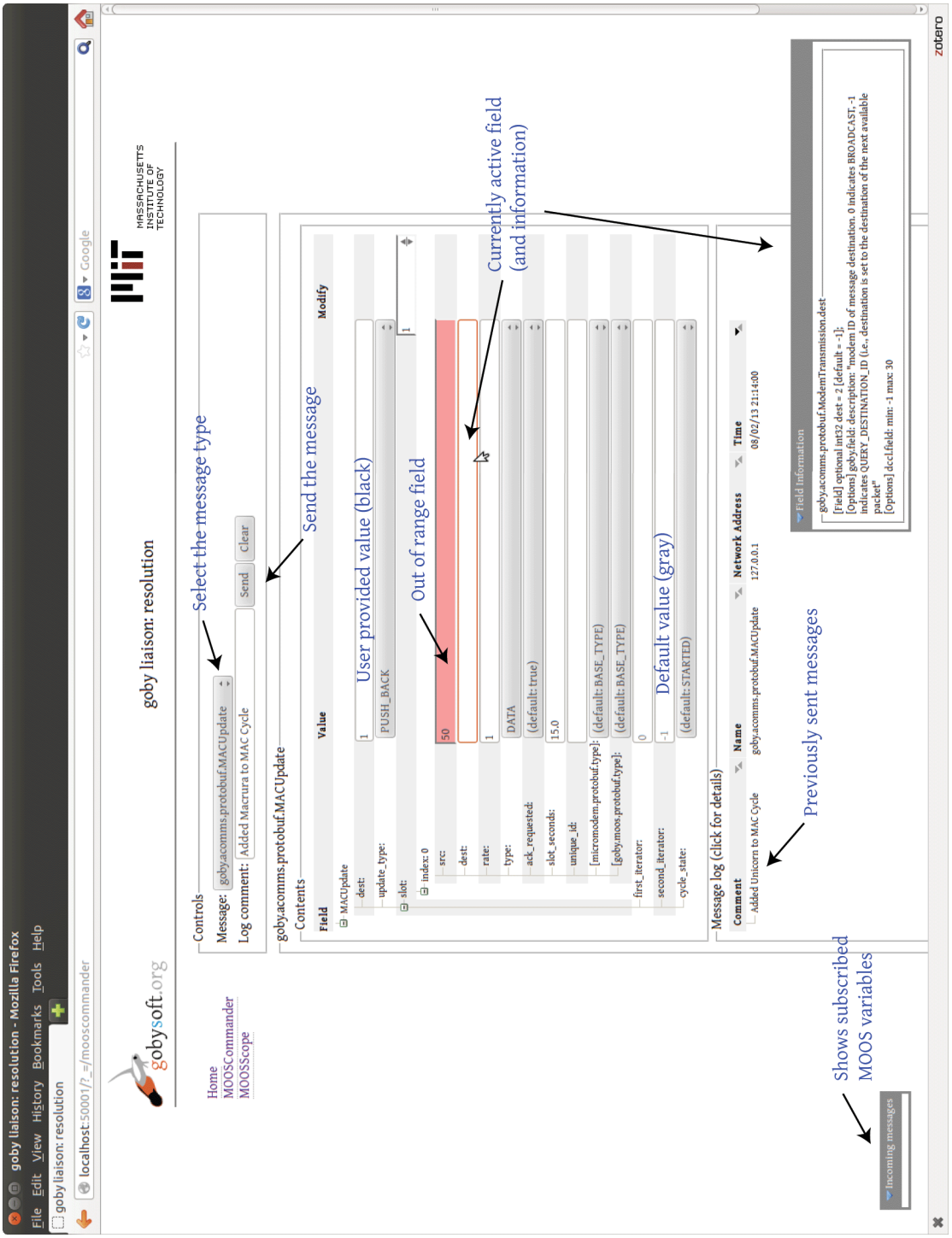

Figure 4.1: The MOOS Commander tab in Goby Liaison

the package name(s), followed by the message name, delimited by periods. For example, `example.MinimalStatus`. The messages will be loaded from `load_shared_library`, `load_proto_file`, and/or `load_proto_dir` configuration values given in the Liaison configuration, and made available for sending using the MOOS Commander GUI.

- `value_width_pixels`: Change the display width of the `value` field (in pixels).
- `modify_width_pixels`: Change the display width of the `modify` field (in pixels).
- `sqlite3_database`: Path to an SQLite3 database file to use (or create) to store all previously sent messages. Delete or modify this file to remove the history.
- `database_pool_size`: The connection pool size for database connections. Generally this should be larger than the number of expected simultaneously connection clients to Liaison.
- `database_view_height`: The height of the `Message log` section of the GUI (in pixels).
- `database_width`: The widths (in pixels) of various components of the `Message log` section.
- `modal_dimensions`: The dimensions (in pixels) of the modal (popup) dialog when sending a message.
- `subscription`: A repeated field, each contains a MOOS variable name to subscribe for. Subscribed variables will be shown in the lower left corner of the Commander GUI.
- `time_source_var`: The source of time e.g. `DB_TIME` from the MOOS database to be used for `codec="_time"` DCCL fields.

4.5.3 MOOS Scope GUI (Liaison)

```
[goby.common.protobuf.moos_scope_config] {
  subscription: ""
  column_width {
    key_width: 150
    type_width: 60
    value_width: 200
```

```
      time_width: 150
      community_width: 80
      source_width: 80
      source_aux_width: 120
    }
    sort_by_column: COLUMN_KEY
    sort_ascending: true
    scope_height: 400
    regex_filter_column: COLUMN_KEY
    regex_filter_expression: ".*"
    history {
      key: ""
      show_plot: false
      plot_width: 800
      plot_height: 300
    }
  }
```

- `subscription`: A repeated field, each with the string name of a MOOS variable. You can optionally use `*` at the end of the string for basic globbing. Use `regex_filter_expression` below for more advanced filtering. You can subscribe to `"*"` for all variables. Note that receiving mail for subscribed variables consumes network bandwidth, so it may be useful to subscribe to a small subset of variables before filtering when on a limited network connection.
- `column_width`: Width (in pixels) for the individual scope columns.
- `sort_by_column`: Which column to sort by on startup: Options are `COLUMN_KEY`, `COLUMN_TYPE`, `COLUMN_VALUE`, `COLUMN_TIME`, `COLUMN_COMMUNITY`, `COLUMN_SOURCE`, `COLUMN_SOURCE_AUX`, `COLUMN_MAX`.
- `sort_ascending`: If true, sort ascending; if false, sort descending.
- `scope_height`: Height (in pixels) of the scope display.
- `regex_filter_column`: Which column to apply the `regex_filter_expression` to.
- `regex_filter_expression`: A regular expression used to filter the scope results. Defaults to `".*"` which is an all-pass filter.
- `history`: Enable one or more history panels showing the log of a given MOOS variable.

- `key`: The MOOS variable name to show history for.
- `show_plot`: If true, shows a graph of the variable history. Only valid for MOOS double types.
- `plot_width`: Width of the plot in pixels.
- `plot_height`: Height of the plot in pixels.

4.5.4 Example working configuration

Here are the configuration files for `moos_gateway_g` and `goby_liaison` with the MOOS plugins enabled from a working system as an example:

```
# moos_gateway_g
base {
  platform_name: "resolution"
  pubsub_config {
    publish_socket {
      transport: IPC
      socket_type: PUBLISH
      connect_or_bind: BIND
      socket_name: "../.tmp/moos_gateway_g_pub_resolution"
    }
    subscribe_socket {
      transport: IPC
      socket_type: SUBSCRIBE
      connect_or_bind: BIND
      socket_name: "../.tmp/moos_gateway_g_sub_resolution"
    }
  }
  glog_config {
    tty_verbosity: QUIET
  }
}
moos_server_host: "localhost"
moos_server_port: 9001
moos_comm_tick: 5
moos_subscribe_filter: ""
goby_subscribe_filter: ""
```

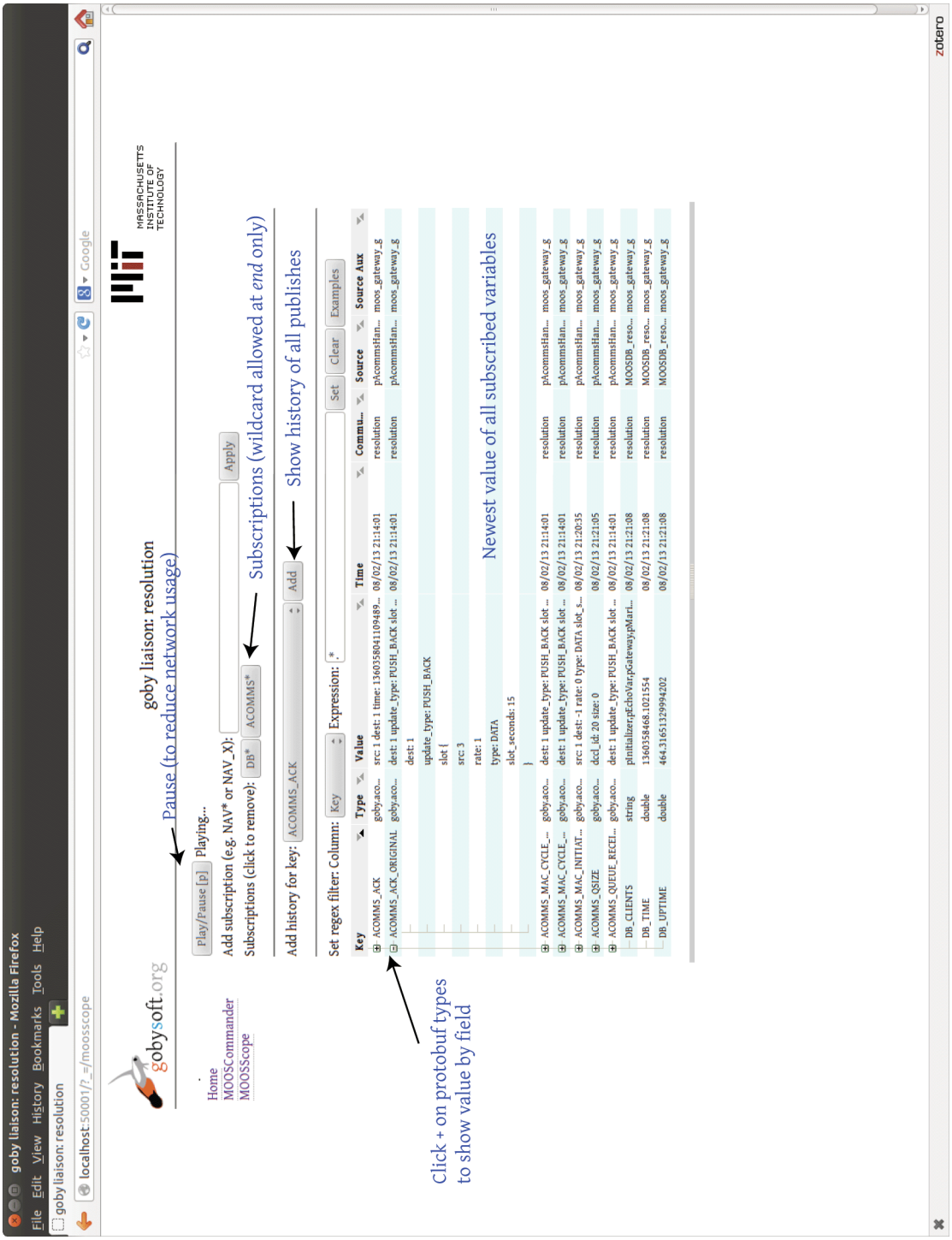

Figure 4.2: The MOOS Scope tab in Goby Liaison

```
# GOBY_LIAISON_PLUGINS=libliaison_plugins_goby_moos.so goby_liaison
base {
  platform_name: "resolution"
  pubsub_config {
    publish_socket {
      transport: IPC
      socket_type: PUBLISH
      connect_or_bind: CONNECT
      socket_name: "../.tmp/moos_gateway_g_sub_resolution"
    }
    subscribe_socket {
      transport: IPC
      socket_type: SUBSCRIBE
      connect_or_bind: CONNECT
      socket_name: "../.tmp/moos_gateway_g_pub_resolution"
    }
  }
  glog_config {
    tty_verbosity: QUIET
        file_log {
      file_name: "../logs/simulation/goby_liaison_%1%.txt"
      verbosity: DEBUG2
    }
  }
}
http_address: "localhost"
http_port: 50001
update_freq: 10
start_paused: false
[goby.common.protobuf.moos_scope_config] {
  subscription: "ACOMMS*"
  column_width {
    key_width: 150
    type_width: 60
    value_width: 200
    time_width: 150
    community_width: 80
    source_width: 80
    source_aux_width: 120
  }
  sort_by_column: COLUMN_KEY
  sort_ascending: true
  scope_height: 400
  regex_filter_column: COLUMN_KEY
  regex_filter_expression: ".*"
}
[goby.common.protobuf.pb_commander_config] {
  subscription: "ACOMMS_ACK_ORIGINAL"
```

```
49    subscription: "ACOMMS_NETWORK_ACK"
50    subscription: "ACOMMS_EXPIRE"
51    time_source_var: "DB_TIME"
52    load_protobuf_name: "LAMSS_DEPLOY"
53    load_protobuf_name: "LAMSS_TRANSIT"
54    load_protobuf_name: "LAMSS_PROSECUTE"
55    load_protobuf_name: "SIMULATE_TARGET"
56    load_protobuf_name: "SURFACE_DEPLOY"
57    load_protobuf_name: "ACOUSTIC_MOOS_POKE"
58    load_protobuf_name: "goby.acomms.protobuf.MACUpdate"
59    sqlite3_database: "../.tmp/liaison_commander_autosave.db"
60  }
61  load_shared_library: "../../lamss/lib/liblamss_protobuf.so"
```

4.6 Migrating from Version 1 to Version 2

pAcommsHandler from Goby version 2 (Goby2) provides nearly full backwards compatibility directly using the XML messages from Goby version 1 (Goby1).

In order to use XML messages from Goby1 in Goby2, simply rename the `dccl_cfg` section of the pAcommsHandler1 configuration block to `transitional_cfg`. In general this is all that needs to be done, as pAcommsHandler2 will automatically internally convert the XML files to .proto files and load them. Some special features of the XML files are not supported in version 2:

- Algorithms without a corresponding source variable.
- Algorithms with reference fields (e.g. subtract:timestamp) for message creation. Reference fields are still allowed upon publish.
- <`format`> tags must be specified in the boost::format `%1%`, `%2%`, etc. format, not using `%d` printf specifiers. The typed specifiers would work in Goby v1 but are not supported at all in Goby v2.

While the XML files can be used directly as a temporary measure, it is recommended to transition all your XML files to .proto files for direct use with pAcommsHandler2. To ease your transition, there is a tool `dccl_xml_to_dccl_proto` that will automatically convert your XML files for you. The usage is

```
dccl_xml_to_dccl_proto message_xml_file.xml [directory for generated .proto (default = pwd (.)]
```

In Goby1, the XML files contained both structure information and MOOS translation information. In Goby2, these are separated to allow better support of non-MOOS systems.

The tool will write the generated .proto files (the structure information) to the directory specified as the second command line parameter (defaults to the current/working directory). It will write to standard output the required additions to the pAcommsHandler2 configuration file for the queuing and translation information present in the XML file. Simply copy these parts to your MOOS file and you can continue to use your old messages natively.

4.7 iFrontSeat

4.7.1 Introduction

Motivation

Broadly, our goal in Goby is facilitate the development of a autonomy, sensing, and communications infrastructure that can operate on a heterogeneous collection of vehicles. One way to help effect this is to split the system into two components: the *frontseat* and *backseat* computing systems. The *frontseat* is provided by the vehicle manufacturer and is typically proprietary. It is responsible for low level control of the vehicle. The *backseat* runs the high level autonomy (typically the IvP Helm), sensing, and communications (typically Goby) components. The requirements of the *frontseat* on the *backseat* is minimally a continuous (e.g. 1 Hz) stream of course directives, such as desired heading, speed, and depth of the AUV. The requirements of the *backseat* on the *frontseat* is a best attempt to carry out these directives constrained by the dynamics of the vehicle, as well as a feed of the vehicles' navigation solution.

Not surprisingly, a piece of software is required to interface between the *frontseat* and the *backseat*. This code (`iFrontSeat`) is the subject of section.

Historically, a new interface has been written for each vehicle that was to be used with MOOS-IvP[1]. This led to a proliferation of approaches for handling the state transitions and control, primarily from `pHelmIvP`. In some cases, misunderstandings involving various aspects of MOOS-IvP led to vehicle runaways. Furthermore, as MOOS-IvP becomes even more widely adopted and the number of manufacturers of robotic assets increases, it seems sensible to minimize the duplication of effort involved in writing interfaces.

[1] For example, the applications iHuxley, iRecon, iOceanServerComms, . . .

Design overview

`iFrontSeat` (and its corresponding components in the library `libgoby_moos`) is comprised of two major components (the full UML structure diagram is given in Fig. 4.3):

- A base class `FrontSeatInterfaceBase` and MOOS Application `iFrontSeat` providing the IvP Helm state transition logic and MOOSDB subscriptions and publications. This is written *once* and used by all the specific drivers.
- A collection of derived classes (which are compiled into individual shared libraries) to implement the interface provided by `FrontSeatInterfaceBase` for a given manufacturer or vehicle type. The currently available drivers include:
  - `BluefinFrontSeat`: Implements `FrontSeatInterfaceBase` for the Bluefin Robotics family of AUVs using the Huxley software.

Running iFrontSeat

iFrontSeat always requires exactly one driver library to be loaded before any command-line parameters will be accepted. The driver libraries are runtime-loaded because this allows for a driver developer to create his or her own driver without changing any of the Goby source code. The driver library is loaded from the environmental variable `IFRONTSEAT_DRIVER_LIBRARY`. For example, use the bash shell, one can load iFrontSeat with the Bluefin driver (see section 4.7.3) with this invocation:

```
IFRONTSEAT_DRIVER_LIBRARY=libgoby_frontseat_bluefin.so.25 iFrontSeat
```

The library specified must be a complete path or on the `ld` library search path (e.g. set using `LD_LIBRARY_PATH`). Alternatively, you could export `IFRONTSEAT_DRIVER_LIBRARY` from one of the shell configuration files (e.g. `~/.bashrc`), and then simply run `iFrontSeat` on the command line.

4.7.2 Shared MOOS Side Components

iFrontSeat is a Goby MOOS application, which means it uses a validating configuration reader based on Google Protocol Buffers instead of the standard MOOS ProcessConfigReader. The syntax is similar, and you can always get all the valid configuration parameters by running

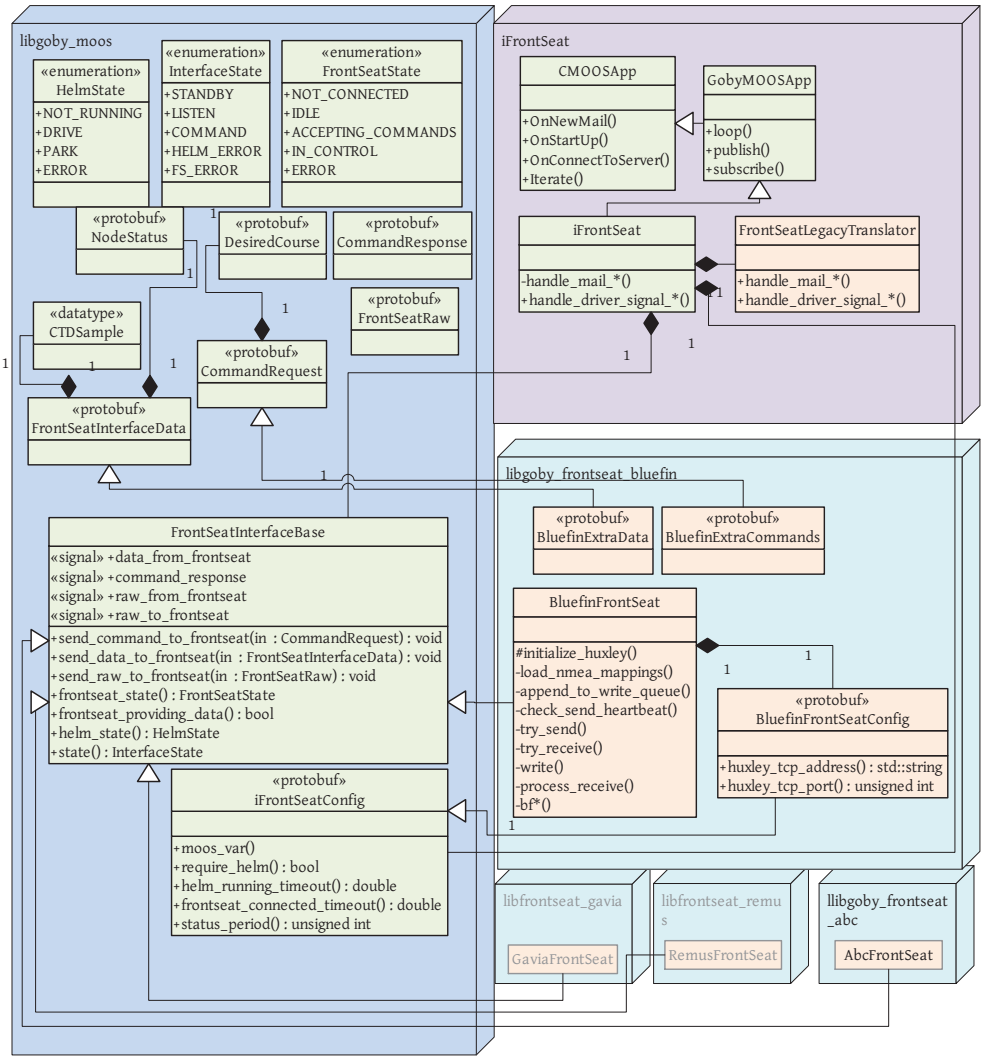

Figure 4.3: Structure diagram of iFrontSeat and supporting libraries. Green components are implemented once and used by all the drivers (see section 4.7.2). Pink components need to be modified/implemented for each specific driver (see section 4.7.3).

```
iFrontSeat --example_config
```

Many of these parameters can be left to their defaults, except for special cases and advanced usages.

This was the configuration of iFrontSeat at the time of writing this technical report. Be sure to check `iFrontSeat --example_config` for the latest possible valid configuration.

```
ProcessConfig = iFrontSeat
{
  common {
// configuration common to all Goby Applications
// ...
    }
  }
  require_helm: true
  helm_running_timeout: 10
  frontseat_connected_timeout: 10
  status_period: 5
  moos_var {
    prefix: "IFS_"
    raw_out: "RAW_OUT"
    raw_in: "RAW_IN"
    command_request: "COMMAND_REQUEST"
    command_response: "COMMAND_RESPONSE"
    data_from_frontseat: "DATA_IN"
    data_to_frontseat: "DATA_OUT"
    status: "STATUS"
  }
  exit_on_error: false
  legacy_cfg {
    subscribe_desired: true
    subscribe_ctd: false
    subscribe_acomms_raw: false
    pub_sub_bf_commands: false
    publish_nav: true
    publish_fs_bs_ready: false
  }

  // vehicle driver specific configuration
  // ...
}
```

The configuration for iFrontSeat has three main parts:

1. The `common` configuration which is the same for all Goby MOOS applications. Please see section 4.1 for details. Setting `verbosity: DEBUG2` is useful for debugging (and also `show_gui: true`, which displays an NCurses screen with useful debugging information).
2. The configuration for the shared MOOS side components, described below in this section.
3. The vehicle driver specific configuration, described in Chapter 4.7.3.

The configuration for the shared MOOS components is:

- `require_helm`: Require the IvP Helm even for a listening mission where the frontseat is in control (default=true).
- `helm_running_timeout`: If `require_helm: true`, how long (in seconds) to wait for the IvP Helm to start before moving to the Helm Error state. (default=10)
- `frontseat_connected_timeout`: How long (in seconds) to wait for the Frontseat to be connected before moving to the Frontseat Error state. (default=10)
- `status_period`: Seconds between publishing the status of iFrontseat. The special value 0 disables posting of the status message (default=5).
- `moos_var`: Change the default values of the MOOS variables published or subscribed to by iFrontSeat. Throughout the manual, these defaults are referenced. If you change the values here, keep this in mind when reading the rest of the manual.
  - `prefix`: Prefix all MOOS variable names with this string (default="IFS_")
  - `raw_out`: variable used to post raw (e.g. NMEA-0183) messages from iFrontSeat to the vehicle frontseat. (default="RAW_OUT")
  - `raw_in`: variable used to post raw messages from the vehicle frontseat to iFrontSeat. (default="RAW_IN")
  - `command_request`: variable used for commands that iFrontSeat should request the vehicle frontseat to carry out. (default="COMMAND_REQUEST")

  - `command_response`: if supported by the frontseat driver, the variable used to post the result (success or failure with error information) of a given request. (default="COMMAND_RESPONSE")
  - `data_from_frontseat`: the variable used to post any navigation and/or sensor data from the frontseat. (default="DATA_IN")
  - `data_to_frontseat`: the variable used to post any data to be sent to the frontseat. (default="DATA_OUT")
  - `status`: the variable used to post the state of the frontseat, IvP helm, and iFrontSeat. (default="STATUS")

- `exit_on_error`: If true, exit the application if it enters one of the error states. Use only for debugging. (default=false)
- `legacy_cfg`: Numerous options to automatically convert legacy variables (e.g., from iHuxley) into the iFrontSeat messages. Generally new projects will not use any of these options and thus this configuration block can be omitted. See section 4.7.2 for details on which of these flags to enable if legacy compatibility is desired.

MOOS Variable Interface

The preferred way to use iFrontSeat is via the new `IFS_` set of variables. The contents of these string MOOS variables are the output of the `TECHNIQUE_PREFIXED_PROTOBUF_TEXT_FORMAT` translator explained section 4.3. Essentially, they are the `TextFormat` human-readable output of the Google Protocol Buffers messages defined in

```
goby/moos/protobuf/frontseat.proto
```

To get access to the C++ equivalent classes generated by the Protobuf C++ compiler (protoc), include this header:

```
#include "goby/moos/protobuf/frontseat.pb.h"
```

Do not parse these messages manually. You can automatically parse and serialize these values to and from the corresponding Protobuf C++ classes using the functions `serialize_for_moos` and `parse_for_moos`, which are declared in the header file:

```
#include "goby/moos/moos_protobuf_helpers.h"
```

The MOOS variables subscribed to by iFrontSeat include (note the names are configurable, the defaults are given here):

- `IFS_COMMAND_REQUEST`: Command from to give to the frontseat driver to be asked of the vehicle's computer. This is typically the desired course (heading, speed, and depth) of the vehicle. Other special commands may be defined by the specific vehicle driver. Protobuf Message type: `CommandRequest`.
- `IFS_DATA_TO_FRONTSEAT`: Data that must be passed to the frontseat driver. For example, the Bluefin AUVs require Conductivity-Temperature-Depth (CTD) measurements when the CTD is connected to the backseat computer. Protobuf Message type: `goby.moos.protobuf.FrontSeatInterfaceData`.

The MOOS variables published by iFrontSeat include:

- `IFS_COMMAND_RESPONSE`: Response to each command request, if a response is requested. Protobuf Message type: `goby.moos.protobuf.CommandResponse`.
- `IFS_STATUS`: The current state of the IvP Helm, the frontseat system, and the interface itself. Protobuf Message type: `goby.moos.protobuf.FrontSeatInterfaceStatus`.
- `IFS_DATA_FROM_FRONTSEAT`: Data from the frontseat driver. This may include navigation data (vehicle's current pose, speed, depth, latitude, longitude, etc), or other vehicle specific data. Protobuf Message type: `goby.moos.protobuf.FrontSeatInterfaceData`.
- `IFS_RAW_IN`: Raw communications packets (e.g. NMEA-0183) from the frontseat computer to iFrontSeat. Protobuf Message type: `goby.moos.protobuf.FrontSeatRaw`
- `IFS_RAW_OUT`: Raw communications packets (e.g. NMEA-0183) from iFrontSeat to the frontseat computer. Protobuf Message type: `goby.moos.protobuf.FrontSeatRaw`

Legacy MOOS Variable Interface

iFrontSeat aims to replace several existing pieces of software, and by necessity provides a number of features to ease transition. This functionality may change or be removed in future versions, so where possible please use the new `IFS_` variables. This legacy functionality is implemented in:

```
goby/src/apps/moos/iFrontSeat/legacy_translator.cpp
goby/src/apps/moos/iFrontSeat/legacy_translator.h
```

The transitional MOOS variables subscribed to by iFrontSeat include:

- If `subscribe_ctd: true`, then `CTD_CONDUCTIVITY` (siemens/meter), `CTD_TEMPERATURE` (degrees C), `CTD_PRESSURE` (decibars), `CTD_SALINITY` (unitless - practical salinity scale). These double values are buffered, then upon receipt of `CTD_TEMPERATURE` are converted into a `FrontSeatInterfaceData` message containing a `CTDSample` and published to `IFS_DATA_TO_FRONTSEAT`.
- If `subscribe_desired: true`, then `DESIRED_HEADING` (degrees), `DESIRED_SPEED` (meters/second), `DESIRED_DEPTH` (meters). These desired course values (typically published by pHelmIvP) are buffered and upon receipt of `DESIRED_SPEED` are converted to a `CommandRequest` and posted to `IFS_COMMAND_REQUEST`.
- If `pub_sub_bf_commands: true`, then `PENDING_SURFACE`: This double value posted by BHV_PeriodicSurface creates a `CommandRequest` of the special Bluefin type: `BluefinExtraCommands::GPS_REQUEST`. This allows the shallow water vehicles that require a GPS fix to occasionally come to the surface for GPS.
- If `subscribe_acomms_raw: true`, then `ACOMMS_RAW_INCOMING`, `ACOMMS_RAW_OUTGOING`: The raw NMEA-0183 feed from the WHOI Micro-Modem (or other acoustic modem) posted by the Goby-Acomms MOOS application pAcommsHandler. These are converted into a `FrontSeatInterfaceData` message containing the special Bluefin extension `BluefinExtraData.micro_modem_raw_in` or `BluefinExtraData.micro_modem_raw_out` and published to `IFS_DATA_TO_FRONTSEAT`. Bluefin still requires our raw Micro-Modem feed as a backup to the hardware tail-cone abort (which uses the Micro-Modem).

The transitional MOOS variables published by iFrontSeat include:

- If `publish_nav: true`, then `NAV_X` (meters), `NAV_Y` (meters), `NAV_LAT` (degrees), `NAV_LONG` (degrees), `NAV_Z` (meters, negative down), `NAV_DEPTH` (meters, positive down), `NAV_YAW` (degrees), `NAV_HEADING` (degrees), `NAV_SPEED` (meters/second), `NAV_PITCH` (radians), `NAV_ROLL` (radians), `NAV_ALTITUDE` (meters). All these double values are generated from the `FrontSeatInterfaceData` message when it contains `node_status` information.

- If `publish_fs_bs_ready: true`, then `BACKSEAT_READY`: Published as 1 (true) when the Helm State becomes `HELM_DRIVE`, otherwise 0 (false).
- If `publish_fs_bs_ready: true`, then `FRONTSEAT_READY`: Published as 1 (true) when the FrontSeat State becomes `FRONTSEAT_ACCEPTING_COMMANDS`, otherwise 0 (false).
- `GPS_UPDATE_RECEIVED`: Published as `Timestamp=double seconds since Unix` when a `BluefinExtraCommands::GPS_REQUEST` command responds successfully.

4.7.3 Vehicle Drivers

BluefinFrontSeat

The driver `BluefinFrontSeat` is designed for the Bluefin Robotics Standard Payload Interface (SPI) Version 1.8 and newer, which must be requested directly from Bluefin Robotics.

Knowledge of some details of Bluefin's SPI will be assumed here; please reference that document as needed while reading this section.

The configuration accepted by iFrontSeat for the `BluefinFrontSeat` driver is as follows:

```

  [bluefin_config] {
    huxley_tcp_address: ""
    huxley_tcp_port: 29500
    reconnect_interval: 10
    nmea_resend_attempts: 3
    nmea_resend_interval: 5
    allowed_nmea_demerits: 3
    allow_missing_nav_interval: 5
    heartbeat_interval: 1
    extra_bplog: ""
    send_tmr_messages: true
    disable_ack: false
    accepting_commands_hook: BFMSC_TRIGGER
  }
```

This configuration values are placed in the .moos file in the `ProcessConfig = iFrontSeat` block:

- `huxley_tcp_address`: IP address or domain name of the Huxley server machine.
- `huxley_tcp_port`: TCP port of the Huxley server. (default=29500)
- `reconnect_interval`: How many seconds to wait between reconnects if the Huxley server disconnects. (default=10)
- `nmea_resend_attempts`: Number of resend attempts for a given NMEA message (default=3)
- `nmea_resend_interval`: How many seconds to wait between resend attempts (default=5)
- `allowed_nmea_demerits`: Number of times Huxley can fail to acknowledge a NMEA message before we close the connection. (default=3)
- `allow_missing_nav_interval`: How many seconds to allow without `$BFNVG` before declaring frontseat not providing us data. (default=5)
- `heartbeat_interval`: How many seconds between heartbeats (`$BPSTS`). (default=1)
- `extra_bplog`: Additional Bluefin messages to enable logging for (e.g. for to send `$BPLOG,CMA,ON`, set this field to 'CMA'. This field can be repeated.
- `send_tmr_messages`: Send the BPTMR message with acoustic comms contents. This is required on certain vehicles outfitted with the WHOI Micro-Modem. Ask Bluefin for details about if they need the BPTMR message sent (default=true).
- `disable_ack`: If true, do not use the BFACK message. Set to true for vehicles without the BFACK support. Note that if this field is set to true, `IFS_COMMAND_RESPONSE` messages will not be posted.
- `accepting_commands_hook`: The mechanism by which the Bluefin frontseat indicates that it is ready to accept commands from iFrontSeat (and also by which it revokes control). The options are:
  - `BFMSC_TRIGGER`: If any BFMSC message is received, the frontseat state is set to `FRONTSEAT_ACCEPTING_COMMANDS`. If a BFMIS message is received with the word "Running" in the fourth field, the frontseat state is set to be `FRONTSEAT_IN_CONTROL`. Any other BFMIS sets the frontseat state to `FRONTSEAT_IDLE`.

- `BFMIS_RUNNING_TRIGGER`: If a BFMIS message is received with the word "Running" in the fourth field, the frontseat state is set to be `FRONTSEAT_ACCEPTING_COMMANDS`. Any other BFMIS sets the frontseat state to `FRONTSEAT_IDLE`.
- `BFCTL_TRIGGER`: If the third field is true, the frontseat state is set to `FRONTSEAT_ACCEPTING_COMMANDS`. Otherwise, it is set to `FRONTSEAT_IN_CONTROL`. Also, if a BFMIS message is received with the word "Running" in the fourth field, the frontseat state is set to be `FRONTSEAT_IN_CONTROL`. Any other BFMIS sets the frontseat state to `FRONTSEAT_IDLE`.

Writing a new driver

A tutorial on how to write a new driver is available in the Goby developers' documentation at [1].

4.8 iCommander

Deprecated. Use goby_liaison as a replacement. See section 4.5.

4.9 pREMUSCodec

Deprecated. DCCL has significant support for interoperating with the CCL protocol using pAcommsHandler with the libgoby_ccl_compat library. Please contact one of us (section 1.6) if you need help getting started with this functionality.

5 What's next

That's all for goby in Release 2.0. There's still a lot to do so keep tuned. If you want the bleeding edge, you can check out the Goby 3.0 branch with
`bzr checkout lp:goby/3.0`.

Here's what's on the horizon:

- Goby-Common: a general purpose interprocess and interplatform communication based on messaging schemes drawn both from the existing marine robotics and global open source communities. The focus is on a high degree of runtime reliability and collaboration between development communities. For advanced users, it provides a transport layer built on ZeroMQ (which supports 20+ languages including C, C++, Java, .NET, Python, and major platforms) for communicating over reliable multicast (PGM) using one or more existing (e.g. MOOS, LCM, Protobuf, CCL, DCCL, ...) messaging schemes. Goby does not mandate a programming language, a messaging scheme, or a development system and thus intends to tie together groups with different goals, styles, and rules. Furthermore, Gateways can be written to interface the ZeroMQ based Goby transport with the native transport systems used by other architectures (e.g. MOOSDB, LCM multicast).
- Goby-PB: The Google Protocol Buffers / C++ implementation of Goby-Common. For introductory users, it provides an "template" application in C++ that allows object-based messaging (based on Google Protocol Buffers) between processes and platforms without any concern for serialization, routing, sockets, and so on.

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

Glossary

autonomy architecture loosely 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

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.". 3

Bibliography

[1] Goby Developers, "Goby underwater autonomy project documentation." [Online]. Available: http://gobysoft.org/doc/2.0

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

[3] M. R. Benjamin, "The MOOS-IvP uField Toolbox for Multi-Vehicle Operations and Simulation," Massachusetts Institute of Technology, Tech. Rep. 12.2, 02 2012.

[4] 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

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

[6] "∅MQ, the intelligent transport layer." [Online]. Available: http://www.zeromq.org/