Goby3: A new open-source middleware for nested communication on autonomous marine vehicles

Toby Schneider
GobySoft, LLC
Woods Hole, MA, USA
Email: toby@gobysoft.org

***Abstract*—Software systems for robotics increasingly require support for robust interprocess communication with common interfaces, which has given rise to the use of "middleware" software projects. However, autonomous underwater vehicles (AUVs) have a significantly different intervehicle communication regime than other branches of robotics due to the physical realities of the ocean as a communication medium.**

Goby3 is a new middleware, the first specifically designed to address intervehicle, interprocess, and interthread communication on AUVs in a unified manner. Goby3 is based on C++11 and is minimally restrictive on the types that can be published and subscribed using it. A reference implementation is given that uses C++ shared pointers for interthread, ZeroMQ for interprocess, and Goby-Acomms for intervehicle communication. This implementation is shown to give similar or better performance to existing middlewares.

I. Introduction

Developing and maintaining software systems for autonomous underwater vehicles (AUVs) and autonomous surface craft is rapidly becoming one of the most complex tasks for successful development of these platforms as hardware components (sensors, actuators, computing elements) reach a plateau of maturity and commodization.

Managing this complexity can be accomplished through several means: abstraction and standardization of interfaces, modularization of software components, and leveraging of existing open source resources. To assist in these goals, various software projects have become widely used in the marine robotics community. These projects are referred to as "middlewares" for their intermediary role in between the operation system resources (especially those pertaining to communication) and the robotic application software. Several examples of middleware that have been used on marine vehicles include MOOS [1], the Robot Operating System (ROS) [2], and the Lightweight Communications and Marshalling (LCM) project [3].

From the prospective of middleware, the marine environment poses a significant unique challenge: the extremely low throughput typically available for intervehicle communications, since acoustic modems and low throughput electromagnetic-based systems (e.g. satellite modems) are often the only practical choice. None of the existing middlewares address this specific challenge, and approaches to intervehicle communication tend to be decoupled from intravehicle communication. At the same time, users are increasingly fielding multiple AUVs due to reduced vehicle cost and increased need for spacial coverage.

Thus, version 3 of the Goby Underwater Autonomy project (Goby3) offers a new middleware specifically designed for allowing nested autonomy [4], where decisions are made as close to the data source as possible to avoid excessive data traffic. However, as needed, messages can be requested to a scope further from the source. To increase its general applicability, the design of Goby3 can support any choice of transport mechanisms used (e.g. TCP/IP for interprocess) or data marshalling schemes (e.g. Google Protocol Buffers, DCCL, JSON, msgpack). However, a reference implementation that makes use of various high-quality open source libraries is presented for immediate use by the community.

A. *Existing middleware used in the marine community*

To the degree any middleware is run on an marine robot, the most common choices are ROS or MOOS, and to a lesser degree LCM. All of these middlewares provide an interprocess communication mechanism and a suggested or required marshalling scheme for converting native system types (e.g. C++ classes) into bytes and vice-versa at the receiver. ROS also provides an interthread communication mechanism ("nodelets").

The ROS and MOOS transport mechanisms are built on the Transmission Control Protocol (TCP) and thus provide reliability at the transport layer for published packets. LCM uses the User Datagram Protocol (UDP) multicast functionality, and thus provides no reliability guarantees (but increased performance for high throughput applications).

ROS and LCM use a conceptually similar interface description language (IDL) that allows the user to define data structures in a language neutral format from a collection of primitive types (integers of various sizes, floating point values, strings, etc.). These message definitions are then compiled by a tool provided by the middleware into one or more language-specific representations (e.g. C++ class, Python class) that can be used in the user's code. The standalone Protocol Buffers also provides a similar (though richer) functionality, but without any transport mechanism. MOOS uses a single class for all data transferred, the C++ "CMOOSMsg," which is a thin wrapper around either a string, a double-precision floating point value, or an array of bytes. Thus, users of MOOS generally develop their own marshallings schemes or adopt a standalone library such as Protocol Buffers.

Interoperability between applications written in different middlewares is generally inefficient as it requires writing code that both ferries data between two different transport mechanisms and converts between similar yet incompatible data representations (marshalling schemes). Goby3 aims to improve this situation by transporting objects of any types that can be serialized to bytes in a cross-platform compatible manner. This includes types from existing middlewares (e.g. LCM types, ROS `msgs`) and standalone projects (e.g. Protocol Buffers, msgpack).

B. *Nested Communications*

Nested communications (which is a subset of Nested Autonomy [5]) is a concept that splits the collection of possible communicating entities into subgroups where each subgroup shares a common order-of-magnitude with regards to data throughput. For example, processes on a single vehicle will likely communicate at similar speeds, regardless of whether they reside on a single computer or multiple computers, given the speed of copper- and fiber-based Ethernet. However, processes between vehicles will communicate at a vastly different rate if the two vehicles are only linked by underwater acoustic wireless connections.

The innermost scope is that which communicates the fastest, out to the slowest outermost scope. All messages sent to outer scopes are automatically sent to all inner scopes. An illustration of four possible scopes is given in Fig. 1 and the order of magnitude data transfer speeds are shown in Fig. 2.

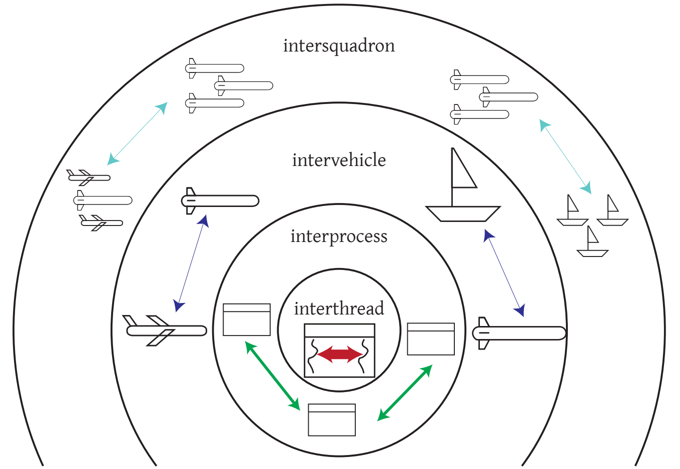

Fig. 1. Nested communications using the scopes in the Goby3 reference implementation.

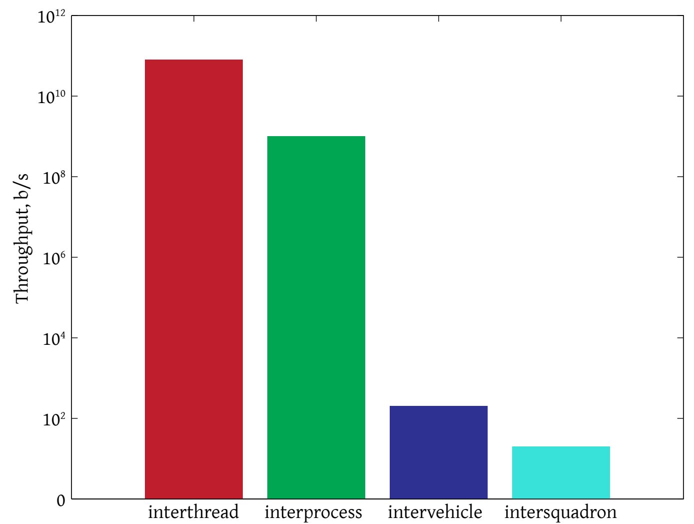

Fig. 2. Logarithmic plot of example order of magnitudes for various nested scopes, based on DRAM speeds for interthread, Gigabit Ethernet for interprocess, and various rates of the WHOI acoustic Micro-Modem for intervehicle and intersquadron (assuming lower bit rate for longer range on the latter).

II. The Publish/Subscribe model using nested communications

The publish/subscribe paradigm is common to many of the middlewares, since its asynchronous nature lends itself well to systems that have many heterogenous parts operating on different realtime constraints.

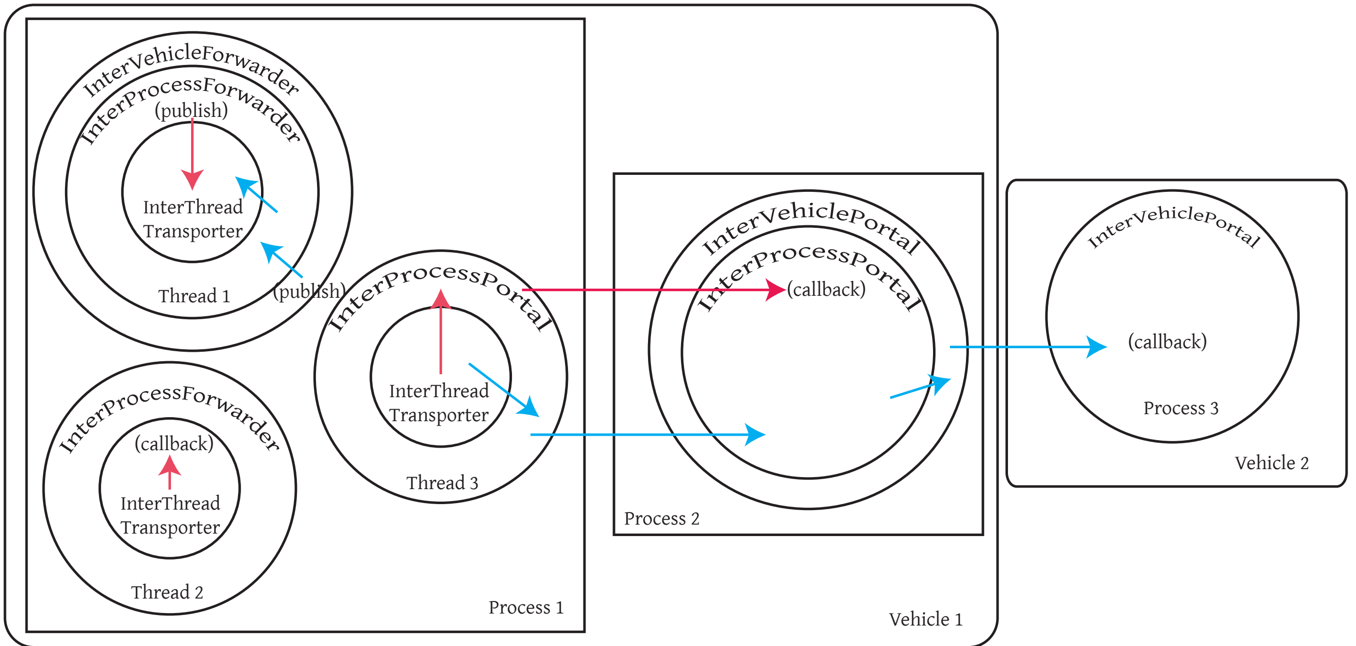

Fig. 3. An example of data flow and the interaction of Forwarder and Portal classes. In this example, one data type/group (illustrated in red) and another data type/group (illustrated in blue) is published by thread 1 (of process 1 on vehicle 1). The red data are subscribed to by thread 2 (of process 1 on vehicle 1) and also by the single-threaded process 2 (of vehicle 1). The blue data have been subscribed to by process 3 (on vehicle 2). This simplified example assumes vehicle2 only runs one process, and process 2 has only one thread.

In the nested implementation of publish/subscribe in Goby3, an entity publishes its value (of some type) to a "group" at the given nested scope. On the first publication, it is advertised to any existing nodes that are subscribed to that group and type. If no subscribers exist, the published values are not transmitted anywhere.

Multiple types can be transmitted in the same group, but subscribers will only receive (in the form of a callback function) the type(s) they have specifically subscribed for. This allows both publications and subscriptions to be strictly typed and not require any parsing or serialization of messages directly by the end-user.

Subscribers can request a variable of a given type or types from a group. Subscriptions will be forwarded into the innermost scope that is fully qualified, and if the variable is being published at that scope, all future publications will be escalated to the subscriber's scope. This allows data to stay as local as possible until needed by an outer scope, saving bandwidth while maintaining operational flexibility.

Each layer of the nested communications is implemented through two C++ classes, the Forwarder (e.g. InterVehicleForwarder) class and the Portal class (e.g. InterProcessPortal). The Forwarder class is used by entities one scope inside (threads in the case of InterProcessForwarder, processes in the case of InterVehicleForwarder, etc.) which do not directly talk to the transport mechanism at that layer. As the name implies, the Forwarder passes publications and receives subscribed data from the Portal class via the inner transport layer (e.g. interthread in the case of InterProcessForwarder). The Portal class actually communicates on the wire with other instantiations of the Portal, and only one Portal exists for each entity at that scope (e.g. one InterProcessPortal for each process, one InterVehiclePortal for each vehicle).

The publish/subscribe interface for the end-user application is essentially the same for for Portals and Forwarders, except Portals need to be configured with the transport related parameters and Forwarders do not since they use the inner scope transport mechanism.

The exception to this is the interthread layer, which has only one implementation class (the InterProcessTransporter) which is shared by all the threads and is essentially the same as a Portal class design (since a Forwarder would be meaningless as there is no further inner scope to forward data through).

An example of the interaction between these classes is given in Fig. 3.

III. Reference Implementation

The Goby3 reference implementation is entirely in C++ as defined by the 2011 standard (C++11). C++11

TABLE I
TYPES SUPPORTED BY GOBY3 IN COMPARISON WITH EXISTING MIDDLEWARES

	Goby3	ROS	LCM	MOOS
interthread	Any C++ type	ROS msg/srv		
interprocess	Any serializable type	ROS msg/srv	LCM types	CMOOSMsg (string, double, bytes)
intervehicle	DCCL messages			

provides numerous new features that are necessary to create this middleware without significant reliance on external projects such as Boost. The major new features used by Goby3 are C++ std::threads, lambda expressions, std::function, and smart pointers (std::shared_ptr).

Goby3 includes a reference implementation that uses three scopes and corresponding transport mechanisms:

- **interthread**: Zero-copy communication between threads using C++11 shared pointers. Since no data are copied (just the pointers), the types used at the interthread layer do not need to be serializable into a byte stream. In our experience, multithreading can be error prone and confusing for newcomers to AUV software. The Goby3 interthread layer allows the transfer of any C++ objects between threads in a publish/subscribe manner that shares the same paradigm and software interface as the outer layers. This allows application designers to write reliable and memory safe multithreaded applications using the same familiar publish/subscribe paradigm, without understanding or debugging custom thread data sharing concepts.
- **interprocess**: TCP/IP or UNIX socket communication using ZeroMQ [6]. This layer of Goby3 uses the ZeroMQ transport layer to allow either single-computer interprocess communications via UNIX sockets or multi-computer (e.g. connected by a gigabit copper Ethernet) interprocess networks using TCP. The assumption is that these processes are all resident on a single vehicle or other node (mooring, or topside on the research vessel).
- **intervehicle**: Acoustic, satellite, or other "slow-link" communications using the Goby-Acomms library [7].

The supported data types (or marshalling schemes) for each of the three scopes in the reference implementation are:

- **interthread**: Any C++ type.
- **interprocess**: Any serializable C++ type (e.g. Protocol Buffers, the Dynamic Compact Control Language version 3 (DCCL3), or msgpack).
- **intervehicle**: The Dynamic Compact Control Language version 3 (DCCL3) [8]. DCCL3 is a interface description language (based on Protocol Buffers) and extensible suite of marshalling algorithms specifically designed for extremely low throughput links such as acoustic modems.

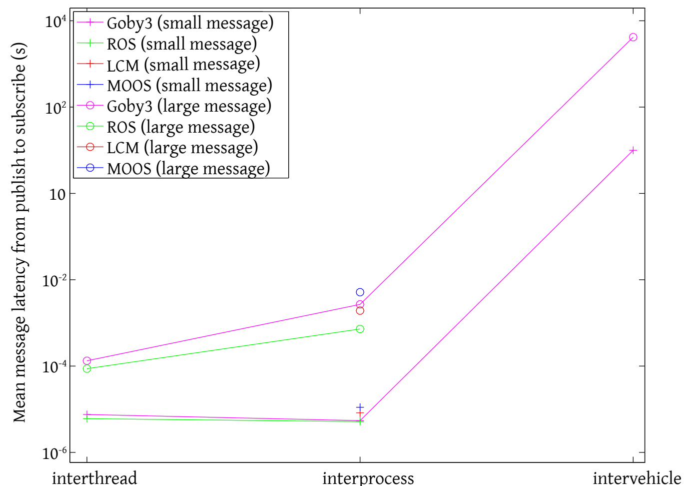

Fig. 4. Logarithmic plot of measured throughputs for the scopes supported by a given middleware (interthread for Goby3 and ROS, interprocess for all). The small message used was a hypothetical sample from a CTD sensor encoded in the middleware's native marshalling scheme, with Protocol Buffers used for Goby3 (about 30 bytes encoded depending on the middleware). The large message was 1 megabyte of text data. For the intervehicle case on Goby3, an acoustic Micro-Modem data rate was used for the small message, and an Iridium satellite data rate for the large message.

These data types are also summarized in Table I in comparison to the data types that other existing middlewares use.

IV. RESULTS

The success of some of the major aims of Goby3 will only be borne out with use by the wider community: 1) bringing the ease of publish/subscribe to interthread and intervehicle communications and 2) easing the interoperability of different systems by relaxing the marshalling scheme requirements that existing middlewares have.

However, the performance of Goby3's reference implementation needs to be acceptable to merit the wider

use that will be necessary to truly assess and realize the aforementioned aims. Thus, benchmark testing of the Goby3 reference implementation was performed against the MOOS, LCM, and ROS middlewares. Two sizes of messages were tested: a small one on the order of tens of bytes, with the exact size depending on the details of the middleware's marshalling scheme and a large one equal to about 1 megabyte. Ten thousand to one million messages of each size were published and subsequently received by a subscriber. The mean time to publish, transfer, and receive each message was calculated. The results of this testing are plotted in Fig. 4 for the scopes supported by each middleware. This figure shows that the performance of Goby3 is similar or better to that of the comparable middlewares. Some of the minor performance deficit relative to ROS is due to the flexibility of Goby3 (which allows any serializable type whereas ROS only allows ROS `msgs`).

V. Conclusion

Existing middlewares do not address the "slow link" problem that is very common for marine intervehicle communications. In the author's experience, solutions to intervehicle communication tend to be "add-ons" to the main middleware used for interprocess communication, and thus tend to be difficult to extend or modify when new data needs to be shared between vehicles or the operator topside. Goby3 is designed to provide a common interface to ease this mismatch.

Existing middlewares tightly couple a required transport layer with a required marshalling scheme. Goby3 relaxes the marshalling scheme requirement as much as is reasonable, allowing easier development between applications and research groups which "talk" different data marshalling "languages". In addition, the core design of Goby3 does not mandate any particular transport layers so a different choice (e.g. UDP for interprocess) could be implemented by the user while still using other parts (e.g. intervehicle and/or interthread) from the reference implementation. This modularity aims to provide flexibility at the same time as providing working, field-quality level code that is ready to use.

Goby3 is open source software (distributed under the LGPL license) and at the time of this writing is in what is generally considered an "alpha" stage of development. Infrastructure (such as middlewares) are critically important pieces of AUV software, but difficult to find the time, interest, or money to create well. Thus, the more shared work that can be done in the AUV community on this topic, the better. Feedback and contributions at this stage is greatly appreciated. The project page for software, issue tracking, etc. is https://github.com/GobySoft/goby.

References

[1] M. R. Benjamin, J. J. Leonard, H. Schmidt, and P. M. Newman, "An overview of MOOS-IvP and a brief users guide to the IvP Helm autonomy software," *Journal of Field Robotics*, 2009.

[2] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2, 2009, p. 5.

[3] A. S. Huang, E. Olson, and D. C. Moore, "LCM: Lightweight communications and marshalling," in *Intelligent robots and systems (IROS), 2010 IEEE/RSJ international conference on*. IEEE, 2010, pp. 4057–4062.

[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.

[5] H. Schmidt, M. R. Benjamin, S. M. Petillo, and R. Lum, "Nested autonomy for distributed ocean sensing," in *Springer Handbook of Ocean Engineering*, M. R. Dhanak and N. I. Xiros, Eds. Springer, 2016, pp. 459–480.

[6] P. Hintjens, *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, Inc., 2013.

[7] T. Schneider and H. Schmidt, "Goby-Acomms version 2: extensible marshalling, queuing, and link layer interfacing for acoustic telemetry," in *9th IFAC Conference on Manoeuvring and Control of Marine Craft, Arenzano, Italy*, 2012.

[8] T. Schneider, S. Petillo, H. Schmidt, and C. Murphy, "The dynamic compact control language version 3," in *OCEANS 2015-Genova*. IEEE, 2015, pp. 1–7.