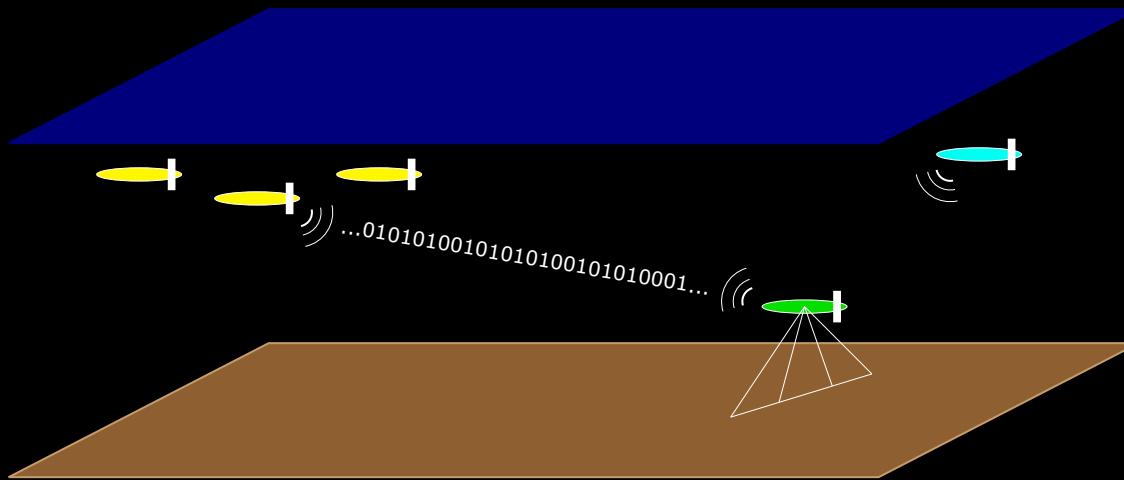


# Goby3 Course

## Day 1: Overview



Toby Schneider

*GobySoft, LLC*

*Mashpee, MA, USA*



# What is Goby?

---

- A C++ **library** with a collection of marine robotics related code?
- **Drivers** for acoustic modems and related “slow links” (e.g. satellites)?
- A publish/subscribe **middleware** (framework)?
- An **open source project** that is freely available under the terms of the LGPL?
- Any of more than 2,200 species of small, carnivorous **fishes**?



*Goby is all of these things.*

*The goal is that you can use what you need, and no more.*

# What is Goby?

---

- A C++ library with modular components for:
  - acoustic communications
  - utilities for marine software (AIS, seawater calculations, NMEA-0183, ...)
  - time functions, supporting faster-than-realtime sims
- A “nested comms” publish/subscribe middleware:
  - a common **transport** interface for thread-to-thread, process-to-process, and vehicle-to-vehicle comms
  - **application** base classes for quickly writing processes
  - extensible **marshalling** schemes, supporting Protocol Buffers, MAVLink, DCCL, and easily many more.
  - a growing list of useful applications (GUIs, GPSD, logger, ...) and threads (I/O: serial, UDP, TCP, CAN, ...)

# Why create Goby?

---

Marine robotics is small field of engineering, but with some relatively unique technical problems:

- communications (low throughput / high latency)
- harsh environments (expensive deployments)
- wide array of sensors, many specific to seawater

My hope:

Goby is a **beginning** to some solutions.

More importantly, it is **open source** so that we (as a **community** of oceanographic engineers) can build on it, and improve our use of software **best practices** along the way.

# History

---

MIT / LAMSS (2007-2012) [lamss.mit.edu]:

- pGeneralCodec MOOS Application -> DCCL v1 (XML)
- pAcommsHandler MOOS Application -> Queue / ModemDriver
- Refactor pAcommsHandler into the beginning of Goby (v1)
- DCCL rewrite (v2): XML to Google Protocol Buffers
- Goby 2 (primarily to support DCCL v2)

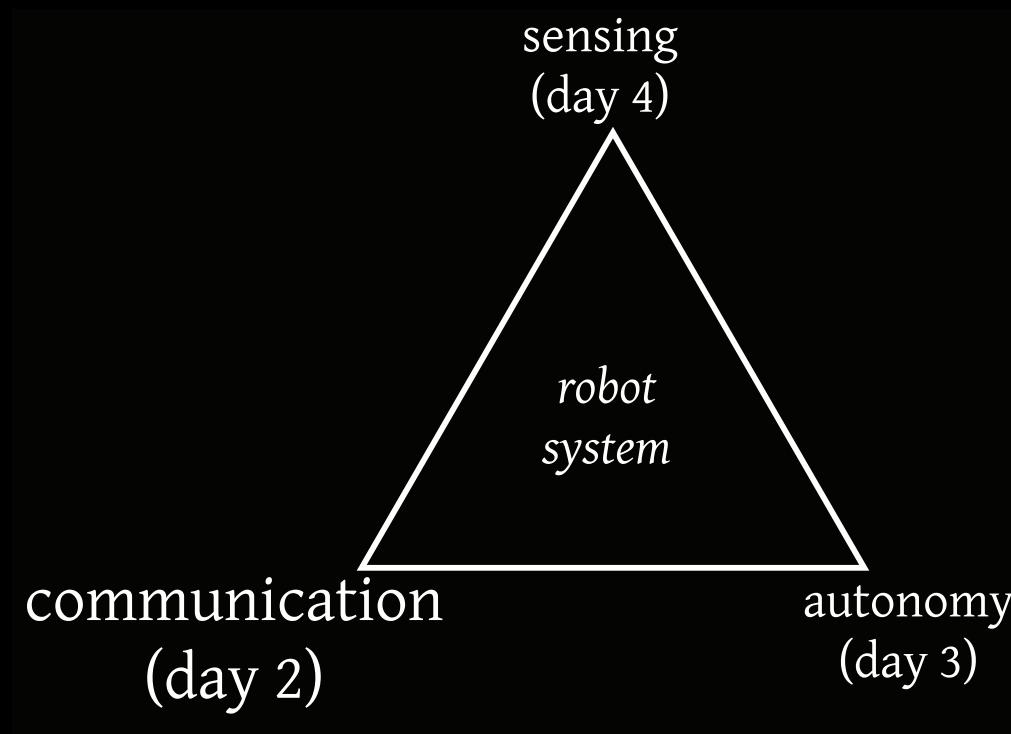
GobySoft (2013-present) [gobysoft.org]:

- DCCL standalone (v3) library, Goby 2.1
- Goby 3 (nested middleware)

# Robots (and syllabus)

In many systems, this triad represents tradeoffs:

- More communications = less need for autonomy (UAVs)
- Better autonomy = better data from cheap sensors (Adaptive sampling)
- Better sensors = less need for outside data (Manned subs)



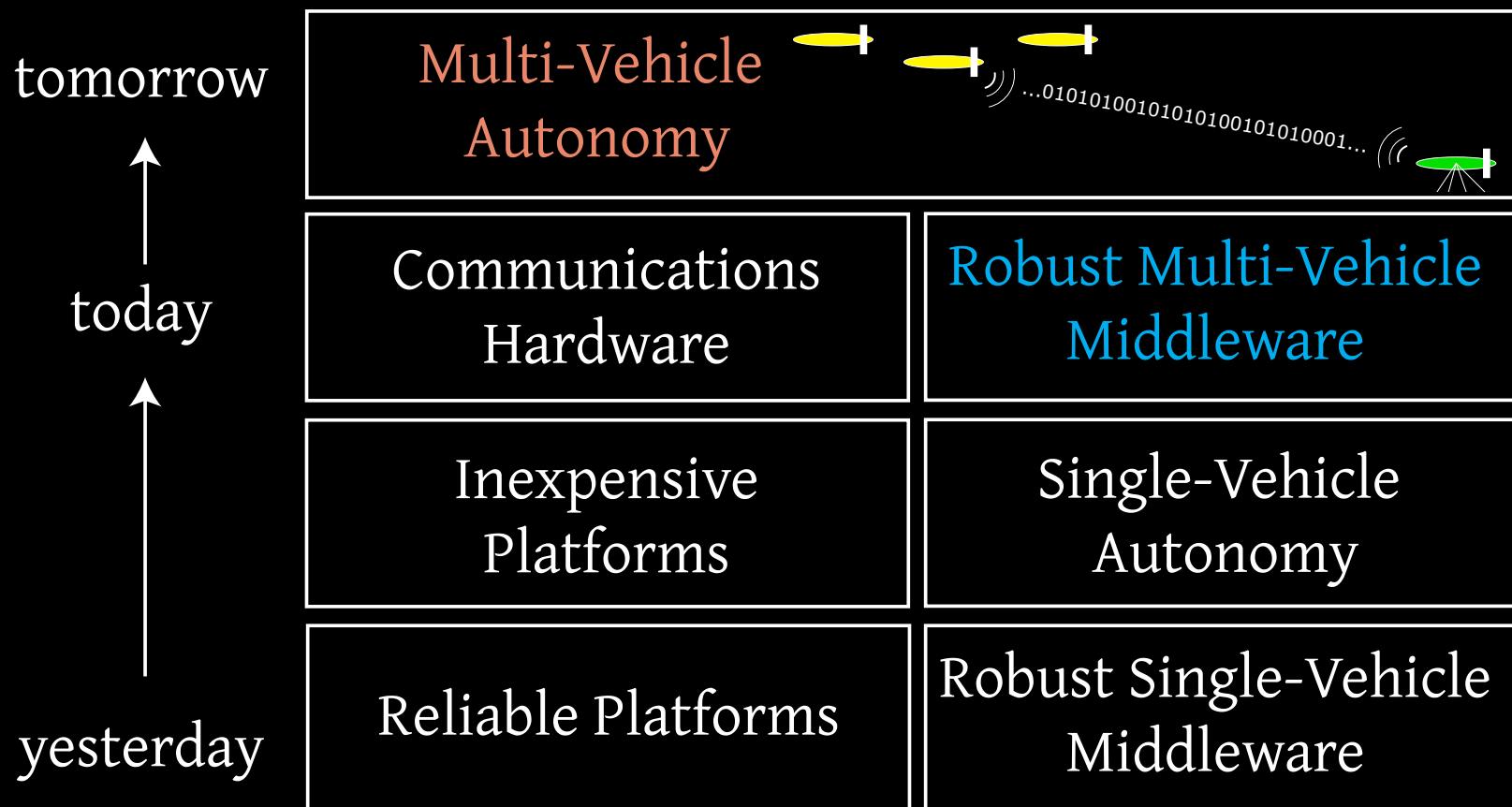
# (More) Syllabus

---

- Day 1 (Monday, March 8): Overview
  - Lecture: 9 am-11 am EST (UTC-5).
  - Homework assignment
- Day 2 (Tuesday, March 9): Technical I: Communications
  - Lecture: 9 am-11 am EST.
  - Homework assignment.
- Day 3 (Wednesday, March 10): Technical II: Autonomy
  - Lecture: 9 am-11 am EST.
  - Homework assignment.
- Day 4 (Thursday, March 11): Technical III: Sensors
  - Lecture: 9 am-11 am EST.
  - Homework assignment.

# The Road to Swarms

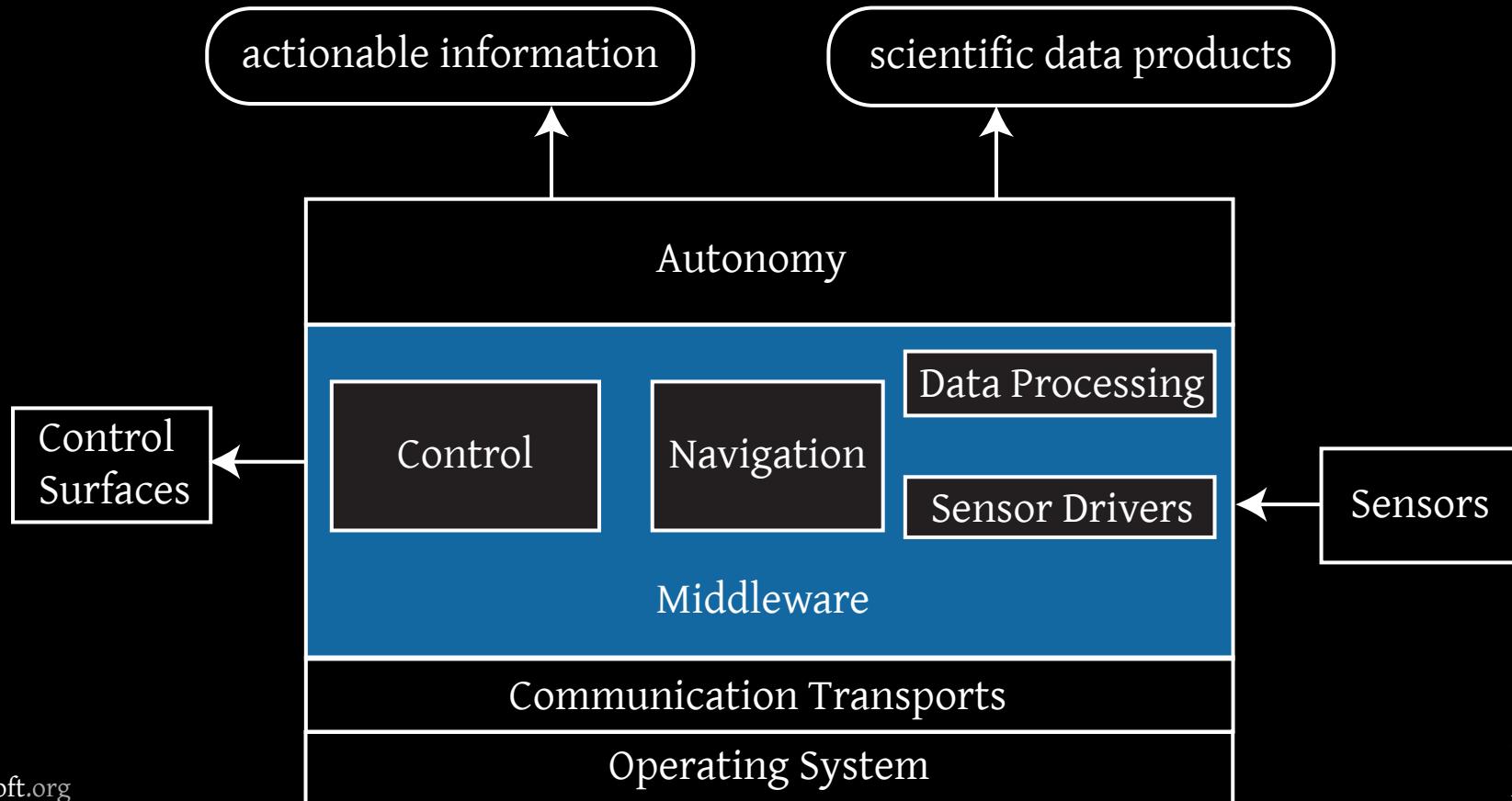
Multi-vehicle autonomy needs to be supported by multi-vehicle middleware.



# What is middleware?

**Primary** role: **asynchronous communication mechanism**, typically publish/subscribe.

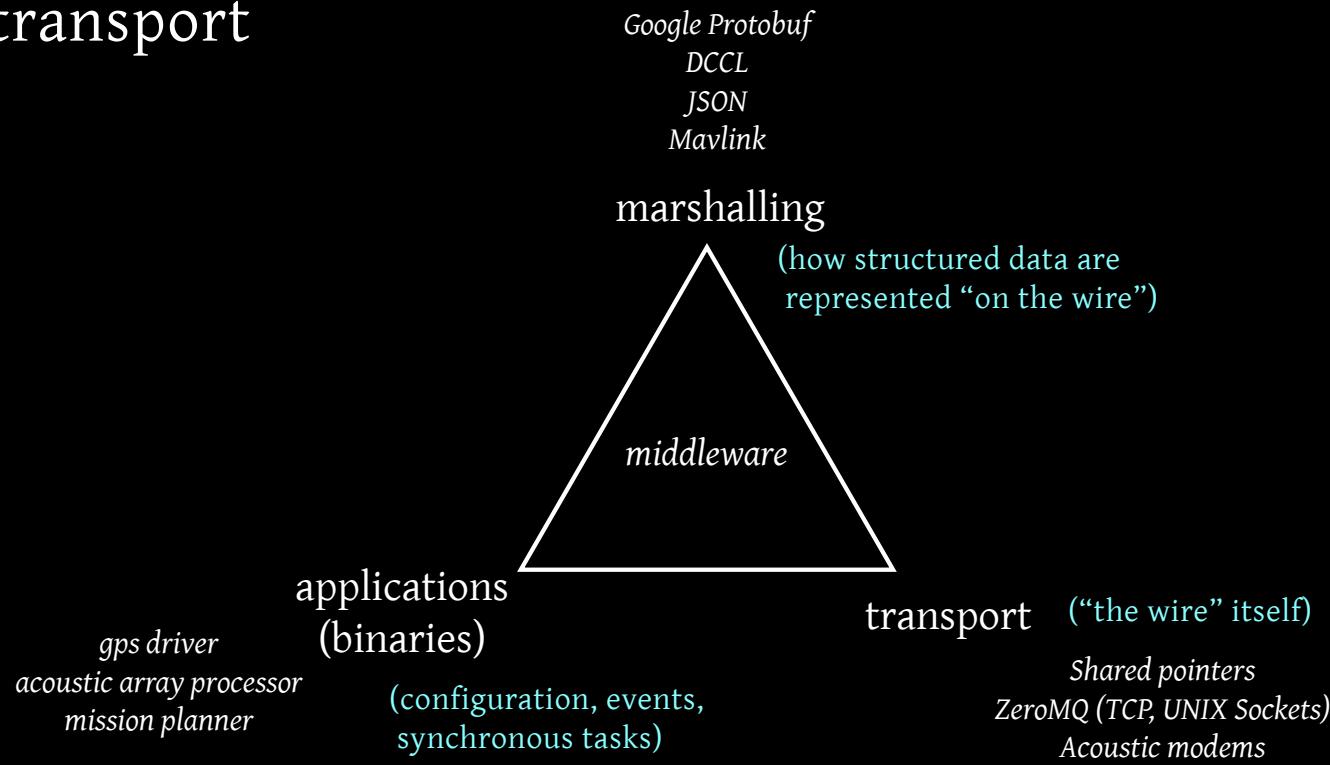
**Secondary** roles: tools for launching processes, logging, management, and monitoring.



# Goby3 (Meta?) Middleware

This triad represents **composition**:

- marshalling + transport + application support = Goby3 middleware
- “meta-middleware”: bring your own marshalling + transport



# Existing middleware used in AUVs



MOOS - Mission Oriented Operating Suite

- *Transport:* TCP with central broker
- *Marshalling:* CMOOSMsg: string, double or binary
- *Application support:* CMOOSApp



ROS - Robot Operating System

- *Transport:* TCP with central broker;  
shared pointer interthread comms
- *Marshalling:* ROSMsg: user-defined “structs”
- *Application support:* ROS Node (loose specification)

LCM - Lightweight Communications and Marshalling

- *Transport:* UDP multicast interprocess comms,  
no central broker
- *Marshalling:* LCMTYPE: conceptually similar to ROSMsg
- *Application support:* None

None are specifically designed for marine vehicles.

All mandate specific transport & marshalling scheme.

# Goby3 Innovations

---

## Nested communications

- Communications approach varies based on characteristic throughput and latency scale.

Neutral about choice of transport & marshalling scheme

- Allows easier inclusion of external innovations on networking and data presentation.
- New or project-specific marshalling languages can be used without middleware modification
- Interoperability with existing middleware

# Nested Communications

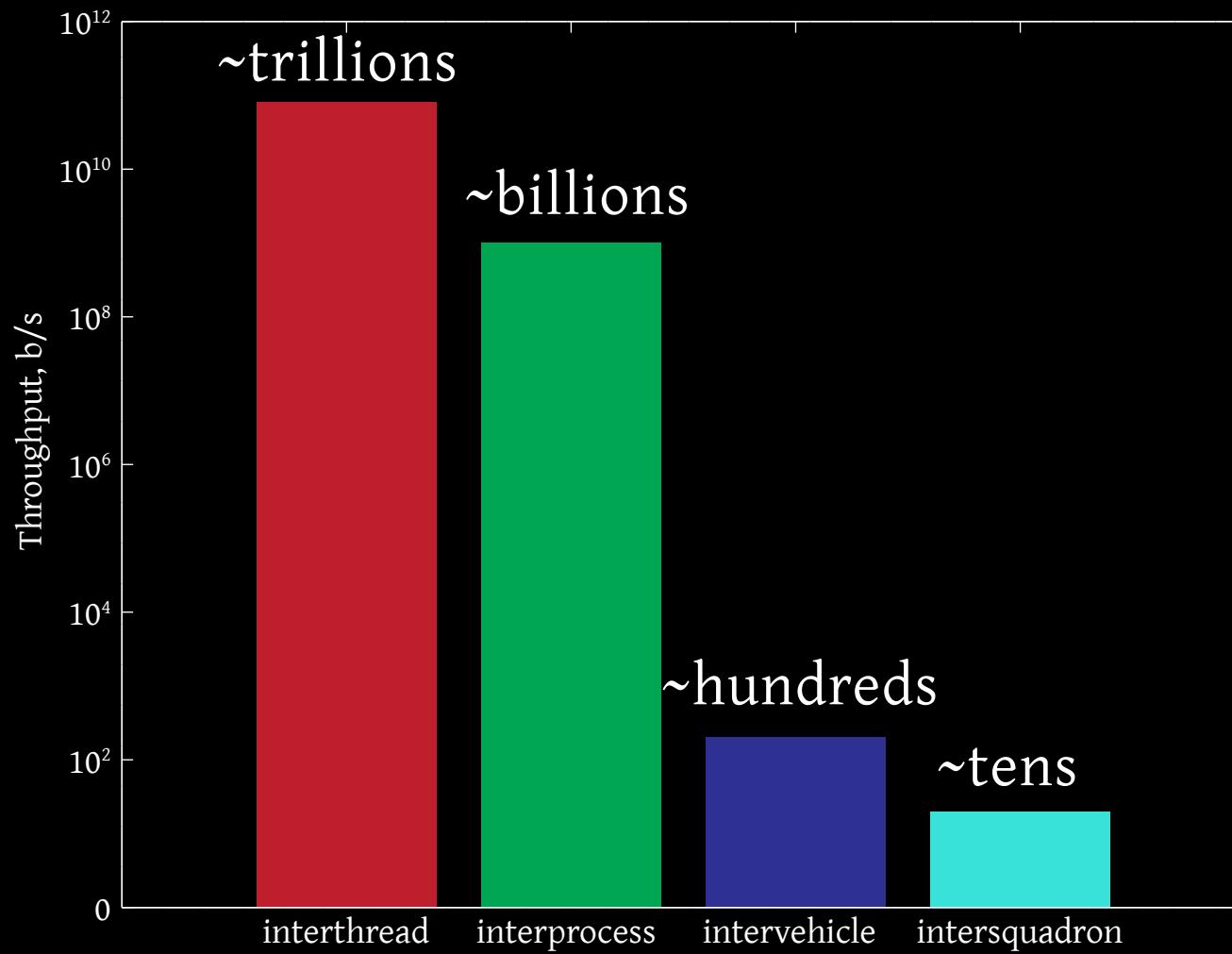
---

Core idea: Networking approaches are fundamentally different for different characteristic latency and throughput scales.

- *Non-marine systems*: interprocess comms **within** and **amongst** vehicles are on the **same scale**.
- *Marine systems*: **interprocess** comms are an entirely **different scale** than **intervehicle** comms. Why?
  - Acoustic communications (fundamental physical limitations)
  - Sparse deployments (require satellite comms, long range radio).

# Nested Communications

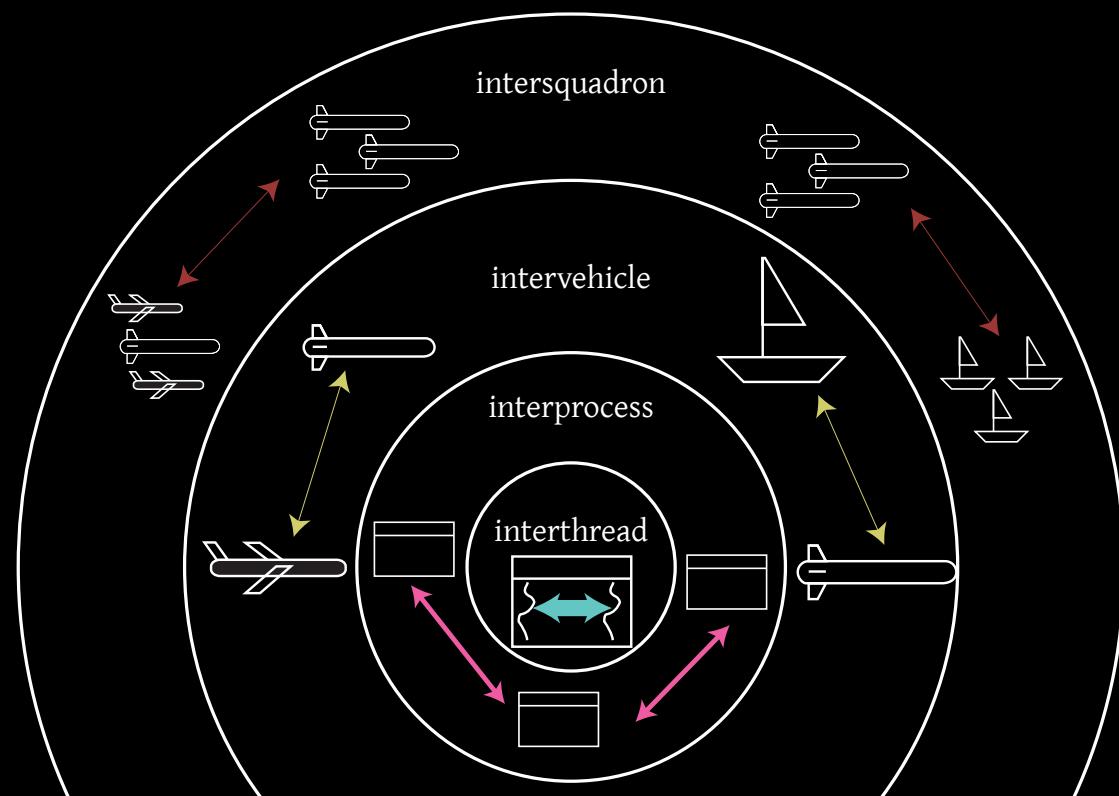
Characteristic throughput scales for AUVs:



# Nested Communications

Each conceptual layer requires different transport and marshalling schemes.

Goby3 allows this and provides a **common interface** paradigm for publish/subscribe and cross layer forwarding.



# Transport(er) Classes

---

Nested comms transporter classes are in two categories (or concepts)

- **Portal** concept: connects one nested layer to the next.  
Exactly one portal per node per layer.

interthread: node == thread

interprocess: node == process

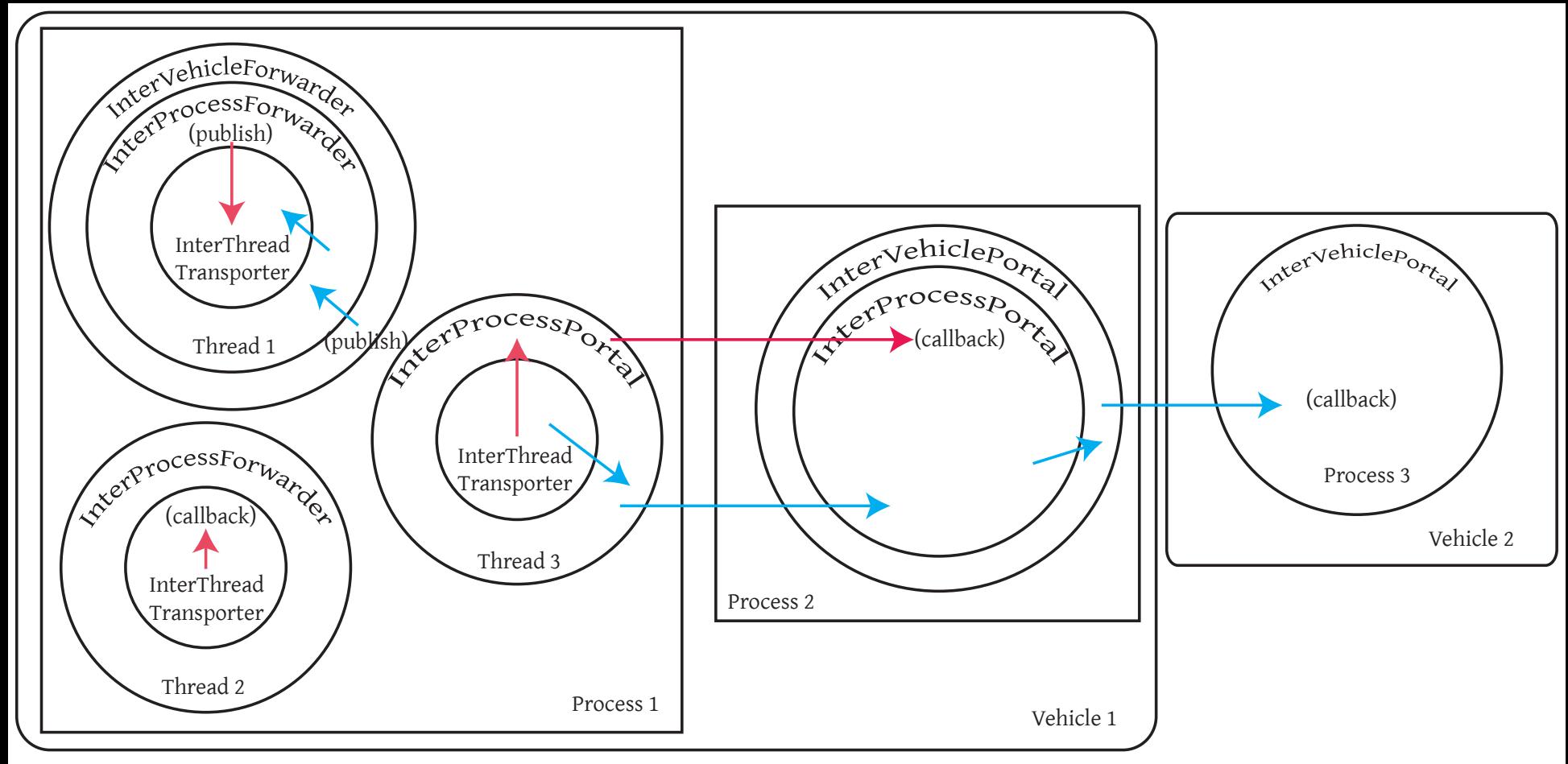
intervehicle: node == vehicle

**Forwarder** concept: forwards data from inner layer nodes to the layer's portal. Any number per node.

For example: **InterProcessPortal** - connects the threads within a process to the rest of the processes.

**InterProcessForwarder** - forwards data from other threads to the **InterProcessPortal** (and thus to other processes).

# Portal/Forwarder Example



- blue is published by the InterVehicleForwarder on Vehicle 1 / Process 1 / Thread 1, and subscribed by the InterVehiclePortal on Vehicle 2 / Process 3
- red is published by the InterProcessForwarder on Vehicle 1 / Process 1 / Thread 1, and is subscribed by Vehicle 1 / Process 1 / Thread 2 & Vehicle 1 / Process 2.

# Reference Implementation

---

Goby3 provides a framework for defining nested layers, and attaching transport layers and marshalling schemes.

For immediate use, it also provides a three-level open source reference implementation in C++14.

- **Interthread:** C++ shared pointers: all types supported
- **Interprocess:** ZeroMQ (TCP/UNIX Sockets): Google Protocol Buffers and DCCL types, extensible to any serializable type
- **Intervehicle:** Goby-Acomms slow link transports and DCCL types

Fully event driven; based on condition variable signaling.

# Marshalling Schemes

---

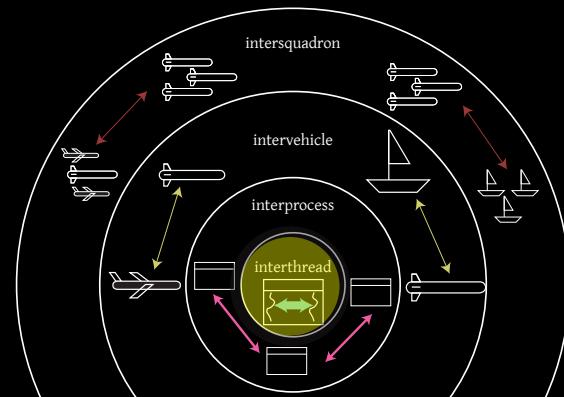
Marshalling is “the process of transforming the memory representation of an object to a data format suitable for storage or transmission” (Wikipedia)

- More concretely, in C++,  
struct/class -> array<byte> -> struct/class
- Many support multiple programming languages
- Example approaches include Google Protocol Buffers, MAVLINK, DCCL, JSON, MessagePack, ASN.1:
  - these are “marshalling schemes” in Goby3 (or just “schemes” for short)

# Transport: Interthread

Zero-copy publish/subscribe implementation using `std::shared_ptr`. No constraints on allowable message types representable in C++.

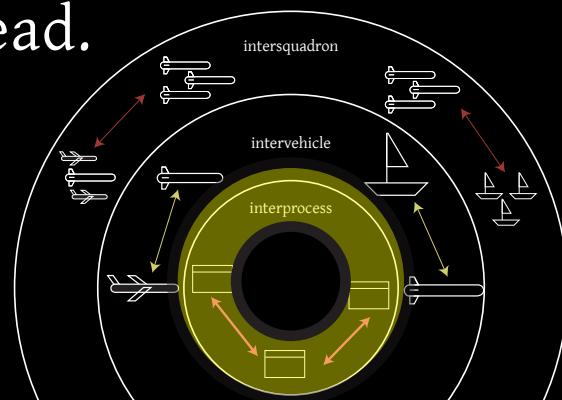
- publisher thread ---`shared_ptr`---> subscriber thread(s)
- Allows users unfamiliar with thread memory management to easily write multithreaded code.
- Allows multi-process code to be changed to multi-thread when needed to improve performance without changing message passing paradigm.



# Transport: Interprocess

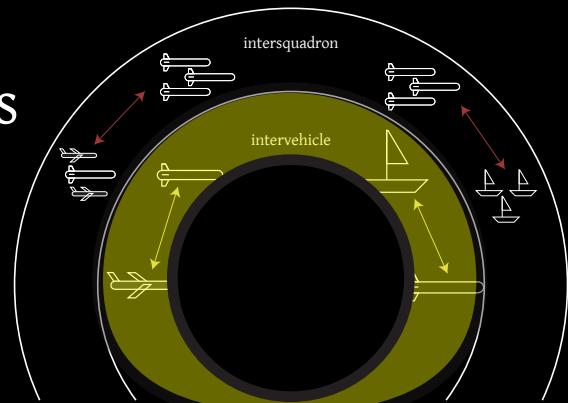
Reference implementation uses **ZeroMQ**. Types supported are **DCCL** and **Google Protocol Buffers**, but trivially extensible to any **Serializable** type (by implementing functions for that marshalling scheme).

- Broker (gobyd) handles multiple publishers and subscribers over TCP or Unix Sockets.
- Subscription forwarding: publishers that have no subscribers use no bandwidth
- The inner layer (if any) is typically InterThread.
- MOOS, ROS comms is roughly analogous to this layer.



# Transport: Intervehicle

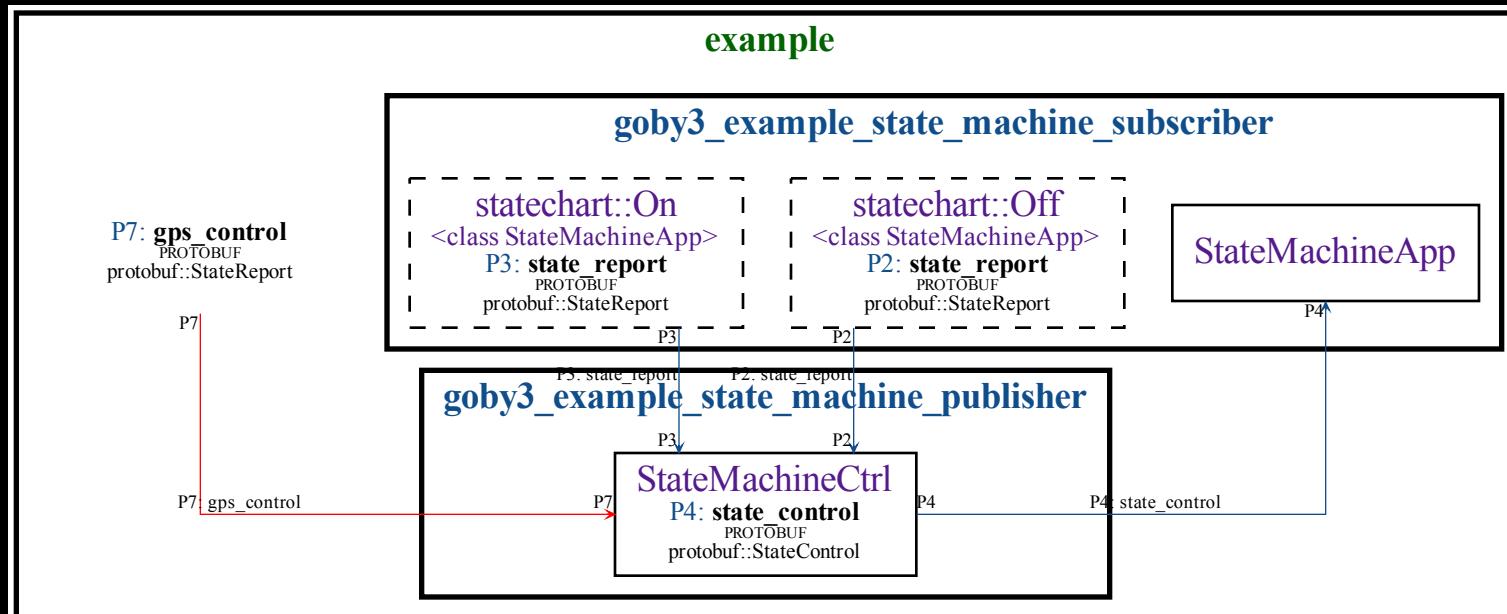
- **ModemDriver**: Common interface to “slow link” devices with implementations for various acoustic, satellite, etc. modems (MicroModem, Benthos, Iridium, ...).
- **DCCL** ([libdccl.org](http://libdccl.org)): static unit-safe lossless marshalling that allows for physics-based bounds to create arbitrary sized fields (e.g. 6-bit int for temperature field for 0° to 40° C)
- Dynamic Buffer: deadline sensitive priority queuing
- Subscription forwarding: published data does not get sent if no subscriber exists.
- The inner layer is typically InterProcess.



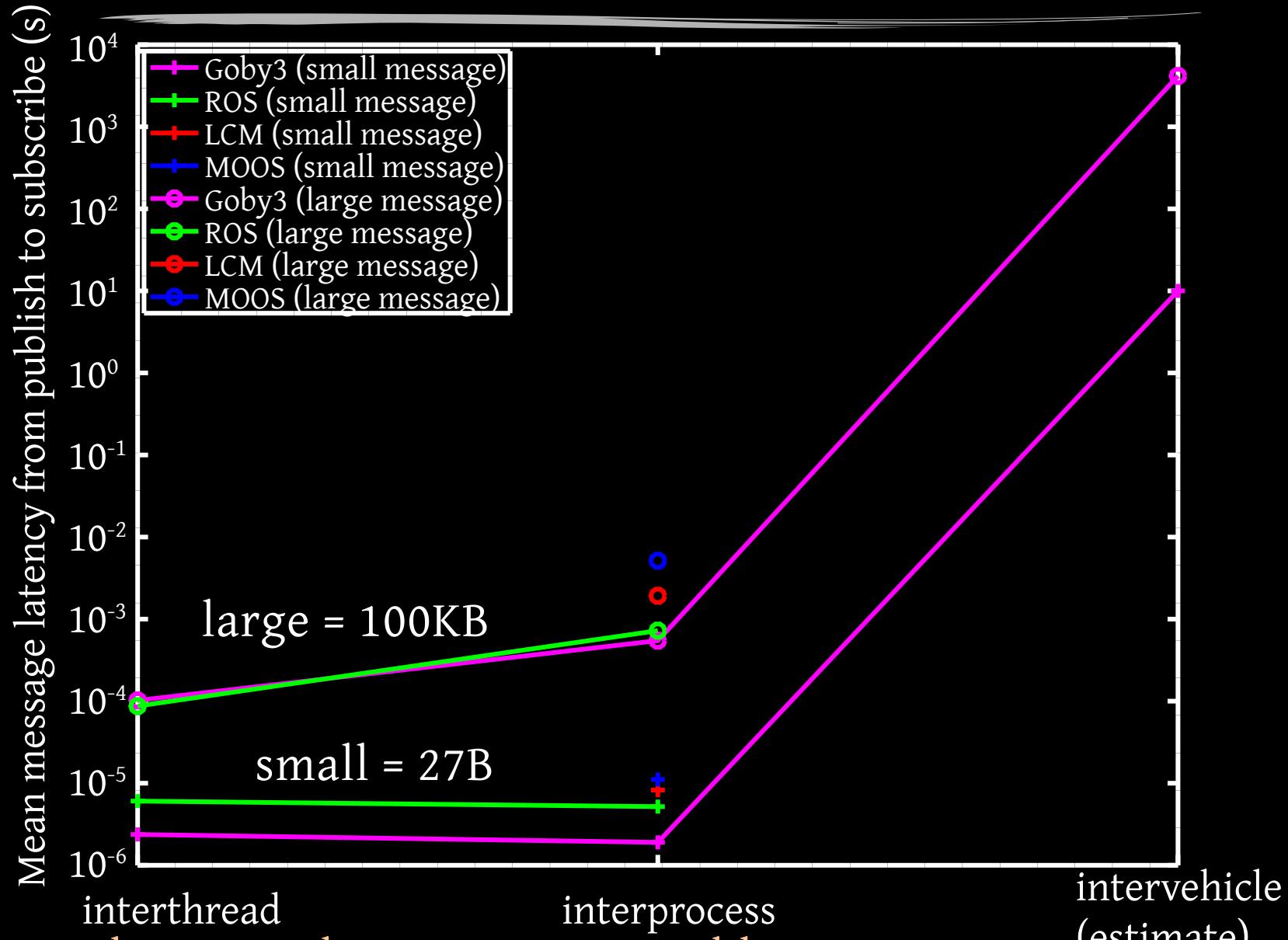
# Static Analysis

Emphasis on static type safety via template “groups” (goby::middleware::Group). (Analogous to ROS topics, LCM channels, MOOS variables).

By using `constexpr`, we can generate static pub/sub graphs (using `goby_clang_tool`):



# Performance Results



Goby3 gives better or comparable performance to existing middlewares

# Application: Single Thread

Rapid prototyping base class: Inspired by `CMOOSApp`, but with significant improvements.

- Automatic configuration and command-line parsing, and syntax checking into Google Protobuf message passed as a template parameter.
- Single virtual method: `loop()`, analog of `CMOOSApp::Iterate()`.
- No other virtuals (Constructor handles the equivalent of `OnStartup` / `OnConnectToServer`).
- Contains an `InterProcessPortal`, with no inner level.

# Example (SingleThreadApplication)

```
#include "goby/middleware/single-thread-application.h"
#include "messages/nav.pb.h"
#include "messages/groups.h"
#include "config.pb.h"

class BasicPublisher :
    public goby::SingleThreadApplication<BasicPublisherConfig>
{
public:
    BasicPublisher() : Base(10 /*hertz*/) {
        // all configuration defined in BasicPublisherConfig is in cfg()
        std::cout << "My configuration int: " << cfg().my_value() << std::endl;
    }

    void loop() override {
        protobuf::NavigationReport nav;
        nav.set_x(95 + std::rand() % 20);
        nav.set_y(195 + std::rand() % 20);
        nav.set_z(-305 + std::rand() % 10);
        transporter().publish<groups::nav>(nav);
    }
};

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

```
import "goby/common/protobuf/app3.proto";
import "goby/middleware/protobuf/interprocess_config.proto";

message BasicPublisherConfig {
    optional goby.protobuf.App3Config app = 1;
    optional goby.protobuf.InterProcessPortalConfig interprocess = 2;
    optional int32 my_value = 3;
```

**publisher.cpp**

```
#include "goby/middleware/serialize_parse.h"
namespace groups {
    constexpr goby::Group nav{"navigation"};
}
```

**groups.h**

package protobuf;  
**nav.proto**

```
message NavigationReport {
    required double x = 1;
    required double y = 2;
    required double z = 3;
    enum VehicleClass { AUV = 1; USV = 2; SHIP = 3; }
    optional VehicleClass veh_class = 4;
    optional bool battery_ok = 5;
}
```

```
#include "goby/middleware/single-thread-application.h"
#include "messages/nav.pb.h"
```

**subscriber.cpp**

```
#include "messages/groups.h"
#include "config.pb.h"

class BasicSubscriber :
    public goby::SingleThreadApplication<BasicSubscriberConfig>
{
public:
    BasicSubscriber() {
        using protobuf::NavigationReport;
        auto nav_callback = [this] (const NavigationReport& nav) {
            std::cout << "Rx: " << nav.DebugString() << std::flush;
        };
        transporter().subscribe<groups::nav, NavigationReport>(nav_callback);
    }
};

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

**config.proto**

# Application: Multi Thread

Extends the SingleThreadApplication to multiple threads.

- Each thread is a subclass of a similar interface to the SingleThreadApplication.
- This approach avoids accidental thread memory contention if all data are stored at the class level (no globals).
- Threads can be spawned and joined as necessary. Joining happens as soon as loop() returns.

# Repository Structure

---

Core:

- libgoby.so: acomms, middleware, time, util
- middleware tools: goby\_clang\_tool, goby\_log\_tool

ZeroMQ:

- libgoby\_zeromq.so
- zeromq apps: gobyd, goby\_logger, etc.

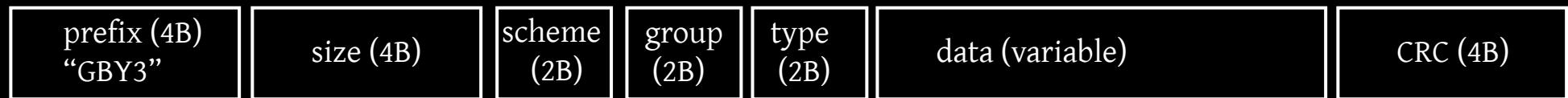
MOOS:

- libgoby\_moos.so
- pAcommsHandler

(Explore *goby3-repository-layout.pdf*)

# Binaries: Logger

goby\_logger writes a flat binary log file, with each message protected by prefix word and CRC32

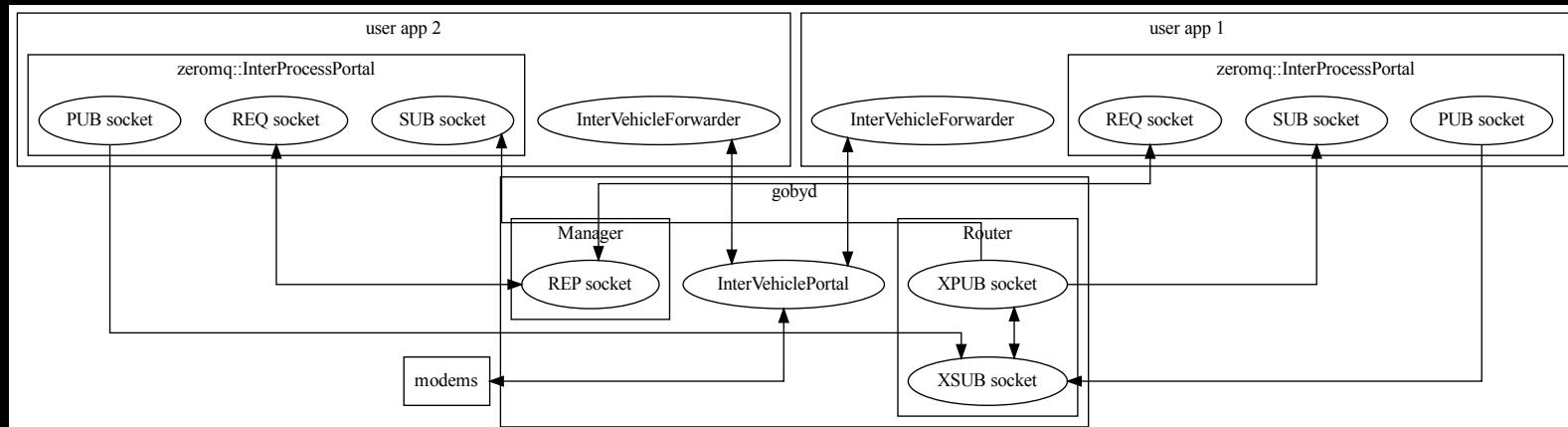


- If corruption occurs, remaining messages can be recovered
- Plugin API for additional storage of metadata for specific schemes (e.g. Protobuf descriptors)
- Post-processing can be handled by scheme-specific parsers. Includes parser in goby\_log\_tool for DCCL/Protobuf -> HDF5 and text. (HDF5 can be directly read in MATLAB/Octave, Python, C++, Julia, etc.)

# Binaries: gobyd

Brokers pub/sub communications for ZeroMQ interprocess reference implementation.

- Manager:
  - ZeroMQ REP socket: Accepts client (zeromq::InterProcessPortal) connections and provides (dynamic) pub/sub ports
- Router:
  - ZeroMQ XPUB/XSUB proxy
- InterVehiclePortal
  - Manages connections to various modems and InterVehicleForwarder(s)



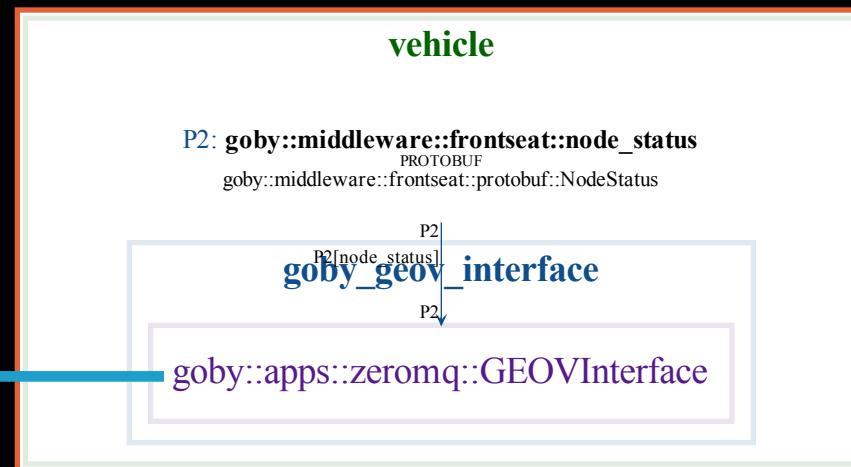
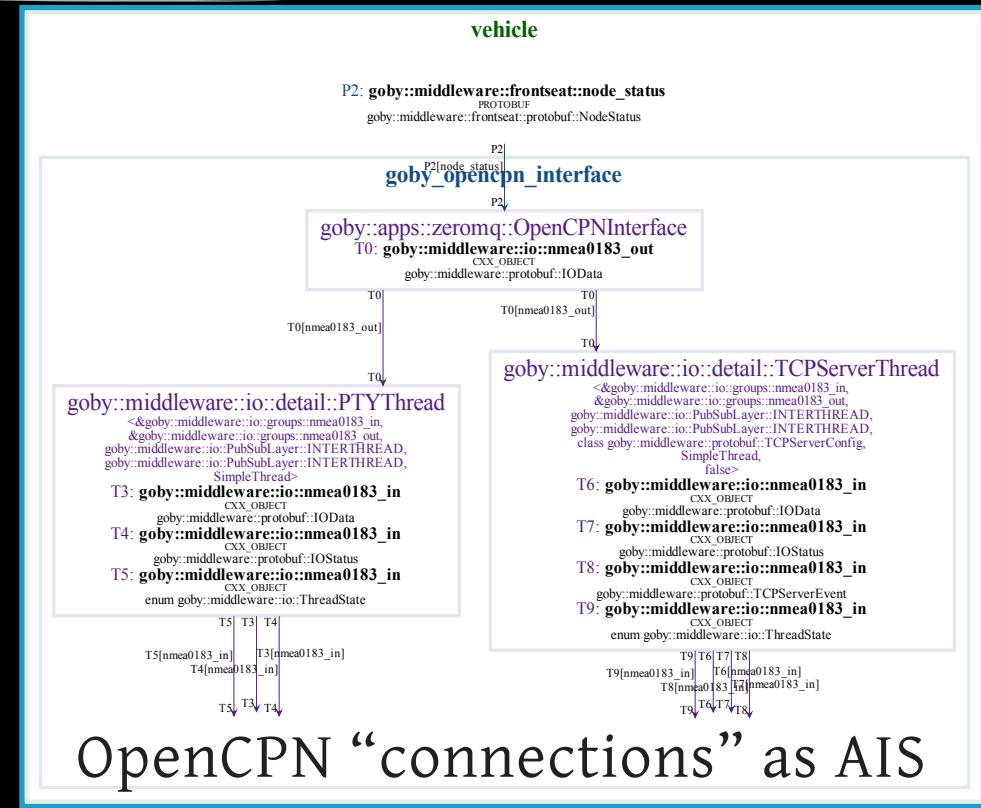
# Binaries: Viewer Interfaces

## goby\_opencpn\_interface

- Provides position data as a fake AIS message to OpenCPN chart plotter (TCP or (pseudo)serial)

## goby\_geov\_interface

- Inputs data to GEOV (Google Earth interface for Ocean Vehicles): <https://gobysoft.org/geov>



Google Earth  
via MySQL

Goby3 Course:  
Day 1: Overview

# Binaries: goby\_liaison

Extensible server/client UI based on Wt (C++ web toolkit).

- Two built-in tabs:
  - Scope: see interprocess data
  - Commander: send Protobuf via a form

The image displays two side-by-side screenshots of the goby liaison web application interface.

**Left Screenshot (Scope Tab):**

- Header:** File, Edit, View, History, Bookmarks, Tools, Help
- Title:** goby liaison: usv
- Content:**
  - Interprocess Messages:** A table showing a list of messages. One message is highlighted: "goby3\_course:auv\_nav:2 vehicle: 2 time: 1613594320 x: 1120.6 y: 455.7 z: -10 speed\_over\_ground: 1.6 head: 2992".
  - Add history for group:** goby3\_course:auv\_nav:2
  - Set regex filter:** Column: Group Expression: \*

**Right Screenshot (Commander Tab):**

- Header:** File, Edit, View, History, Bookmarks, Tools, Help
- Title:** goby liaison: sea\_dragon
- Content:**
  - Controls:**
    - Message: lamss.icex.protobuf.TransmitSelection
    - Log comment:
    - Send, Clear buttons
  - Group:** icex\_selection\_request [interprocess]
  - Contents:** A table showing field values for a TransmitSelection message. Fields include mode (TRANSMIT\_SELECTION\_MANUAL), beacon (TRANSMIT\_H2), rx\_layer (RECEIVE\_DEEP), and metric (default: OPTIMIZE\_SNR).
  - Sent message log:** A table showing transmitted messages. One entry is shown: "[mode: TRANSMIT\_SELECTION... lamss.icex.protobuf.TransmitSelection icex\_selection\_request interprocess 127.0.0.1 2021-02-04 01:53:28]".

# Binaries: goby\_frontseat\_interface

Implements “backseat” part of “frontseat”/“backseat” abstractions

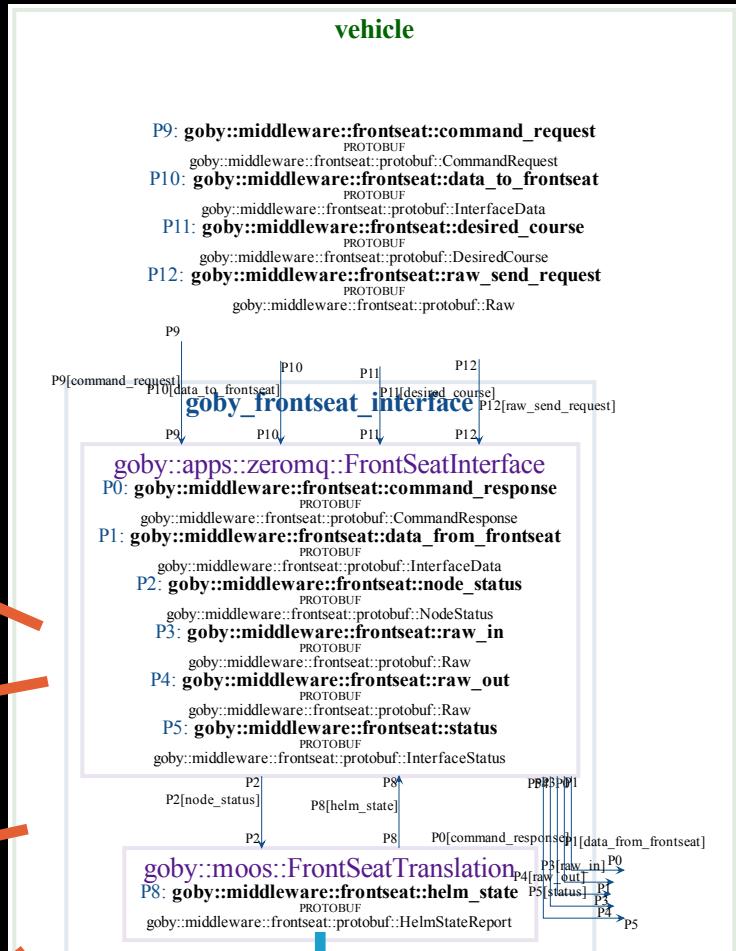
- Backseat = payload autonomy: desired set-points
- Frontseat = vehicle control computer
- New vehicles are C++ .so plugin

plugin: basic simulator

plugin: Bluefin

plugin: Waveglider (SV2)

plugin: Iver3

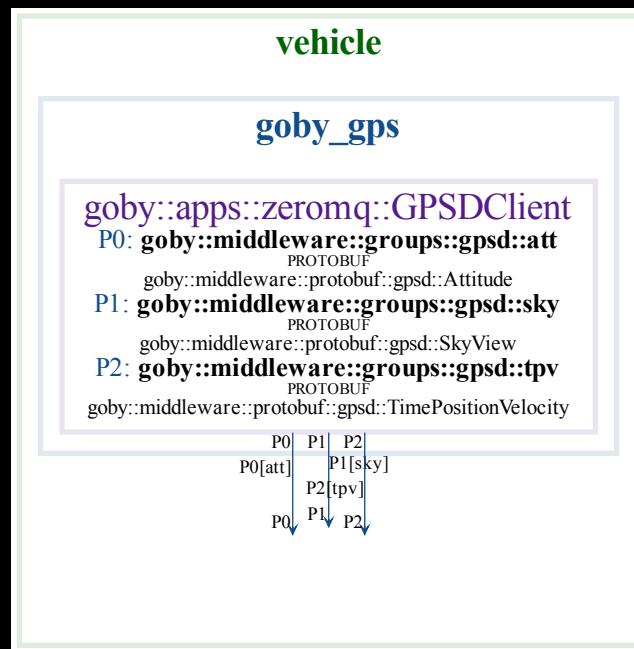


pHelmIvP (MOOS)

# Binaries: goby\_gps (beta)

Connects to gpsd (Linux “interface daemon for GPS receivers”) and publishes the data (altitude, sky view, time/position/velocity) on Goby3 interprocess

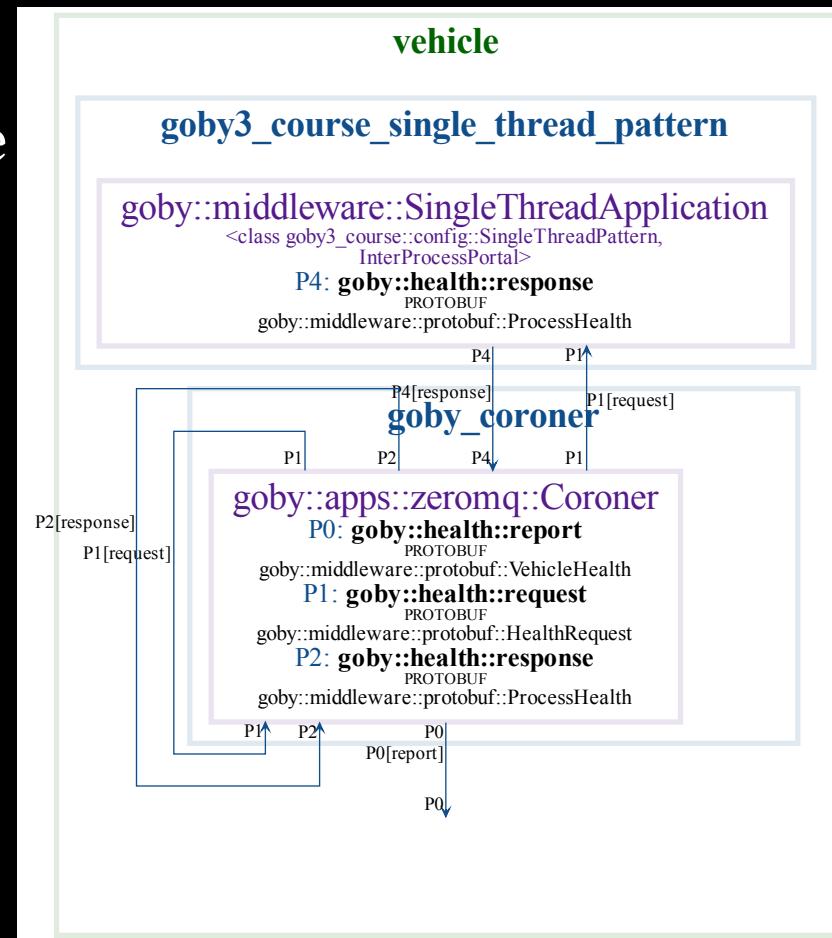
- Can handle filter publications based on device name or publish data from all devices



# Binaries: goby\_coroner

Keep track of health of Goby applications

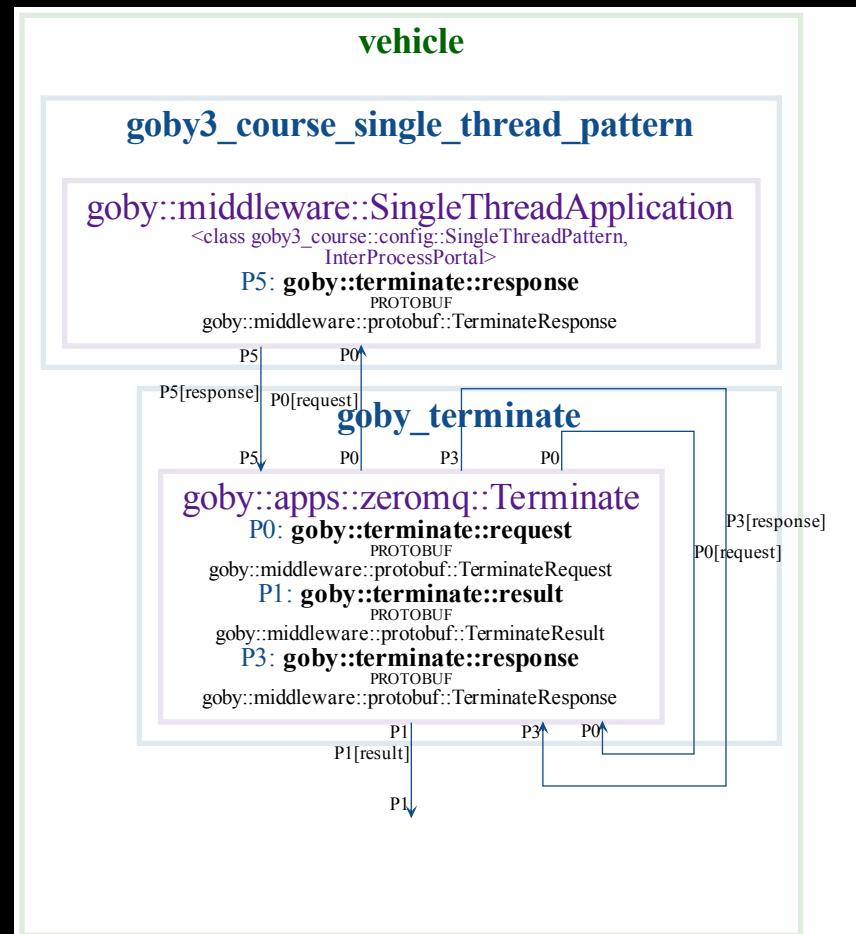
- Request for health given periodically, aggregate as `goby::health::report`
- Applications can override response to give more detail.



# Binaries: goby\_terminate

Clean shutdown tool.

- goby\_terminate requests application to quit (based on PID or name)
- Application responds and quits (cleanly)
- If no response after timeout, goby\_terminate exits with non-zero code.



# Scripts: goby\_launch

Bash script to launch and cleanup applications.

- Designed for multi-process/multi-vehicle simulations (systemd is more suited for actual deployments)
- By default, uses goby\_terminate to cleanup, failing that: SIGINT, failing that: SIGKILL.
- At its simplest, launch file is just a list of applications and command line parameters:

```
#!/usr/bin/env goby_launch

gobyd <(config/topside.pb.cfg.py gobyd) -vvvv -n

goby3_course_topside_manager <(config/topside.pb.cfg.py goby3_
course_topside_manager)

goby_liaison <(config/topside.pb.cfg.py goby_liaison)

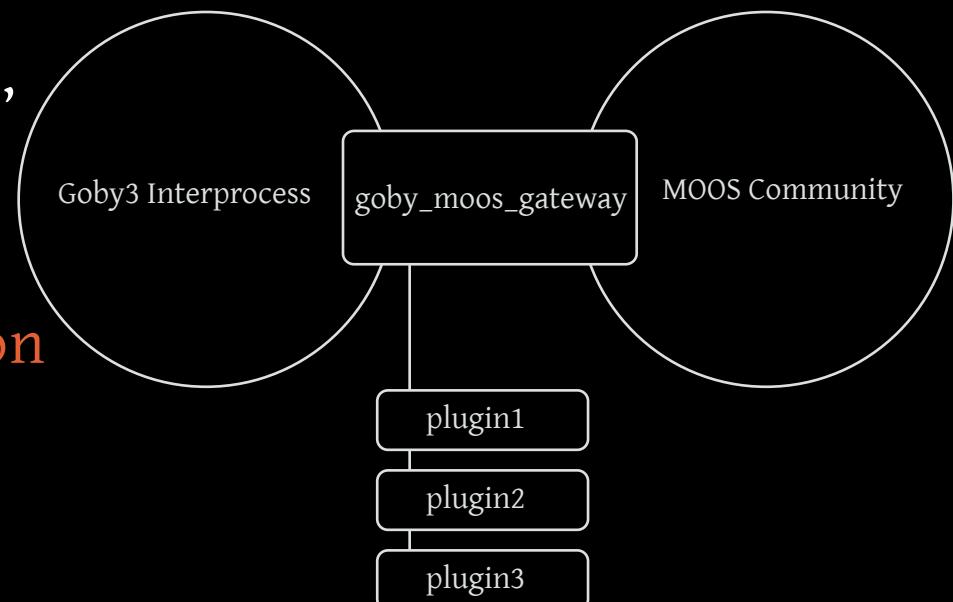
goby_opencpn_interface <(config/topside.pb.cfg.py goby_opencpn_
interface)

goby_geov_interface <(config/topside.pb.cfg.py goby_geov_inter-
face)
```

# Interoperability

Plugin API for Goby3 <-> MOOS translation.  
(future) Goby3 <-> ROS translation will be similar.

- Allows translations to be done in C++, with minimal code; can be as simple or complicated as necessary.
- Example: convert MOOS variables “TEST\_LATITUDE” and “TEST\_LONGITUDE” into Protobuf GPSPosition. Conversely, turn GPSPosition into MOOS variables “GPS\_LATITUDE” and “GPS\_LONGITUDE.”



# Interoperability

```

#include "goby/moos/middleware/moos_plugin_translator.h"
#include "messages/gps.pb.h"
#include "messages/groups.h"

using goby::moos::Translator;
class NavTranslation : public goby::moos::Translator
{
public:
    NavTranslation(const GobyMOOSGatewayConfig& cfg) : Translator(cfg) {
        goby_comms().subscribe<groups::gps_data, GPSPosition>(
            [this](const GPSPosition& pos) {
                moos_comms().Notify("GPS_LATITUDE", pos.latitude());
                moos_comms().Notify("GPS_LONGITUDE", pos.longitude());
            });
        add_moos_trigger("TEST_LATITUDE");
        add_moos_buffer("TEST_LONGITUDE");
    }
private:
    void moos_to_goby(const CMOOSMsg& moos_msg) override {
        if(moos_msg.GetKey() == "TEST_LATITUDE" && moos_buffer().count("TEST_LONGITUDE")) {
            protobuf::GPSPosition pos;
            pos.set_time(moos_msg.GetTime());
            pos.set_latitude(moos_msg.GetDouble());
            pos.set_longitude(moos_buffer().at("TEST_LONGITUDE").GetDouble());
            goby_comms().publish<groups::gps_data>(pos);
        }
    }
};

extern "C"
{
    void goby3_moos_gateway_load(goby::MultiThreadApplication<GobyMOOSGatewayConfig>* handler)
    { handler->launch_thread<NavTranslation>(); }
    void goby3_moos_gateway_unload(goby::MultiThreadApplication<GobyMOOSGatewayConfig>* handler)
    { handler->join_thread<NavTranslation>(); }
}

```

Goby to  
MOOS

MOOS to  
Goby

# BREAK

---

5 minute break

Comments, questions, feedback: Please leave in Slack  
channel!

# Preview: Trail Example

---

This week we will focus around a particular deployment:

- Fleet of ~10 autonomous underwater vehicles (AUVs) trailing a unmanned surface vehicle (USV)
- USV following pre-planned route.
- Topside (ship or shore station) communicating to the USV.

Communications:

- AUV <-> USV: Acoustic communications
- USV <-> topside: Satellite communications

*Run trail example*

# Alpha Example

---

Let's put aside the trail example, and build back up to it.

This is the “s1\_alpha” mission from MOOS-IvP but with:

- goby\_frontseat\_interface with basic sim plugin  
(connects to goby\_basic\_frontseat\_simulator)
  - could swap plugins to use real vehicle or vehicle sim
- goby3\_course\_usv\_manager
  - publishes data to topside
- gobyd with satellite link simulated via UDP multicast  
Topside with gobyd, goby3\_course\_topside\_manager,  
goby\_opencpn\_interface, goby\_geov\_interface.

# Launching

---

```
cd goby3-course/launch/alpha  
./usv.launch  
(new terminal)  
./topside.launch  
(new terminal)  
openCpn  
(new terminal)  
google-earth-pro  
(click deploy from pMarineViewer)  
...  
to end, <CTRL>+C on usv.launch, topside.launch
```

# Debug logger

---

All Goby applications have access to `goby::glog` which is much like `std::cout`, except `glog` has:

- runtime configurable verbosity settings  
(`warn > verbose > debug1 > debug2 > debug3`)
- thread safety (mutex lock from `.is_*`() to `std::endl`/  
`std::flush`)
- ability to write to terminal and file(s) simultaneously
- support for colors and grouping of streams

```
glog.is_verbose() && glog << "This is verbose" <<  
std::endl;
```

# Configuration

---

Goby applications will automatically parse command line parameters and/or configuration files using a given “Configurator”

The default configurator is ProtobufConfigurator:

- The application writer defines a Protobuf message that forms the valid configuration for that application.
- The message is then used to automatically define command line parameters (`> app --help`) and configuration file syntax (`> app --example_config`)
- Alpha mission configs are in topside\_config, usv\_config directories

# Configuration (app)

All applications will have an common “app { }” block which can be used to change:

- glog verbosity and output locations
- simulation settings (faster-than-realtime clock)
- geodesy (basic UTM projection)

```
app {
    glog_config {
        file_log {
            file_dir: "../../logs/usv" verbosity: DEBUG2
        }
    }
    simulation {
        time { use_sim_time: true warp_factor: 10 }
    }
}
```

# Configuration (interprocess)

All applications that communicate on interprocess will have an common “interprocess { }” block:

- platform name (used to form socket path with using the default transport, `transport: IPC`)
- transport type (IPC or TCP), address, port for TCP
- ZeroMQ queue size and worker thread count

```
interprocess {
    platform: "usv"
} // or

interprocess {
    platform: "usv"
    transport: TCP
    ipv4_address: "192.168.1.1"
    tcp_port: "11144"
}
```

# Debugging

---

goby\_launch by default sends each application's output to GNU screen.

- See all screens: screen -ls <filter>
- Attach a screen: screen -r usv.gobyd
- Detach a screen: <CTRL>+A, D

goby applications default to QUIET (no terminal output). To increase verbosity use -v[vvv] on command line (or in .launch file) where:

-v is glog\_config {tty\_verbosity: VERBOSE}  
-vv is glog\_config {tty\_verbosity: DEBUG1}  
-vvv is glog\_config {tty\_verbosity: DEBUG2}  
-vvvv is glog\_config {tty\_verbosity: DEBUG3}

and/or use NCurses GUI (-n)