# Day 2: Communications

Before you start, update your branch to post-lecture2 which includes all the code changes I made during the lecture today:

```
cd goby3-course
git fetch
git checkout post-lecture2
```

I would advise working on your own branch and committing as you go along. You can fork goby3-course to your personal Github account (assuming you're familiar with this), or just commit to a local branch:

```
# use post-lecture2 as a starting point
git checkout post-lecture2
# create a new branch called "homework2" to do your work
git checkout -b homework2
# do some work, then
git add
git commit
```

## **Assignment 1:**

Goal: Within the Trail example, create a Command message and publish it from the topside so that the USV can subscribe to it over the intervehicle layer.

#### Task:

Within the Trail example, we are currently only sending the NavigationReport message on the intervehicle layer. While this allows us to see where our vehicles are, we have no way of changing their behavior.

In preparation for tomorrow's lecture on Autonomy, this assignment will see us create a DCCL Command message, publish it on the topside, and subscribe to on the USV.

Steps:

- Create a goby3\_course::dccl::USVCommand message, defined in DCCL and using the DCCL msg id 126. At a minimum this message should contain:
  - o a timestamp
  - o a desired Mission state enumeration (WAYPOINTS, POLYGON)
  - o (for polygon): number of sides
  - o (for polygon): radius (meters)
- Create a group (perhaps "usv\_command" with numeric id broadcast\_group) for this message.
- · Create a testing application to run on the topside which will publish this message (intervehicle) on some regular interval (e.g. every 60 seconds).
- Subscribe to this message on the USV (probably in the existing goby3\_course\_usv\_manager is fine, or you could create a new application to handle commands). Things to consider:
  - ack\_required: true or false?
  - o max\_queue:?
- Run the Trail example (./all.launch) and ensure you're receiving the commands by examining the glog output of the subscribing process (goby3\_course\_usv\_manager or your new USV command handler). Make sure to enable VERBOSE output on glog by adding -v to the appropriate command lines of the launch file(s), or alternatively increase the verbosity on the log files written to goby3-course/logs by changing the log\_file\_verbosity = setting within launch/trail/config/usv.pb.cfg.py, auv.pb.cfg.py or topside.pb.cfg.py.

(optional) if you want things to slow down a bit, you can run at real time speeds by setting (before launching all.launch):

```
# launch/trail/config/common/sim.py
warp=1
```

Tomorrow we will work on the last step of connecting this to the autonomy system (pHelmIvP).

### **Bonus Task**

Add this publication to goby liaison so you can publish your message from the Commander tab (instead of your testing application):

```
# launch/trail/config/templates/liaison.pb.cfg.in
# ...

pb_commander_config {
    load_protobuf {
        name: "goby3_course.dccl.USVCommand"
        publish_to {
            group: "usv_command"
            group_numeric: 0
            layer: LAYER_INTERVEHICLE
        }
    }
}
```

Now you can load this command and send it from http://localhost:50000/?\_=/commander

Ensure that after you send it that you can still see your command show up on the USV side.

## Assignment 2:

Goal: Add a health monitoring process to the USV based on our intervehicle1/publisher application, and extend it to use the goby\_coroner output to determine whether the USV is in "GOOD" or "FAILED" health.

Task:

#### Code

Using the code in src/bin/intervehicle1/publisher as a starting point, make a new application called goby3\_course\_usv\_health\_monitor.

We are going to use the existing <code>goby\_coroner</code> tool to tell us whether our applications are all running (at a minimum) and then determine if all our code is running that the USV is in "GOOD" health, or if not, it's "FAILED":

Taking a look at the interface file for goby\_coroner

```
# goby3/build/share/goby/interfaces/goby_coroner_interface.yml
application: goby_coroner
interprocess:
publishes:
    - group: goby::health::report
    scheme: PROTOBUF
    type: goby::middleware::protobuf::VehicleHealth
        thread: goby::apps::zeromq::Coroner
# ...
```

we see that it publishes a vehicleHealth Protobuf message to the goby::health::report group. The group and message are defined in:

```
#include <goby/middleware/coroner/groups>
// generated from goby/middleware/protobuf/coroner.proto
#include <goby/middleware/protobuf/coroner.pb.h>
```

Within the <a href="mailto:goby3\_course\_usv\_health\_monitor">goby\_coroner</a>. Based on this information, publish the HealthStatus message on <a href="mailto:information">information</a>, subscribe to the <a href="mailto:vehicle-tealth">vehicle-tealth</a> message from <a href="mailto:goby\_coroner">goby\_coroner</a>. Based on this information, publish the HealthStatus message on <a href="mailto:information">information</a>, publish the <a hr

Update the goby3\_course\_topside\_manager to subscribe to this health message, and report the USV's health via glog.

## Configuration

Once you have the code done, you'll need to insert your configuration and add to the appropriate launch files.

Create:

- $\bullet \ \ launch/trail/config/templates/goby 3\_course\_usv\_health\_monitor.pb.cfg. in$ 
  - o \$app\_block will be expanded to the app {} section
  - \$interprocess block will be expanded to the interprocess {} section
- launch/trail/config/templates/goby\_coroner.pb.cfg
  - o same as above for plock and interprocess block

Add a new generation block in launch/trail/config/usv.pb.cfg.py

And finally add the new binaries to the  ${\tt usv.launch}$  file:

```
# launch/trail/usv.launch
goby3_course_usv_health_monitor <(config/usv.pb.cfg.py goby3_course_usv_health_monitor)
goby_coroner <(config/usv.pb.cfg.py goby_coroner)</pre>
```

Also, for anything you want to monitor glog VERBOSE output on, add a -v to the launch line:

```
# launch/trail/topside.launch
goby3_course_topside_manager <(config/topside.pb.cfg.py goby3_course_topside_manager) -v
```

(optional) and as, above, if the sim is too fast, slow it down:

```
# launch/trail/config/common/sim.py
warp=1
```

#### Run

Run using '-r' so we can see the status of all the applications:

```
cd launch/trail
# instead of ./all.launch which runs "goby_launch -s -P -k30 -ptrail -d500"
goby_launch -r -P -k30 -ptrail -d500 all.launch
```

Check out our health report by attaching to topside's manager screen

```
screen -r topside.goby3_course_topside_manager
```

Try manually terminating a process on the USV to ensure that your health reports as "FAILED":

```
goby_terminate --target_name "goby3_course_usv_manager" --interprocess 'platform: "usv"'
# or a bit more bluntly
killall goby3_course_usv_manager
```

#### **Bonus Task**

We really don't care that much about the HealthStatus message when things are "GOOD", but we would like to know when they aren't.

Let's split our HealthStatus publication into two groups:

```
// GOOD
constexpr goby::middleware::Group health_status_good {"goby3_course::health_status_good", 1};
// FAILED
constexpr goby::middleware::Group health_status_failed {"goby3_course::health_status_failed", 2};
// we could add similar groups for degraded, failing, etc.
```

Using the set\_group\_func callback to Publisher on the publication side, set the state field of HealthStatus based on the published group.

Then, publish GOOD messages to health\_status\_good with a low base priority value (e.g. 0.5) and those that are FAILED to health\_status\_failed with a high base priority value (e.g. 10). Remember these priority values are relative to other messages, and the only other message we're currently publishing from the USV is the NavigationReport at the default priority value of 1.

Update the topside to subscribe to both groups. You don't need to set the priority values again here at the subscriber (but if you do they will be averaged with the publisher's values, leading to the same result).

Currently the topside/USV link has more throughput that we're sending so you won't really see a difference. To notice the priority change, let's crank down the throughput by changing the MAC cycle:

```
# launch/trail/config/templates/_link_satellite.pb.cfg.in
# ...
mac {
    type: MAC_FIXED_DECENTRALIZED
    slot { src: 1 slot_seconds: 10 max_frame_bytes: 26 }
    slot { src: 2 slot_seconds: 10 max_frame_bytes: 26 }
}
```

Now we're only sending 26 bytes (two NavigationReports) every 10 seconds, so we should see our health\_status\_good messages take priority behind the NavigationReports but then health\_status\_failed should come through right away.

## Wrap up

Good work - now we are set up to command our USV to perform another autonomy mission (which we'll look at during the lecture tomorrow), and we can report (at a basic level) the health of the vehicle.

From here, hopefully you can see a path forward to building a full system and filling out all the details that are required to function in a real deployment.

# Solutions (Toby)

My solutions are pushed to the post-homework2 branch of goby3-course. Please reference the code together with this text.

## Assignment 1:

Created src/lib/messages/command\_dccl.proto, and added to CMakeLists.txt:

```
# src/lib/messages/CMakeLists.txt
protobuf_generate_cpp(
# ...
goby3-course/messages/command_dccl.proto
)
```

#### Added to groups.h:

Copied pattern/single\_thread to src/bin/command\_test and renamed application Class to commandTest. Added add\_subdirectory(command\_test) to parent CMakeLists.txt. Named binary goby3 course\_command\_test in command\_test/CMakeLists.txt.

Created publication of USVCommand in goby3\_course\_command\_test's app.cpp using the intervehicle1/publisher as a starting point.

Then, added a subscription to command message in goby3\_course\_usv\_manager setting:

- ack\_required: true (we want our commands to resend until ack'd or until their expire)
- max\_queue: 1

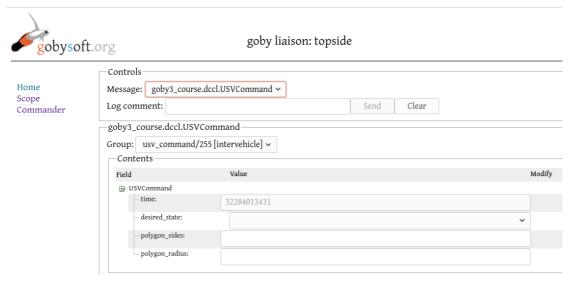
Ran the example and saw that we the command gets to the USV (see the red commands window):

```
Ungrouped messages
10:19:51 | Adding FlexOstream group: auv_nav
             Adding FlexOstream group: usv_nav
10:19:51
10:19:51 | Adding FlexOstream group:
             Received DCCL nav: vehicle: 8 time: 1614200126 x: 351.5 y: 484 z: -70 speed_over_ground: 1.5 heading: 42 type: AUV
10:28:50
             Received DCCL nav: vehicle: 9 time: 1614200127 x: 368.9 y: 519.7 z: -80 speed_over_ground: 1.5 heading: 45 type: AUV
             Received DCCL nav: vehicle: 10 time: 1614200128 x: 388.9 y: 556.1 z: -90 speed_over_ground: 1.5 heading: 45 type: AUV
pth: 0 } local_fix { x: 458.7167633693316 y: 566.78959774412215 z: -0 } pose { heading: 51 } speed { over ground: 1.5 }
10:29:49 | ^^ Converts to DCCL nav: vehicle: 1 time: 1614200129.4557319 x: 458.7167633693316 y: 566.78959774412215 z: -0 speed_over
 1.5 heading: 51 type: USV
10:27:02
             Received USVCommand: time: 1614162420 desired state: WAYPOINTS
             Received USVCommand: time: 1614162480 desired_state: WAYPOINTS
             Received USVCommand: time: 1614162540 desired_state: WAYPOINTS
10:29:02
help: [+]/[-]: expand/contract window | [w][a][s][d]: move window | spacebar: toggle minimize | [r]: reset | [CTRL][A]: select all
[3]...[n] select window n | [SHIFT][[n] select multiple | [enter] pause and scroll | [c]/[C] combine/uncombine selected windows
```

## Bonus Task

Now, I copied the suggested configuration for goby liaison, and reran ./all.launch.

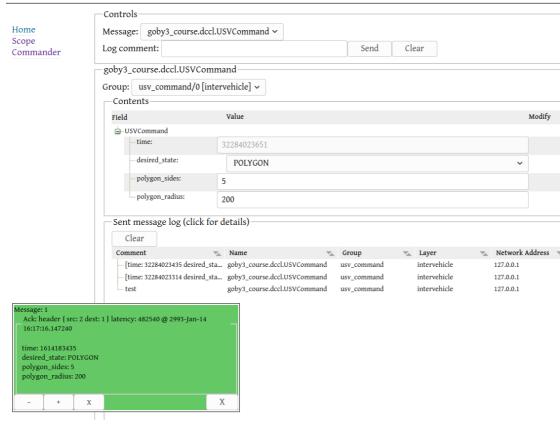
When I open liaison (http://localhost:50000/? =/commander), I can now fill in the message:



Once I fill out the message and send it, I get the acknowledgment of the sent message:

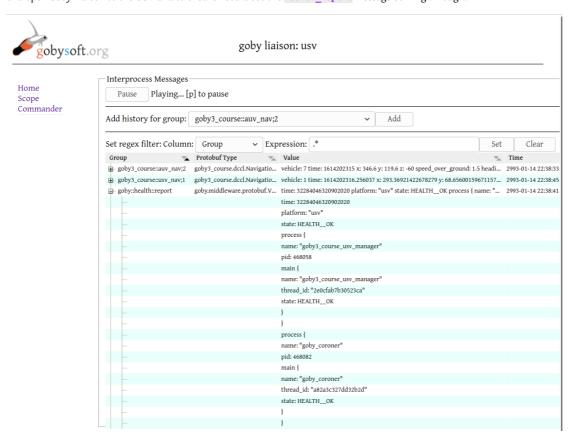


## goby liaison: topside



# Assignment 2:

I did this a bit out of order. First I added the configurations for goby\_coroner to launch/trail/config/templates and usv.launch. Then I ran ./all.launch and open Goby Liaison to the USV and to ensure I could see the health report message coming through:



After I got that working I created the <code>goby3\_course\_usv\_health\_monitor</code> code, which I put in <code>src/bin/health\_monitor</code>. Using the <code>intervehicle1</code> code as a starting point, I subscribed to the <code>health\_report</code> from <code>goby\_coroner</code> and determined that if <code>HEALTH\_OK</code>, we'd say <code>HealthStatus::good</code>. In a real system we'd want to aggregate data from more sources than just <code>goby\_coroner</code> before making that determination, but for this course, that will do.

From here, I published the HealthStatus message on the intervehicle layer. I added a subscribe usv\_health() method to the topside Manager to subscribe to this message. I also added some glog "groups" (streams) to more clearly see what is going on.

After terminated one of the processes as suggested, we can see that my health report switches to failed: