Goby Underwater Autonomy Project (version 1)

pAcommsHandler Tutorial for MOOS-DAWG 2011

<https://launchpad.net/goby>

Contents

1 Background

Please download the referenced documents and example missions from http://gobysoft.com/dl/moosdawg2011_tutorial.tgz. This tutorial will cover version 1 of Goby. Read http://gobysoft.org/wiki/InstallingGoby to see how to install Goby version 1 (which includes `pAcommsHandler`). If at any time you want to know what version of Goby `pAcommsHandler` is using, type `pAcommsHandler --version` on the command line.

`pAcommsHandler`[1] is the MOOS interface to a collection of four C++ libraries (goby-acomms[2]) that handle various aspects of single-hop acoustic networking. Currently `pAcommsHandler` only supports the WHOI Micro-Modem[3], but the design is abstracted such that introducing support for a different acoustic modem (or even different physical channel) would only involve a single subclass.

The four libraries are (listed in order of a traditional networking "stack" from most to least abstract):

- *libdccl* - highly compact data marshalling provided by the Dynamic Compact Control Language.
- *libqueue* - prioritized buffering of DCCL (and legacy CCL[4]) messages for sending when dictated by the Medium Access Control (*libamac*)
- *libamac* - Time Division Multiple Access (TDMA) Medium Access Control. Supports both centralized and decentralized modes, with optional peer discovery.
- *libmodemdriver* - abstract driver and WHOI Micro-Modem realization. Handles serial communications with the physical modem.

Each of these libraries can be used alone or together as part of a non-MOOS networking application. The latter situation is not within the scope of this tutorial, but for reference see http://gobysoft.com/doc/1.0.

Copies of many of the references noted in this document are given in the `references` folder of `pAcommsHandler_tutorial`. Many example (real usage) DCCL messages can be found in `goby/share/xml` and `missions-lamss/data/acomms`.

[1] http://gobysoft.com/#/software/MOOS-IvP
[2] https://launchpad.net/goby
[3] http://acomms.whoi.edu
[4] http://acomms.whoi.edu/ccl/

2 Philosophy

2.1 Acoustic Communications are slow

Do not take too much from traditional networking (e.g. TCP/IP), some things we are doing here for acoustic communications (hereafter, acomms) are unconventional from the approach of networking on electromagnetic carriers (hereafter, EM networking). The difference is a result of a vast spread in the expected throughput of a standard internet hardware carrier and acoustic communications. For example, an optical fiber can put through greater than 10 Tbps over greater than 100 km, whereas the WHOI acoustic Micro-Modem can (at best) do 5000 bps over several km. This is a difference of thirteen orders of magnitude for the bit-rate distance product. Or (roughly speaking), you can transfer the contents of the books in the Library of Congress over an optical link in less than the amount of time it would take to send a simple plain text email over an acoustic link.

2.2 Efficiency to make messages small is good: use your knowledge of the data

Extremely low throughput means that essentially every efficiency in bit packing messages to the smallest size possible is desirable. The traditional approach of layering (e.g. TCP/IP) creates inefficiencies as each layer wraps the message of the higher layer with its own header. See RFC3439 chapter 3 ("Layering Considered Harmful") for an interesting discussion of this issue[1]. Thus, the "layers" of goby-acomms are more tightly interrelated than TCP/IP, for example. Higher layers depend on lower layers to carry out functions such as error checking and do not replicate this functionality.

The Dynamic Compact Control Language makes use of the data's physical origin to encode messages losslessly as small as possible. The designer of the message must input upper and lower bounds on numeric values, which should be physically derived (e.g. a sensible bound on salinity in all the world's oceans might be [25, 45]). Starting in version 2 of Goby, fields can use (lossy or lossless) custom user-defined encoders that make even better use of this kind of data inspection to create smaller, but equally useful, messages. Analogously. think anything from MP3 (which uses psychoacoustics to throw away sounds we will not hear) for ocean data.

[1] http://tools.ietf.org/html/rfc3439#page-7

2.3 Total throughput unrealistic: prioritize data

The second major difference stemming from this bandwidth constraint is that total throughput is often an unrealistic goal. The quality of the acoustic channel varies widely from place to place, and even from hour to hour as changes in the sea affect propagation of sound. This means that it is also difficult to predict what one's throughput will be at any given time.

These two considerations manifest themselves in the goby-acomms design as a priority based queuing system for the transport layer. Messages are placed in different queues based on their priority (which is determined by the designer of the message). This means that the channel is always utilized (low priority data are sent when the channel quality is high) but important messages are not swamped by low priority data. In contrast, TCP/IP considers all packets equally. Packets made from a spam email are given the same consideration as a high priority email from the President. This is a trade-off in efficiency versus simplicity that makes sense for EM networking, but does not for acoustic communications.

2.4 Despite all this, simplicity is good

The "law of diminishing returns" means that at some point, if we try to optimize excessively, we will end up making the system more complex without substantial gain. Thus, goby-acomms makes some concessions for the sake of simplicity:

Numerical message fields are bounded by powers of 10, rather than 2. Humans deal much better with decimal than binary. User data splitting (and subsequent stitching) is not done. This is a key component of TCP/IP, but with the number of dropped packets one can expect with acomms, at the moment this does not seem like a good idea. The user is expected to provide data that is smaller or equal to the packet size of the physical layer (e.g. 32 - 256 bytes for the WHOI Micro-Modem).

2.5 Build on high quality open source projects

Rather than reinvent the wheel (and probably not do as good a job doing it), `pAcommsHandler` relies on a number of high quality (and generally well documented) projects (as well as obviously MOOS / MOOS-IvP):

- version control: bzr [2]
- build system: cmake [3]

[2] http://wiki.bazaar.canonical.com/Documentation
[3] http://www.cmake.org/cmake/help/cmake2.6docs.html

- general purpose libraries: boost [4]
- terminal GUI library: ncurses [5]
- XML parsing library: Xerces-C [6]
- Object-based marshalling: Google Protobuf [7]
- asynchronous networking and serial communications library: asio [8]
- cryptography: crypto++ [9]

[4] http://www.boost.org/doc/libs/1_34_0
[5] http://www.c-for-dummies.com/ncurses/
[6] http://xerces.apache.org/xerces-c/program-3.html
[7] http://code.google.com/apis/protocolbuffers/docs/overview.html
[8] http://think-async.com/Asio/asio-1.4.1/doc/
[9] http://www.cryptopp.com

3 Configuration

Within MOOS, the runtime configuration for all four components of goby-acomms are set in the same `pAcommsHandler` ProcessConfig block of the .moos file. `pAcommsHandler` (as with all Goby MOOS applications in Goby version 1 and later) does not use the MOOS `ProcessConfigReader` but rather read the ProcessConfig block using the Google Protobuf TextFormat representation. This allows configuration to have:

- A rich object representation including embedded objects
- Syntactic and type error checking.

`pAcommsHandler` will fail gracefully with a useful error message if any problems are found with configuration. The syntax is similar to `ProcessConfigReader`, but strings must be quoted, and ':' is used instead of '=' to separate keys from values.

Sensible defaults for most values are available, so a bare (but working) block looks like:

```
ProcessConfig = pAcommsHandler
{
  modem_id: 1

  driver_type: DRIVER_WHOI_MICROMODEM
  driver_cfg
  {
    serial_port: "/dev/ttyS0"
    [MicroModemConfig.reset_nvram]: true
  }

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

  dccl_cfg
  {
    message_file { path: "../../../xml/simple_status.xml" }
  }

  queue_cfg
  {
    // XML definitions automatically copied from dccl_cfg
```

```
27    }
28  }
```

for a full listing of available .moos file parameters, type `pAcommsHandler --example_config` or see the Goby Version 1 User Manual[1].

The reason this ProcessConfig block is small is because the bulk of the configuration for encoding / decoding the DCCL messages (*libdccl*) and their subsequent buffering (*libqueue*) is done in one or more XML files and are included using one or more `message_file {}` lines in the .moos file. Creating this Message XML files will be the subject of chapter 6.

[1] http://gobysoft.org/dl/goby1-user-manual.pdf

4 Complementary Applications

Several MOOS applications are especially useful for projects using `pAcommsHandler`.

4.1 iModemSim

Roughly simulates the AUV 0.92.0.85 revision of the WHOI Micro-Modem firmware by providing a virtual "ocean" on UDP broadcast. Only useful for packet rate 0 (FSK / 32 bytes) as the simulation for higher rates has serious errors. Introduces realistic delays and questionable selective dropping of packets (based on range). See `lamss/src/moos/iModemSim/doc`[1] for details on using this process.

Alternatively to a software simulator, WHOI makes a modem emulator box that has two modems connected by a coax cable instead of power electronics and transducers. This is the best choice for software development, but you must remember that packet delays and dropouts are not simulated.

`pAcommsHandler` will work with iModemSim, and all WHOI Micro-Modem firmware revisions since AUV 0.92.0.85. I currently test primarily on AUV 0.93.0.30 and newer and highly recommend upgrading to these newer revisions as I know of a number of bugs in 0.92.0.85.

4.2 iCommander

`iCommander` provides a ncurses terminal-based graphical user interface (GUI) for a human to enter fields of an DCCL messsage (as given by its Message XML structure file). This is primarily useful for generating commands for a vehicle, and requires no reconfiguration when the structure (XML) of the message is changed (to add / change some field in the message).

[1] Get the LAMSS project from `http://launchpad.net/lamss`

5 Example 1: Designing a simple Message (or learning libdccl)

I find the best way to learn something is to dive right in with an example. We will start off with a very simple example: sending the boolean command "DEPLOY" from one MOOS community (representing the topside or command computer on board the ship) to another (representing the vehicle). This is provided as `example1` within the `pAcommsHandler_tutorial` folder.

We always begin with the standard XML declaration and root tag (which is `<message_set>`):

```
<?xml version="1.0" encoding="UTF-8"?>
<message_set>
```

A message set may contain any number of messages. We will start with just one:

```
  <message>
```

We need to pick a name for our message. This is used for debugging, but is also a valid unique key to the message. Thus, no two messages in a network should have the same name.

```
    <name>deploy_tutorial</name>
```

Now, we must decide when our messages should be encoded. We can either create them on some time interval using the newest variables in the MOOSDB or triggered by a publish to some MOOS variable. Here we choose the latter.

```
    <trigger>publish</trigger>
    <trigger_moos_var>DEPLOY_ALL</trigger_moos_var>
```

Because `pAcommsHandler` (by design) does not provide facilities for splitting messages into packets, we must specify a maximum size our message can be. Generally we want to make this the same as the physical frame size for the Micro-Modem rate that we plan to use. Since we'd like to use this with the lowest rate (FSK / 0) which has a single frame of 32 bytes:

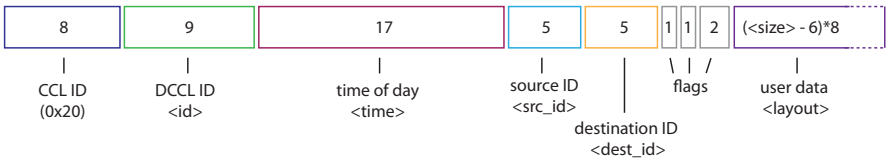

Figure 5.1: Layout of the DCCL header, showing the fixed size (in bits) of each header field.

```
<size>32</size>
```

We can specify the values for some parts of the fixed size header, namely the DCCL ID (<id>), source modem ID (<src_id>), destination modem ID (<dest_id>), and time (<time>). Time is given as seconds since UNIX 1/1/1970 0:00 UTC, consistent with MOOS conventions. The structure and fields of the header are given in Fig. 5.1. The DCCL ID (<id>) must be unique within a network.

```
<header>
  <id>1</id>
</header>
```

Now for the part of the message we get to control most. We can form the message from any number of <string>s, <int>s, <float>s, <enum>s, <bool>s, and <hex>s. The difference from these DCCL types and normal C++ (or other programming language) types is that DCCL types are strictly bounded (maximum / mininum values) for each instantiation. We'll see this later. For now, we use the simple <bool>:

```
<layout>
  <bool>
    <src_var>DEPLOY_ALL</src_var>
    <name>value</name>
  </bool>
</layout>
```

Technically, this is a "tribool" (true / false / unknown) as all DCCL types support a "undefined", "out of bound", or "NaN" value as well as the values given in their range. This undefined value is always encoded as binary 0. <src_var> indicates

which MOOS variable to grab the value for this field from when encoding. If omitted, the <trigger_var> is used, so in this case I have redundantly defined it (since the <trigger_var> and <src_var> are both DEPLOY_ALL).

Finally, we decide what the recipient of the message should do. We can have any number of publishes to the MOOSDB using some or all of the message fields (message variables or "message_vars"). A <message_var> is the <name> of any of the types given in <layout> (<string>, <int>, etc.):

```
    <on_receipt>
      <publish>
        <moos_var>DEPLOY</moos_var>
        <format>%1%</format>
        <message_var>value</message_var>
      </publish>
    </on_receipt>
```

Now, if all goes well, if we publish "DEPLOY_ALL: true" in one MOOS community, all those within broadcast range of the sender will decode and publish "DEPLOY: true" in their communities. We'll deal with directed (single destination) messages later.

We can test our message for syntax (against the DCCL XML schema) and logic errors (such as <max> less than <min>) using the analyze_dccl_xml tool. It will also give the exact sizes used by each message variable:

```
cd example1
analyze_dccl_xml deploy.xml
```

Now, you can run the full MOOS example and use iCommander to publish the DEPLOY_ALL command.

```
./README
```

6 Example 2: Replacing pMOOSBridge with pAcommsHandler in a multi-vehicle simulation (or learning libqueue and more libdccl)

6.1 Augmenting our DEPLOY message (deploy.xml)

Looking at shoreside.moos, we see that there are several commands that `pMOOSBridge` passes across to the vehicles. We have already replaced these lines in Example 1:

```
1    SHARE = [DEPLOY_ALL]  -> henry @ localhost:9201 [DEPLOY]
2    SHARE = [DEPLOY_ALL]  -> gilda @ localhost:9202 [DEPLOY]
```

We will need a similar message to enable the IvP Helm on Deploy and replace these lines:

```
1    SHARE = [MOOS_MANUAL_OVERIDE_ALL] -> henry @ localhost:9201 [MOOS_MANUAL_OVERIDE]
2    SHARE = [MOOS_MANUAL_OVERIDE_ALL] -> gilda @ localhost:9202 [MOOS_MANUAL_OVERIDE]
```

Since it is similar, I added this message (manual_override_tutorial) to deploy.xml. Also new is a <`queuing`> block:

```
1    <queuing>
2       <ttl>300</ttl>
3       <value_base>100</value_base>
4       <max_queue>1</max_queue>
5    </queuing>
```

This tells *libqueue* to how to prioritize different queues when the vehicle has a chance to send an acoustic message. The time to live (<`ttl`>) governs both when a message is discarded if not sent (in seconds) *and* how quickly the priority of that queue grows. The base value governs the overall priority of the queue. Every DCCL type has its own queue but all messages within a queue are considered equal.

The queue with the highest priority P at time t is popped and the next message is sent. Priorities for each queue are computed as follows:

$$P(t) = V_{base}\frac{(t - t_{last})}{ttl} \tag{6.1}$$

where ttl is <ttl>, V_{base} is <value_base> and t_{last} is the time a message was last sent from this queue.

The basic idea is that messages with a shorter time-to-live and/or a higher base value are sent first.

6.2 Adding a status message for NODE_REPORTs (node_report.xml)

To report the position and other stats of the vehicles to the topside, we use the NODE_REPORT message, which is a structure of key=value comma-delimited pairs. pAcommsHandler understands how to read such strings, looking for the <name> of a given field as the key within the string.

An example NODE_REPORT looks like

```
NAME=gilda,TYPE=KAYAK,MOOSDB_TIME=1.01,UTC_TIME=1282599270.36,X=80.00,Y=0.00,\
LAT=42.358418,LON=-71.086479,SPD=0.00,HDG=180.00,YAW=180.00000,DEPTH=0.00,\
LENGTH=4.0,MODE=DISENGAGED,ALLSTOP=unknown
```

Given this, I need to define a number of numeric fields such as

```
      <float>
        <name>X</name>
        <precision>1</precision>
        <min>-1000</min>
        <max>1000</max>
      </float>
```

<precision> is the number of decimal places to keep, and can be negative. <min> and <max> should be self-explanatory. The encoded size (in bits) to store this field is directly related to how loose these bounds are, so you want to make them as tight as (reasonably) possible. Rather than 32 bits usually used to store a float, the variable above only uses 15 bits because of bounding.

The other new thing here are algorithms, for example:

```
<int algorithm="angle_0_360">
```

Before encoding, any number is wrapped into the bounds [0, 360). That is, -170 is converted to 190 before encoding.

`pAcommsHandler` provides a number of useful algorithms that applied to the value *before* encoding (if given in the message variable declaration as above) or *post* decoding (if given in the <`publish`><`message_var`> section). An algorithm and its inverse can be used to provide non-linear encoding of numeric fields (such as logarithmic, e.g. Decibels). For all supported algorithms see the Goby Version 1 User Manual[1].

6.3 Commanding new loiter positions (update_loiter.xml)

The final message needed was to command the vehicles to change their loiter positions. This message is in update_loiter.xml and should be fairly clear in light of what we've learned so far.

The main difference is that these messages are directed. That is, they have a destination that is not broadcast. This is defined by the header message variable <`dest_id`>:

```
    <header>
  ...
      <dest_id>
        <name>destination</name>
      </dest_id>
    </header>
```

DCCL is looking for the key "destination" in UP_LOITER_COMMAND. If UP_LOITER_COMMAND was "destination=3,center_x=200,center_y=300", the message would be sent to modem_id 3.

Now we're ready to run the mission:

```
cd example2
./README
```

[1] http://gobysoft.org/dl/goby1-user-manual.pdf

7 Designing your Network (Medium Access Control, libamac)

The Medium Access Control schemes provided by libamac are based on Time Division Multiple Access (TDMA) where different communicators share the same bandwidth but transmit at different times to avoid conflicts. Time is divided into slots and each vehicle is given a slot to transmit on. The set of slots comprising all the vehicles is referred to here as a cycle, which repeats itself when it reaches the end. The three variations on this scheme provided by libamac are:

- Decentralized (Auto discovery) TDMA (`mac_cfg { type: MAC_AUTO_DECENTRALIZED }`): Each vehicle has a single slot in the cycle on which it transmits. Each vehicle initiates its own transmission at the start of its slot. Collisions are avoided by each vehicle following the same rules about slot placement within the time window (based on real time of day). This scheme requires that each vehicle have reasonably accurate clocks (perhaps better than +/- 0.5 seconds). Vehicles are discovered by shifting a blank time in each cycle in a pseudorandom place based on their knowledge of the world and the time of day. If a new vehicle is heard from during the blank, it is added to the listening vehicle's knowledge of the world and hence their cycle. *If you have synchronized clocks, this is the easiest MAC to configure and works best with a small number of vehicles.*

  Example (at time 135 and beyond the two vehicles are synched)

```
ProcessConfig = pAcommsHandler
{
...
  mac_cfg {
    type: MAC_AUTO_DECENTRALIZED
    rate: 0
    slot_seconds: 15
  }
}
```

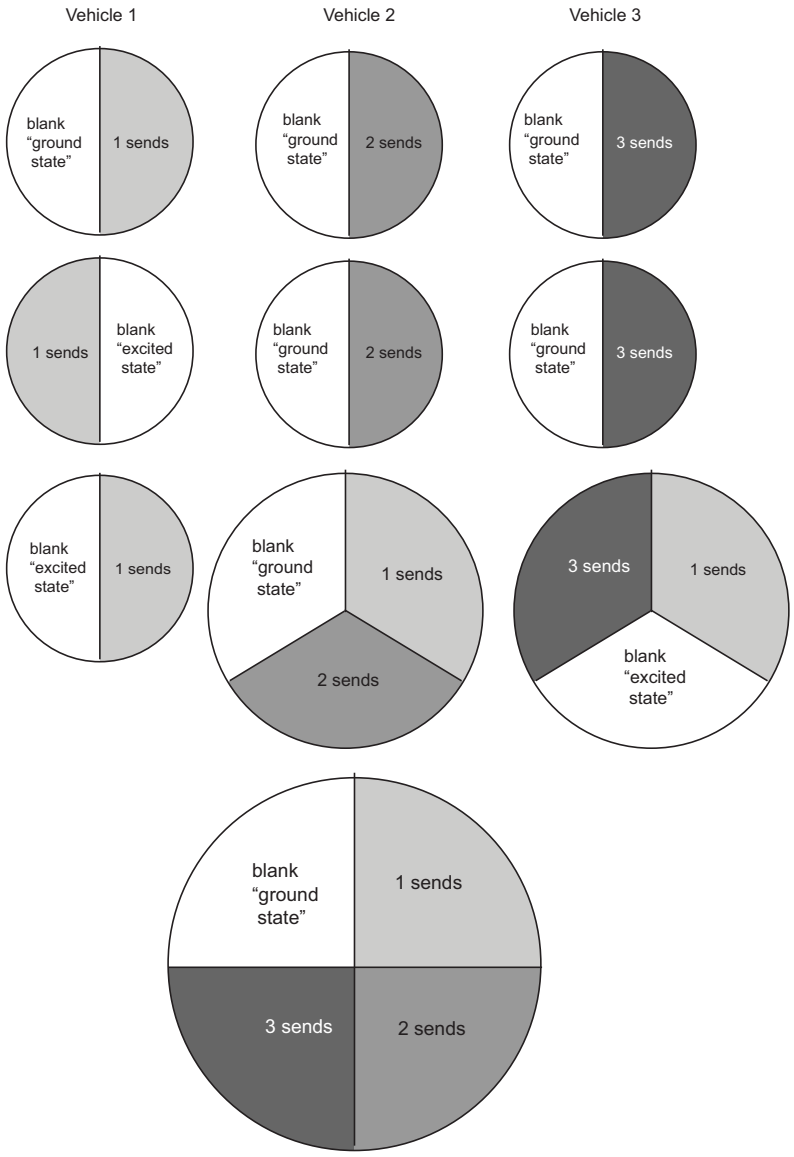

Figure 7.1: Graphical example of auto discovery for three nodes launched at the same time. Each circle represents the vehicle's cycle at each time step (represented by horizontal rows) based on the vehicle's current knowledge of the world. In the first row, all vehicles only know of themselves and put the blank slot in the last slot; thus, all communications collide and no discoveries are made. In the second row, vehicle 1's blank is moved (by pseudo-chance) to the penultimate (first) slot, so vehicles 2 and 3 discover 1. Then, in the third row vehicles 2 and 3 are discovered by the others because vehicle 3 moves its blank slot. By the fourth row all vehicles have discovered the others and continue to transmit without collision following the cycle diagrammed on this row.

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

- Decentralized (Fixed Cycle) TDMA (`mac_cfg { type: MAC_FIXED_DECENTRALIZED }`): Similar to the Auto discovery TDMA but you set a fixed cycle on all the vehicles before they go in the water. This is less flexible to taking vehicles in and out of the water, but guarentees no collisions during the discovery time. *If you have synchronized clocks and you cannot tolerate collisions or want more control over the cycle, this is the best choice.*

```
ProcessConfig = pAcommsHandler
{
...
  mac_cfg {
    type: MAC_FIXED_DECENTRALIZED

    // dest of '-1' (the default) means to query the queuing layer for the next destination
    // before sending a message
    // this is generally the best choice for decentralized MAC
    // (unless a node always sends to the same recipient)
    slot { src: 1  dest: -1  rate: 0  type: SLOT_DATA  slot_seconds: 10 }
    slot { src: 2  dest: -1  rate: 0  type: SLOT_DATA  slot_seconds: 10 }
    slot { src: 3  dest: -1  rate: 0  type: SLOT_DATA  slot_seconds: 10 }
  }
}
```

time	vehicle 1	vehicle 2	result
0	send	listen for 1	success
15	listen for 2	send	success
30	send	listen for 1	success
45	listen for 2	send	success

- Centralized Polling (`mac_cfg { type: MAC_POLLED }` on the topside, `mac_cfg { type: MAC_NONE }` on the vehicles): The TDMA cycle is set up and operated by a centralized modem ("poller"), which is usually the modem connected to the vehicle operator's topside. The poller initiates each transmission and thus the vehicles are not required to maintain synchronous clocks. *If you cannot guarantee synchronous clocks or you want runtime control of the TDMA cycle, this is the best MAC to use.*

```
1  //topside.moos
2  ProcessConfig = pAcommsHandler
3  {
4  ...
5    mac_cfg {
6      type: MAC_POLLED
7
8      slot { src: 1  dest: -1  rate: 0  type: SLOT_DATA  slot_seconds: 10 }
9      slot { src: 2  dest: 0  rate: 0  type: SLOT_DATA  slot_seconds: 10 }
10     slot { src: 3  dest: 0  rate: 0  type: SLOT_DATA  slot_seconds: 10 }
11   }
12 }
13
14 //vehicle.moos
15 ProcessConfig = pAcommsHandler
16 {
17 ...
18   mac_cfg {
19     type: MAC_NONE
20   }
21 }
22
```

time	topside 1	vehicle 2	vehicle 3	result
0	send	listen for 1	listen for 1	success
15	command 2 send to 1	send	listen for 2	success
30	command 3 send to 1	listen for 3	send	success

Whichever MAC scheme you use, you must pick a slot length in seconds (`slot_seconds: #`. 10 to 15 seconds is a good choice, depending on how synchronized your clocks are and how far you expect vehicles to be apart.

For a separation of 1500 meters, the math looks like this where t is the slot time:

$$\begin{aligned} t &= 1.5 \text{ [cycle init \$CCCYC]} \\ &\quad +1 \text{ [travel time]} \\ &\quad +3.5 \text{ [data packet]} \\ &\quad +1 \text{ [travel time]} \\ &\quad +1.5 \text{ [ack \$CAACK]} \\ &\quad +1 \text{ [travel time]} \\ &= 9.5 \text{ seconds} \end{aligned}$$

and you'll want to add a little time for CPU processing on both ends (mostly within the modem), DRQ, etc. (Advanced) If you're using rate 2,3, or 5 (PSK) and you're using the Decentralized Polling, the cycle time is shorter by the cycle init time and propagation (2.5 seconds above) since this is sent with the message itself.

You also pick the rate (`rate=#`) to send with, which is an integer 0, 2, 3, or 5 that indicates the bit rate. Lower rates are less susceptible to error and dropouts, but carry less data.

- Rate 0 - 1 frame of 32 bytes = 32 bytes total
- Rate 2 - 3 frames of 64 bytes = 192 bytes total (requires coprocessor)
- Rate 3 - 2 frames of 256 bytes = 512 bytes total (requires coprocessor and receiving array)
- Rate 5 - 8 frames of 256 bytes = 2048 bytes total (requires coprocessor and receiving array)

Rates 0 and 2 are best to start out with, especially if you don't own a buoy with a receiving array.

8 Configuring the driver (libmodemdriver)

If your modem is connected directly to a serial port, all you need to specify in the .moos file is

```
driver_cfg {
  serial_port = "/dev/ttyS0"
}
```

All NVRAM (configuration) parameters of the modem are reset to factory defaults on launch, except SRC which is set to the `modem_id` given). If you need to configure other settings, use

```
[MicroModemConfig.nvram_cfg]: "SNR,1"
[MicroModemConfig.nvram_cfg]: "GPS,1"
```

See the Micro-Modem Software Interface Guide[1] for details on all of these parameters.

If you need the raw NMEA stream from the modem, subscribe to ACOMMS_NMEA_IN. If you need to directly control the modem, publish to ACOMMS_NMEA_OUT.

[1] http://acomms.whoi.edu/documents/uModem%20Software%20Interface%20Guide.pdf

9 Advanced features of note

The following features are described in more detail in the Goby Version 1 User Manual[1]. and/or in the Goby documentation[2].

9.1 Initializers

If you need to initialize any MOOS variable at startup with `pAcommsHandler`, use the syntax (within the `common {}` subblock of the `ProcessConfig` block:

```
initializer { type: INI_STRING moos_var: "NAME1" sval: "string_value" }
initializer { type: INI_DOUBLE moos_var: "NAME2" dval: 3.23452 }
```

This should probably be a feature of MOOS, but isn't, so we make it available here.

9.2 Manipulators

`pAcommsHandler` supports loading XML files with certain manipulators disabling or enabling additional features. Instead of

```
message_file { path: "deploy.xml" }
```

you can write

```
message_file { path: "deploy.xml"
               manipulator: NO_ENCODE }
```

and the messages defined in deploy.xml will never be encoded on that vehicle (even if the <`trigger`> conditions are met).

Another example is

```
message_file { path: "deploy.xml"
               manipulator: NO_QUEUE
               manipulator: LOOPBACK }
```

[1] http://gobysoft.org/dl/goby1-user-manual.pdf
[2] http://gobysoft.com/doc/1.0

which means that all messages generated locally are looped-back (i.e. decoded immediately) and are not queued to send acoustically. This is somewhat analogous to localhost (the `lo` interface on Linux).

Note that regardless of whether you use `loopback` or not, any messages addressed to oneself (i.e. source and destination match `modem_id`) are immediately looped-back and not sent to the modem.

9.3 Crytography

AES encryption is enabled by providing some passphrase

```
dccl_cfg {
  crypto_passphrase: "supersecret!"
}
```

All receiving nodes must have the same crypto_passphrase or they will decode messages completely incorrectly. You are, of course, responsible for the security of the .moos file by using appropriate filesystem permissions.

9.4 Modem ID Lookup file

For some `pAcommsHandler` DCCL algorithms (namely `modemid2name`, `modemid2type` and `name2modemid`), a lookup table must be provided. The file is a simple comma delimited file like:

```
// modemidlookup.txt
// modem id, vehicle name (should be community name), vehicle type
0, broadcast, broadcast
1, shoreside, topside
2, gilda, kayak
3, henry, kayak
```

and is included in the .moos file with

```
modem_id_lookup_path: "modemidlookup.txt"
```