Goby-Acomms: A modular acoustic networking framework for short-range marine vehicle communications

Toby Schneider, *Member, IEEE,* Henrik Schmidt

***Abstract*—Autonomous marine vehicles are increasingly used in clusters. The effectiveness of this robotic collaboration is often limited by communications: throughput, latency, and ease of reconfiguration. We developed a modular acoustic networking framework, realized through a set of C++ libraries called `goby-acomms`, to provide collaborating underwater vehicles with an efficient short-range single-hop network. `goby-acomms` is comprised of four components that provide: 1) losslessly compressed encoding of short (i.e. tens to hundreds of bytes) messages configured in XML via the Dynamic Compact Control Language (*libdccl*); 2) a set of message queues that dynamically prioritize messages based both on overall importance and time sensitivity (*libqueue*); 3) Time Division Multiple Access (TDMA) Medium Access Control (MAC) with automatic discovery (*libamac*); and 4) an abstract acoustic modem driver with a subclass for the WHOI Micro-Modem acoustic modem (*libmodemdriver*). `goby-acomms` has been tested in numerous field trials using a variety of underwater vehicles and surface nodes using the WHOI Micro-Modem.**

***Index Terms*—underwater acoustic network, data marshalling, priority queue, acoustic MAC, AUV communication**

I. Introduction

Undersea communication over any significant range is widely accepted to be only practical using an acoustic carrier [1]. However, the quality of acoustic communications is often poor due a number of physical realities of acoustic waves: highly restricted bandwidth (the ocean is a low pass filter for acoustics), slow phase speeds (acoustic waves are $2 \cdot 10^5$ times slower than electromagnetic waves), multipath caused by surface and bottom reflections, and Doppler spread caused by moving waves and the relative motion of senders and receivers (which are a significant fraction of the slow speed of sound). The difficulty of obtaining high rate acoustic transmissions due to the ocean environment and how this impacts acoustic networking is summarized by many researchers such as Baggeroer [2], Kilfoyle [3], Preisig [4], Stojanovic [5], and Partan [6].

Transmission of digital signals through the ocean has been investigated since at least the 1950s [7], but only recently have relatively mature acoustic modems been available, such as the WHOI Micro-Modem used to develop and test this work [8]. With this advent of a reliable hardware layer it is possible to build and test networking systems. Due in part to decreasing costs, marine robots (such as autonomous underwater vehicles or AUVs) are increasingly being used in clusters to perform tasks collaboratively such as the work performed at MIT [9] [10] [11] and elsewhere [12]. Almost by definition, collaborative autonomy requires runtime communications between vehicles. The need for intervehicle communication for the purposes of research into underwater autonomy motivated the development of `goby-acomms`.

Networking is a well studied problem in the terrestial domain; an example is the ubiquitous Internet Protocol Suite. However, the aforementioned limitations to throughput and latency in an underwater acoustic network suggest we should perform careful analysis before applying terrestial networking solutions to the marine environment. Specifically, we suggest that certain tradeoffs of efficiency for abstraction that are desirable on high throughput, low latency links involving thousands of computers are not desirable for the low throughput, high latency acoustic links involving at most tens of autonomous underwater vehicles (AUVs). A common form of networking abstraction is the concept of "layers" (together, the layers form a network "stack"). The Open System Interconnection Reference Model (OSI Model) presented in [13] provides a framework for this type of abstraction. In the OSI Model, each layer is abstracted from the previous layer; that is, higher levels do not need to concern themselves with the implementation details of lower levels. This abstraction allows for complicated systems to be broken into more manageable pieces and is likely a contributer to the success of the internet. However, such layering comes with tradeoffs. Higher layers duplicate header information (such as addressing) and error checking that may be already implemented on one or more of the lower layers. Hence, with `goby-acomms`, in order to produce shorter messages, we chose to maintain the general concept of networking layers (where each is a separate C++ library), but with more explicit and implicit interactions between layers.

The layers (or modules) of `goby-acomms` are summarized in Table I with an approximation of the corresponding layer(s) of OSI Model, and illustrated generally in Fig. 1 and with more detail in Fig. 2. While each layer is dependent on one or more of the other layers, any layer could be replaced as long as the replacement fulfills the necessary interface requirements. This modularity of `goby-acomms` should improve its flexibility for use in a variety of future acoustic networks, as needs

This work was funded by the Office of Naval Research (ONR) under projects N00014-08-1-0011 and N00014-08-1-0013.

T. Schneider and H. Schmidt are with the Center for Ocean Engineering in the Department of Mechanical Engineering, Massachusetts Institute of Technology (MIT), Cambridge, MA, 02139 USA. T. Schneider is also affiliated with the Woods Hole Oceanographic Institution (WHOI), Woods Hole, MA, 02139 USA, as a student in the MIT/WHOI Joint Program (e-mail: tes@mit.edu, henrik@mit.edu).

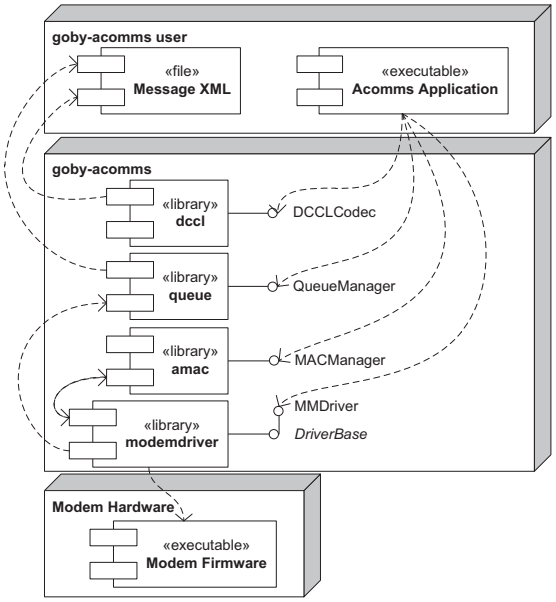


Fig. 1: Unified Modeling Language (UML) component model of `goby-acomms`. Dependencies are indicated with a dotted arrow pointing from an object to its dependency. The interface class to each library is given as a line terminated by a semi-circle (e.g. DCCLCodec). See Fig. 2 for a more detailed structure diagram of `goby-acomms`. UML is presented in [14].

change and new research comes to light. For example, a future project that just needs encoding could use *libdccl* alone. Or an acoustic network with an improved buffering system could replace *libqueue* while making use of the remaining layers of `goby-acomms`.

A variety of terms that may be ambiguous are clearly defined for this paper in the glossary (section VIII).

II. *libdccl*: Encoding and decoding

A. Motivation

The limited throughput constraint of acoustic communications suggests that compressing messages as much as possible is a useful goal. Also, due to the highly time varying nature of the acoustic channel, it is difficult to maintain error free transmission of long packets. For these reasons, the WHOI Micro-Modem uses frames of 32 to 256 bytes. Again due to the high error rates caused by the acoustic channel, guaranteeing receipt of multiple frames can often take an unacceptable amount of time for AUV collaboration. Thus, all of `goby-acomms` deals with data frames smaller than or equal to the size of the hardware layer's frame size. This requires that the application layer produce data that are useful or at least potentially useful as standalone frames. Given this requirement, it is not feasible to compress packets using lossless encoding that has overhead (such as Huffman coding), as the size of the tree would almost certainly be larger than the space stored (in all but the most inefficient messages). Thus, for the Dynamic Compact Control Language, the user must strictly specify and name the fields that a given message can take. Furthermore, all numeric fields must have tight bounds that represent the realistic set of values that field will take. For example, it is inefficient to use a 32-bit integer to represent the operation depth which might vary at most from 0-11021 meters on Earth and thus fit in 14 bits or less.

B. Prior work

1) Compact Control Language: libdccl owes inspiration and part of the name to the Compact Control Language (CCL) developed at WHOI by Roger Stokey and others for the REMUS series of AUVs. An overview of CCL is available in [15], and the specification is given in [16]. In our experience, before DCCL, CCL was the *de facto* standard data marshalling scheme for acoustic networks based on the WHOI Micro-Modem.

DCCL is intended to build on the ideas developed in CCL but with several notable improvements. DCCL provides the ability for messages to adapt quickly to changing needs of the researchers without changing software code (i.e. *dynamic*). CCL messages are hard coded in software while DCCL messages are configured using XML.

Also, significantly smaller messages are created with DCCL than with CCL since the former uses unaligned fields, while the latter, with the exception of a few custom fields (e.g. latitude and longitude), requires that message fields fit into an even number of bytes. Thus, if a value needs eleven bits to be encoded, CCL uses two bytes (sixteen bits), whereas DCCL uses the exact number of bits (eleven in this case). DCCL also offers several features that CCL does not, including encryption, delta-differencing, and data parsing abilities.

To the best of the authors' knowledge (which is supported by Chitre, et al. in [17]), CCL is the only previous effort to provide an open structure for defining messages to be sent through an underwater acoustic network. Other attempts have been ad-hoc encoding for a specific project. In order not to trample on Stokey's work and maximize interoperability, we have made DCCL compatible with a CCL network, giving DCCL the CCL initial byte flag of 0x20 (decimal 32). This allows vehicles using CCL and DCCL to interoperate, assuming all nodes have appropriate encoders for both message languages.

2) Text Encoding: Two approaches to encoding that have proven useful in other applications for compressing data are dictionary coders (e.g. LZW [18]) and entropy coders (e.g. Huffman coding [19]). Both of these are successful on sparse data, such as human readable text. Their utility for the types of messages encountered commonly in marine robotics is limited, however. These messages tend to be short and full of numeric values, whose information entropy is much greater than that of human generated text.

Furthermore, the overhead cost incurred by these text encoders means that the compressed message may not be more efficient than the original message until a sizable amount of data (perhaps several kilobytes) has been encoded. This exceeds the size of individual frames in the WHOI Micro-Modem, meaning that in messages would have be split across frames and reassembled. Given the low throughput and high

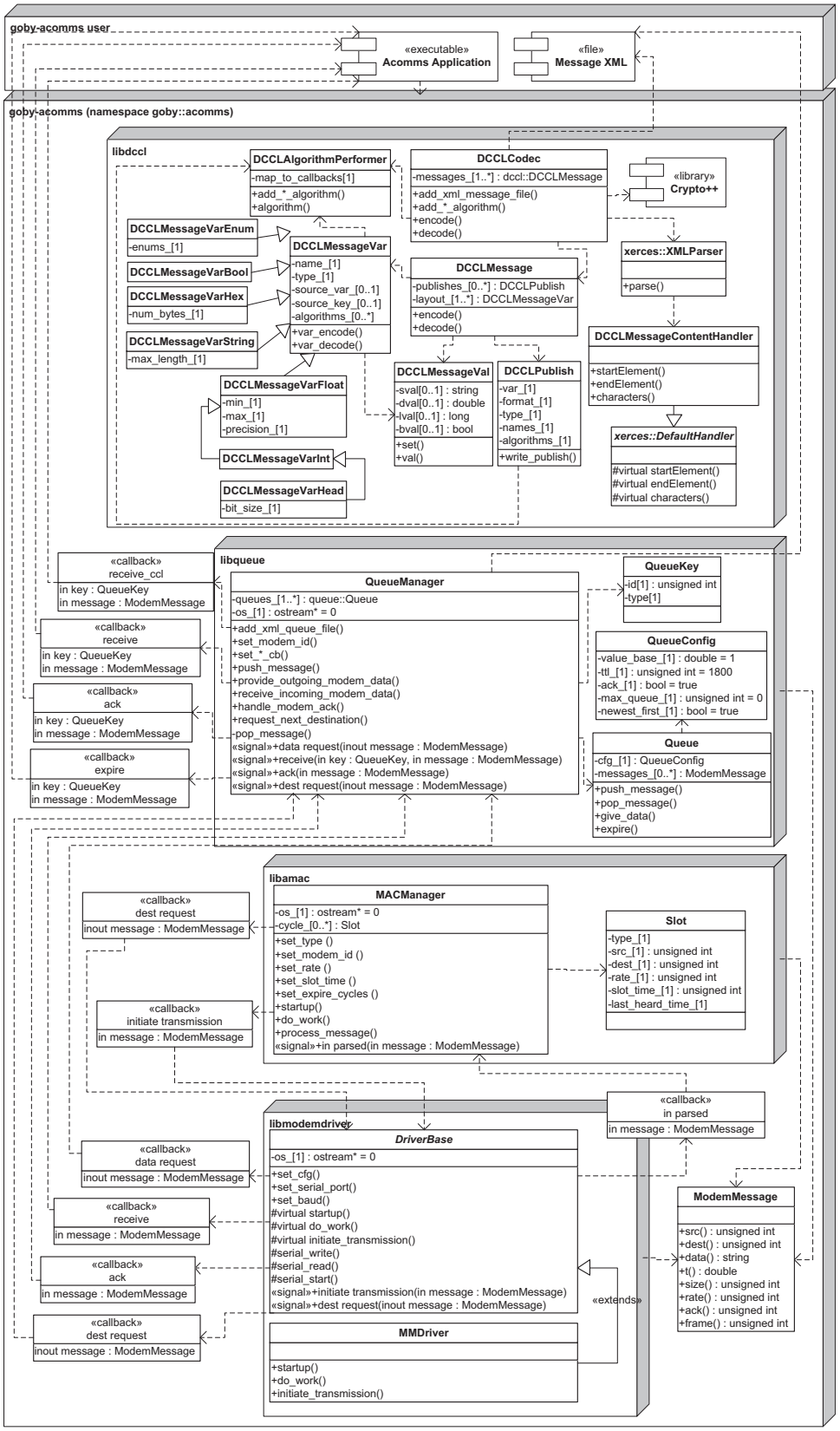


Fig. 2: UML structure model of `goby-acomms` showing all the classes and most important members and methods. See Fig. 1 for a broad overview component diagram. Each library in `goby-acomms` has an interface class (DCCLCodec, QueueManager, MACManager, and DriverBase), allowing any module to be replaced by a different library with the same interface. Since digital communications, especially acoustic communciations, are highly asynchronous, the libraries (except *libdccl*) communicate via asynchronous signals implemented via callbacks.

TABLE I: Comparison of goby-acomms layers with those of the OSI Model

OSI Model layer	goby-acomms layer (library)	Provides
Application	Provided by the goby-acomms user.	Configuration and data.
Presentation	*libdccl* [a]	Encoding and decoding.
Session	Not used. Sessions are passive.	
Transport	*libqueue* [b]	Priority buffering, concatentation of multiple DCCL messages, and guarantee of receipt.
Network	Not provided (goby-acomms is single-hop).	
Data Link	*libamac* [c]	Division of time into slots for multiple vehicles over the half-duplex link.
	libmodemdriver [d]	Configuration of, interaction with, and abstraction of the physical layer.
Physical	e.g. WHOI Micro-Modem	Transmission and receipt of messages.

[a] section II
[b] section III
[c] section IV
[d] section V

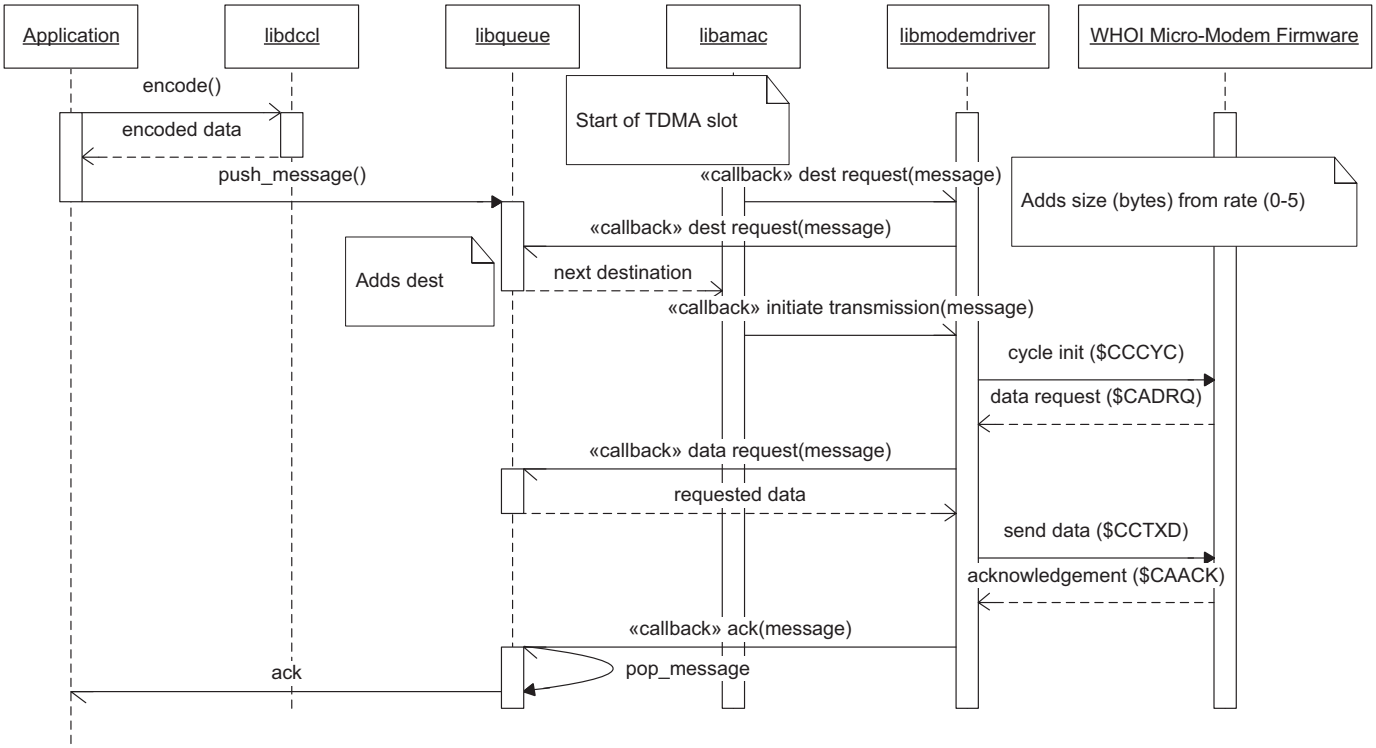


Fig. 3: The UML sequence diagram for sending a message using all the goby-acomms components. See Fig. 2 for the corresponding class structure diagram.

error rate of the acoustic channel, it is impractical to attempt to send a message that is more than several frames before being decodable. Furthermore, the resulting message from these text encoders is variable length, as the compressibility depends on the input data. This can cause further difficulties transporting these data across the acoustic network.

Given these considerations, we decided that currently available text encoders would not an acceptable solution to the problem at hand, i.e. creating short messages for acoustic communications.

3) Abstract Syntax Notation One: Abstract Syntax Notation One (ASN.1) is a mature and widely used standard for abstractly representing data structures (or messages) in a human-readable textual form. It also specifies a variety of rules for encoding data using the ASN.1 structures. In both these areas, ASN.1 is similar to DCCL: DCCL also provides a structure language (based on XML in this case), and a set of encoding rules. In fact, the rules used by DCCL are very similar to the ASN.1 unaligned Packed Encoding Rules (PER). For a good treatment of ASN.1, see Larmouth's book [20].

If DCCL used the ASN.1 notation, it could hope to draw on the advantages of being standards compliant. However, DCCL does not currently use the ASN.1 representation at this time for two main reasons:

1) Given the severe restrictions on message size due to the acoustic modem hardware, existing ASN.1 structures are unlikely to be useful, unless the designers were originally careful in specifying bounds on numerical types

(e.g. INTEGER) and minimizing use of string types (UTF8STRING/IA5STRING). Thus, for simplicity of the DCCL specification, the authors prefer the XML specification given in section II-D and currently used by DCCL.

2) ASN.1 structures are commonly "compiled" into source code which is then compiled into the finished program. This does not allow for dynamic message structures, which is at the core of the DCCL goal. DCCL does not compile the message structure, but rather translates it into a collection of objects at runtime. We argue that for the underwater robotics research community, at least, changes to messages should not need recompilation of code. Perhaps as the field matures and messages become widely used and standardized, support for compiling of messages will become more desirable.

Support for ASN.1 may become a desirable goal in the future to take advantage of the knowledge base and experience of this well accepted standard. However, we will likely have to choose a tightly reduced subset of the ASN.1 specification to meet the restrictive demands of the underwater acoustic channel. One possible path would be to match the XML definition of DCCL to the ASN.1 XML Encoding rules. Then, either the ASN.1 definition or XML definition could be used to encode messages using the Packed Encoding Rules, which are similar to the rules already used in DCCL (see section II-E).

C. Design overview

DCCL is comprised of two components: 1) a structure language based on XML with which to define messages (described in section II-D); and 2) a C++ library (*libdccl*, detailed in section II-E) that validates the XML structure and implements consistent encoding and decoding of each message.

In order to produce messages as small as possible, DCCL offers these features:

- Defined bounded field types with customizable ranges. For example, an integer with minimum value of 0 and maximum value of 5000 takes 13 bits instead of the 32 bits often used for the integer type, regardless of whether the full integer type is needed.
- Dissolved byte boundaries (unaligned messages): fields in the message can be an arbitrary number of bits. Octets (bytes) are only used in the final message produced.
- Delta-difference encoding of correlated data (e.g. CTD instrument data): rather then sending the full value for each sample in a message, each value is differenced from both a pre-shared key and the first sample within the message. This feature is described in more detail in section VI-E.

We also wanted to remove some of the complexity and potential sources of human error involved in binary encoding and bit arithmetic. To make DCCL straightforward, we made several design choices:

- All bounds on types can be specified as any number, such as powers of ten, rather than restricting the message designer to powers of two. This leads to a small inefficiency since the message is encoded by powers of two, but this drawback is balanced by the value of simplicity since the human mind is much more comfortable with powers of ten than powers of two.
- XML is the basis of the markup language that defines the structure of a DCCL message. XML was chosen for its ubiquity (e.g. XHTML for the web, RSS for news, KML for Google Earth), which means a host of tools are already available for editing and checking the validity of DCCL messages.
- Encoding and decoding for basic types are predefined and handled automatically by the DCCL C++ library (*libdccl*), meaning that in the vast majority of the cases no new code needs to be written to create or redefine a DCCL message. Writing code on cruises is always a risky endeavor, and minimizing that risk is important to maximizing use of ship time. However, flexibility to define custom algorithms to assist with encoding is provided for the fairly rare case when the basic encoding does not satisfy the needs of a particular message.

D. Defining Messages

DCCL messages are defined using a custom language built from XML. Thus, the message structure is given by a text file composed of a series of nested tags (e.g. <`message`>). Such files can be edited by any text editor or any of a large of tools designed specifically for composing XML. The basic tags needed to define a message are given in this section. A number of additional tags are available for interacting with the vehicle's autonomy architecture; these tags are described in section VI-A.

1) XML Specification: The full XML schema is available with the source code at <http://launchpad.net/goby>; here we give a summary of the tags. A DCCL message file always consists of the root tag <`message_set`> which has one or more <`message`> tags as its children. The <`message`> children are as follows:

- <`id`>: an identification number (9 bits, so <`id`> $\in [0, 511]$) representing this message to all decoding nodes [`unsigned integer`].
- <`name`>: a name for the message. This tag and <`id`> must *each* be a unique identifier for this message. [`string`].
- <`size`>: the maximum size of this message in bytes [`unsigned integer`]. DCCL may produce a smaller message, but will not validate this message XML file if it exceeds this size.
- <`repeat`>: empty tag that can be specified to tell DCCL to repeatedly create the entire message to fill the entire <`size`> of the message.
- <`header`>: the children of this tag allow the user to rename the header parts of the DCCL message. See Fig. 4 for a sketch of the DCCL header format. These names are used when passing values at encode time for the various header fields.
  - <`time`>: seconds elapsed since 1/1/1970 ("UNIX time"). In the DCCL encoding, this reduced to seconds since the start of the day, with precision of

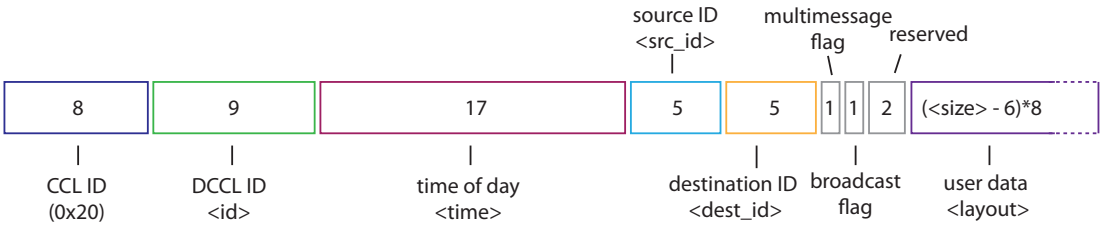


Fig. 4: Layout of the DCCL header, showing the fixed size (in bits) of each header field. The user cannot modify the size of these header fields, but can access and set the data inside through the same methods used for the customizable data fields specified in <layout>. The multimessage and broadcast flags are not used by DCCL, but are included for use by priority queuing (see Section III).

one second. Upon decoding, assuming the message arrives within twelve hours of its creation, it is properly restored to a full UNIX time.
  - <name>: the name of this field; optional, the default is "_time". [string]
- <src_id>: a unique address (<src_id> $\in [0, 31]$) of the sender of this message. For a given experiment these short unique identifiers can be mapped on to more global keys (such as vehicle name, type, ethernet MAC address, etc.).
  - <name>: default is "_src_id". [string]
- <dest_id>: the eventual destination of this message (also an unsigned integer in the range [0,31]). If this destination exists on the same subnet as the sender, this will also be the hardware layer destination id number.
  - <name>: default is "_dest_id". [string]

- <layout>: the children of this tag define the generic data fields of the message, which can be drawn from any combination of the following types, summarized in Table II.
  - <bool>: a boolean value.
    - <name>: the name of this field. [string]
    - <array_length>: optional; specifying this makes this field an array of bool instead of a single bool [unsigned integer].
  - <int>: a bounded integer value.
    - <name>: see <bool><name>.
    - <max>: the maximum value this field can take. [real number].
    - <min>: the minimum value this field can take. [real number].
    - <max_delta>: gives the maximum value for the difference of delta fields when using delta-difference encoding. Optional; the use of this tag enables delta-differencing encoding. This feature is explained where it is motivated in as part of the CHAMPLAIN09 experiment in section VI-E [real number].
    - <array_length>: see <bool><array_length>.
  - <float>: a bounded real number value.
    - <name>: see <bool><name>.
    - <max>: see <int><max>.
    - <min>: see <int><min>.
    - <max_delta>: see <int><max_delta>.
    - <precision>: specifies the number of decimal digits to preserve. For example, a precision of "2" causes 1042.1234 to be rounded to 1042.12; a precision of "-1" rounds 1042.1234 to 1.04e3. [integer].
    - <array_length>: see <bool><array_length>.
  - <string>: a fixed length string value.
    - <name>: see <bool><name>.
    - <max_length>: the length of the string value in this field. Longer strings are truncated. <max_length>4</max_length> means "ABCDEFG" is sent as "ABCD". [unsigned integer].
    - <array_length>: see <bool><array_length>.
  - <enum>: an enumeration of string values.
    - <name>: see <bool><name>.
    - <value>: a possible value the enumeration can take. Any number of values can be specified.
    - <array_length>: see <bool><array_length>. [string].
  - <hex>: a pre-encoded hexadecimal value.
    - <name>: see <bool><name>.
    - <num_bytes>: the number of bytes for this field. The string provided should have twice as many characters as <num_bytes> since each character of a hexadecimal string is one nibble (4 bits or $\frac{1}{2}$ byte). [unsigned integer].

2) Message Design: When designing a DCCL message, a few considerations must be made. Each message needs to be given a <name> and <id> unique within the DCCL network that this message is intended to live. Sometimes messages may have limited scope or may be mutually exclusive, in which case duplicate <id> numbers may be assigned.

Furthermore, the overall size of the message needs to be determined. This may be a constraint imposed by the hardware layer that this message is intended to traverse. In the case of the WHOI Micro-Modem, this should match the frame size of the intended data rate to be used (32 bytes for rate 0, 64 bytes for rate 2, and 256 bytes for rates 3 and 5). The size of the

TABLE II: Types supported by the Dynamic Compact Control Language

Type Name	DCCL XML Tag	C++ Type[a]
Bounded integer	<int>	long int
Bounded real	<float>	double
String	<string>	std::string
Enumeration	<enum>	std::string
Boolean	<bool>	bool
Pre-encoded hexadecimal	<hex>	std::string

[a] the *preferred* C++ type when encoding using *libdccl*, however any meaningful casts from other types (using streams from the std library) will be made.

message is given by the header overhead (six bytes) and the sum of the sizes of the fields. The field sizes are calculated using the expressions given in the "Size" column of Table III. These sizes are calculated at runtime with *libdccl*, so it is rarely necessary to calculate these by hand. However, these expressions give a sense of how much space a given field will typically take, which is important when considering how to type and bound the data.

An example XML message file, showing all the field tags, is provided in Fig. 5.

E. Algorithms and Implementation

Along with the XML message structure defined in section II-D, DCCL provides a set of consistent encoding and decoding tools in the C++ *libdccl* library. The class structure of *libdccl* is modeled in Fig. 2. The tools provided by *libdccl* include:

- XML file parsing and validation using the Xerces-C++ XML Parser [21]. This ensures that the syntax of the XML file is valid and structure matches that of the DCCL schema.
- Calculation of message field sizes and comparison to the mandated maximum size (specified in the <size> tag). Messages exceeding this size are rejected and the designer must choose to remove and/or reduce fields or increase the message <size>.
- Encoding of DCCL messages using the expressions given in Table III. The user passes values of the C++ types given in Table II for all the fields in <layout> and desired fields in <header>. Fig. 5 provides an example of the encoding process for a DCCL message.
- Decoding of DCCL messages using the reciprocal of the expressions used for encoding. The user of *libdccl* will receive values of the C++ types as given in Table II for all header and layout fields.

F. Encryption

libdccl provides encryption of the <layout> portion of the message using the Advanced Encryption Standard (AES or Rijndael) [22]. AES is a National Institute of Standards and Technology (NIST) certified cipher for securely encrypting data. It has been certified by the National Security Agency (NSA) for use encrypting top secret data.

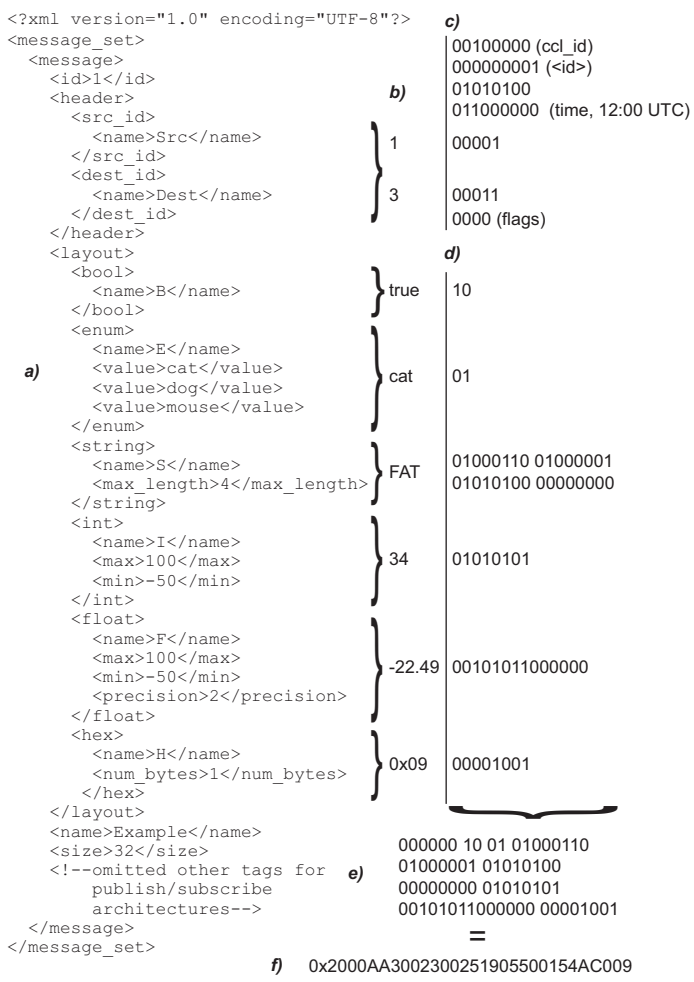


Fig. 5: Example of the DCCL encoding process. The process of encoding starts with the DCCL XML file (a). Data are provided by the application (b). *libdccl* encodes these data to binary via the algorithms given in Table III to form the header (c) and layout (d), concatenates and zero fills the encoded layout from most significant bit to closest byte (e) to produce the full encoded message (f). Finally, this point the message is encrypted (if desired).

libdccl uses a SHA-256 hash of a user provided passphrase to form the secret key for the AES cipher (see [23] for the specification of SHA-256). In order to further secure the message, an initialization vector (IV) is used with the AES cipher. The IV used for DCCL is the most significant 128 bits of a SHA-256 hash of the header of the message. Since the message header contains the time of day, it provides the continually changing value required of an IV. This ensures that the ciphertext created from the same data encrypted with the same secret key will only look the same in the future on a given day on the exact second it was created. The open source Crypto++ library available at [24] is used to perform the cryptography tasks.

G. User supplied algorithms

While the basic encoding expressions given in Table III are sufficient for representing most data, occasionally the

TABLE III: Formulas for encoding the DCCL types.

DCCL Type	Size (bits)	Encode[a]
<bool>	2	$x_{enc} = \begin{cases} 2 & \text{if } x \text{ is true} \\ 1 & \text{if } x \text{ is false} \\ 0 & \text{if } x \text{ is undefined} \end{cases}$
<enum>	$\lceil \log_2(1 + \sum \epsilon_i) \rceil$	$x_{enc} = \begin{cases} i+1 & \text{if } x \in \{\epsilon_i\} \\ 0 & \text{otherwise} \end{cases}$
<string>	$length \cdot 8$	ASCII[b]
<int>	$\lceil \log_2(x_{max} - x_{min} + 2) \rceil$	$x_{enc} = \begin{cases} \text{nint}(x - x_{min}) + 1 & \text{if } x \in [x_{min}, x_{max}] \\ 0 & \text{otherwise} \end{cases}$
<float>	$\lceil \log_2((x_{max} - x_{min}) \cdot 10^{prec} + 2) \rceil$	$x_{enc} = \begin{cases} \text{nint}((x - x_{min}) \cdot 10^{prec}) + 1 & \text{if } x \in [x_{min}, x_{max}] \\ 0 & \text{otherwise} \end{cases}$
<hex>	$num\_bytes \cdot 8$	$x_{enc} = x$

· x is the original (and decoded) value; x_{enc} is the encoded value.
· x_{min}, x_{max}, $length$, $prec$, num_bytes are the contents of the <min>, <max>, <max_length>, <precision>, and <num_bytes> tags, respectively. ϵ_i is the ith <value> child of the <enum> tag (where $i = 0, 1, 2, \ldots$).
· nint(x) means round x to the nearest integer.
[a] for all types except <string> and <hex>, if data are not provided or they are out of range (e.g. $x > max$), they are encoded as zero ($x_{enc} = 0$) and decoded as not-a-number (NaN).
[b] the end of the string is padded with zeros to $length$ before encoding if necessary.

user wants to provide a simple pre-encode and post-decode algorithm of their own. An example of this would be to encode a logarithmic value or wrap an angle into the range $[0, 2\pi]$. In this case, the field tags (i.e. <int>, <float>, <string>, <bool>, <enum>, or <hex>) all take an optional parameter `algorithm`. If the `algorithm` parameter is provided, *libdccl* calls the user provided algorithm corresponding to a callback provided on startup of the library.

For example, the user provides a callback function called `log_function` which it passes to *libdccl* as the algorithm "log". Now, when *libdccl* encounters <int algorithm="log"> it passes the value intended for that field to the `log_function`. The return value of `log_function` is then used to encode the corresponding field of the message.

Similarly, the <message_var> tag used in the <publish> sections also takes the `algorithm` parameter, allowing for post-decoding algorithms to be processed.

III. *libqueue*: Dynamic priority based buffering

A. Motivation

Field experience has taught us that in a network of AUVs, desired throughput almost always exceeds the available channel capacity. Based on the available capacity, the engineers and scientists topside prefer to see as much data as possible. The upper limit on desired throughput would perhaps be a real-time feed of all sensor data, but this can easily be order of megabits per second or higher (especially if video is involved). Maximum data throughput of available commercial modems, such as the WHOI Micro-Modem, is order kilobits per second or much lower in realistic environments. Given that this spread is unlikely to close due to the physical limitations of the acoustic carrier, users will always have be selective about which data are sent over the network.

One solution to this problem is to fix (before launch) a small subset of data that will be transmitted acoustically. Approaches to acoustic networking before *goby-acomms* such as the approaches in [25] and [26] use this solution, typically only sending a vehicle status and maybe a single sensor data type that is most relevant to the experiment at hand. This technique is generally suboptimal, given the designer must account for the worst case communications scenario or risk filling the sending buffers faster than messages can be transmitted over the channel. Due to the highly variable communications environment experienced using acoustics in the ocean, this minimax approach will under-utilize the available capacity.

To better utilize the channel, we need a solution that dynamically scales with the moment-to-moment available capacity. Qualitatively, when have poor throughput, we want to send highly valuable messages. These may correspond to status messages or time sensitive mission specific messages such as target or event detection alerts. When we have good throughput, we also want to send less critical, but still useful data. *libqueue* provides a prioritized set of buffers with time varying values to effect this desired behavior. Each DCCL message type is assigned a buffer. When the Medium Access Control requests data from *libqueue*, a priority contest is performed between all the buffers that contain messages. The winning buffer provides data from either its front or back, based on the user's desire for a first-in / first-out (FIFO) or first-in / last-out (FILO) queue respectively.

B. Prior work

1) Priority Queues: Priority queues are a widely available container type in modern programming languages (for example, C++, Java, and Python all provide implementation in their standard libraries). In a priority queue, messages are added with some priority value. When data are requested from the queue, the highest priority data are given first. *libqueue* provides a dynamic priority queue of (ordinary) double-ended queues (or deques). The dynamic part is how *libqueue* differs from ordinary priority queues. Rather than having a fixed priority, entries in *libqueue* have a priority that varies in time.

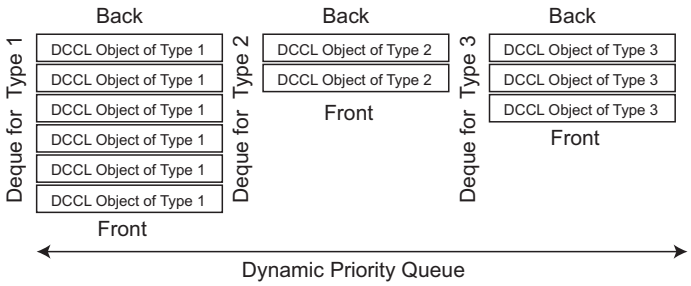


Fig. 6: Data structure of *libqueue* for three declared DCCL types. All objects within a deque are of the same DCCL type and each deque is dynamically prioritized using Equation 1. Whether a deque is accessed from the back or front is configurable for each DCCL type.

The structure of this dynamic priority queue of deques is outlined through an example in Fig. 6.

C. Implementation

Each buffer (one buffer for each DCCL type[1]) is given a base value (V_{base}) and a time-to-live (ttl) that create the priority ($P(t)$) at any given time (t):

$$P(t) = V_{base}\frac{(t - t_{last})}{ttl} \quad (1)$$

where t_{last} is the time of the last time an object was sent from this buffer.

This means for every buffer, the user has control over two variables (V_{base} and ttl). V_{base} is intended to capture how important the message type is in general. Higher base values mean the message is of higher importance. However, with only the V_{base}, higher value messages would always supercede lower value messages. AUV operators know, however, that messages of some types become more valuable if one has not been received in a long period of time, where "long" is defined by the preferences of the operators and the goals of the mission. For example, the value of a vehicle's status message grows in time as the operators become increasingly concerned with the health and location of the vehicle. The ttl parameter works to incorporate this notion of time varying value.

As the name suggests, the ttl governs the number of seconds the message lives from creation until it is destroyed by libqueue. But more importantly, the ttl also factors into the priority calculation. More time sensitive messages (those with lower ttl values) grow in priority faster.

So with these two parameters, the user can capture both overall value (i.e. V_{base}) and latency tolerance (ttl) of the message buffer. An example of how queuing manifests itself for different spacing of the Medium Access Control cycles is given in Fig. 7.

Another way to think of this dynamic priority buffering is in analogy to the economics of supply and demand. DCCL messages are analogous to perishable goods (such as food). The message sender has certain supply of each type of message. The receiver demands messages at a fixed price based on the type of good (V_{base}) that grows as time passes since the last "shipment" (successful transmission). The "perishability" of goods is reflected in the ttl. The sender uses Equation 1 to maximize his "profit" (assuming linear[2] laws).

See Fig. 2 for the software structure of *libqueue*.

D. Message stitching

While `goby-acomms` does not provide splitting and subsequent restitching of hardware layer frames to allow transmission of large DCCL messages[3], it does provide the opposite. Using the *multimessage flag* in the DCCL header (see Fig. 4), *libqueue* will stitch small DCCL messages together to form a larger hardware layer frame. For example, `goby-acomms` *will not* break a 256 byte DCCL message into parts to fit a 32 byte WHOI Micro-Modem frame, but it *will* stitch 2 16 byte DCCL messages to fit a 32 byte WHOI Micro-Modem frame.

The reasoning behind this is acoustic telemetry is so error prone that each received hardware frame should be useful in its own right. The size of hardware frames are chosen as a decent compromise between size and acceptable frame error rates. Hence, we feel providing abstraction of multiple frames per DCCL message would lead to unacceptable error rates (or unacceptable delays waiting for successful receipt and acknowledgement). However, providing facilities to fully utilize the entire hardware frame when the DCCL messages are small is useful and efficient.

IV. *libamac*: Medium Access Control

A. Motivation

The `goby-acomms` acoustic Medium Access Control module is intended to provide a robust and easily usable MAC layer. MAC is perhaps the most widely studied question in acoustic networking; Partan does a good job summarizing the various options [6]. *libamac* focuses on providing collision free communications with acceptable utilization of the available bandwidth under the following assumptions:

- All nodes are within broadcast range of one another for much of the time. (i.e., there are no hidden nodes).
- The hardware layer can support time division multiple access (TDMA).

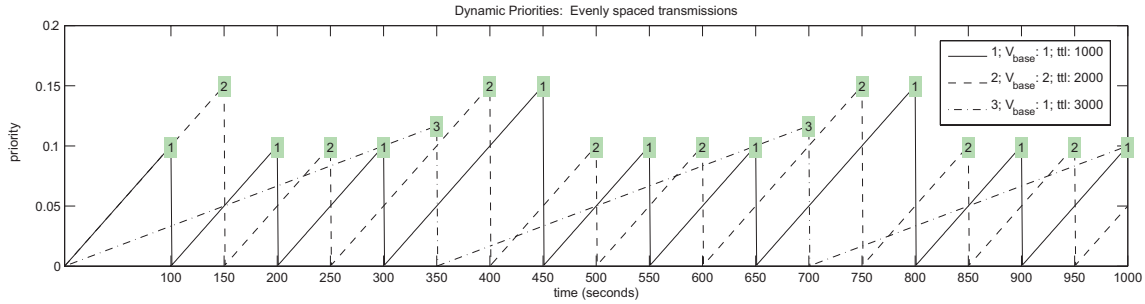
These assumptions may seem rather strong, but in our experience they are practical for the numerous present day AUV applications that do not require routing. Routing is still a significantly difficult problem for mobile acoustic networks and will likely prove the next addition to `goby-acomms`.

`goby-acomms` supports two variants of the TDMA MAC scheme: centralized and decentralized. As the names suggest, Centralized TDMA 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.

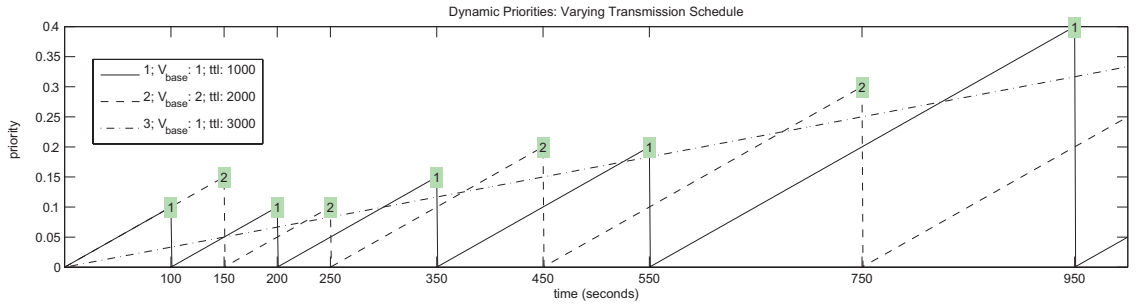
[1] as uniquely defined by <`id`> or <`name`>

[2] other functional forms (e.g. exponential) were tested but the authors could not find any substantial benefit versus linear, so linear was chosen for simplicity.

[3] for example, TCP provides such a message splitting feature



(a) TDMA Cycles are evenly spaced (period of 100 seconds). Types **1** and **2** alternate until **3** grows enough (long time since last t_{last})



(b) TDMA Cycles are unevenly spaced (increasing period of 50, 100, and then 200 seconds). **3** never gets to send in this scenario.

Fig. 7: Comparison of message priority selection for three different types (**1**, **2**, and **3**) using *libqueue*. Types **1** and **2** are equally valuable (since **1** is more time sensitive with its lower ttl and **2** is more valuable overall with a higher V_{base}). **3** is the least valuable. While clearly dependent on the spacing of transmissions, *libqueue* ensures a mix of all types of messages are sent, weighting the valuable ones more.

B. Centralized TDMA (Polling)

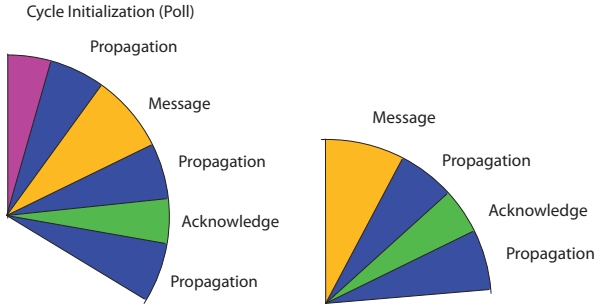
Centralized TDMA involves a master node (usually aboard the Research Vessel or on land) which initiates every transmission for the entire communcations cycle (i.e. "polls" each node for data). Thus, the other nodes are not required to maintain synchronized clocks as the timing is all performed on the master node.

This style of MAC has been widely used for small AUV operations using the WHOI Micro-Modem. Its principal advantages are that it has 1) no requirement for synchronized clocks, 2) full control over the communications cycle at runtime (assuming the master is accessible to the vehicle operators, as is usually the case); and 3) a master who can acknowledge "broadcast" messages.

However, centralized TDMA has a number of substantial disadvantages. In order for a third-party master to initiate a transmission, an acoustic packet must be sent for this initialization. This additional "cycle initialization" packet, like any acoustic message, has a high chance of being lost (after which the data are never sent because the sending node did not receive a cycle initialization message), consumes power, and lengthens the time of the communications slot. See Fig. 8 for the various parts of the communication cycle with (for Centralized TDMA) and without (for Decentralized TDMA) the cycle initialization message. The additional time required for each slot of Centralized TDMA is

$$\tau_{ci} + r_{max}/c \tag{2}$$

where τ_{ci} is the length (in seconds) of the cycle initalization packet (about one second for the WHOI Micro-Modem), r_{max} is the maximum range of the network (typically of order 1000s of meters), and c is the compressional speed of sound (nominally 1500 m/s).



(a) Centralized TDMA (b) Decentralized TDMA

Fig. 8: Comparison of the time needed for a single slot for the two types of TDMA supported by `goby-acomms` *libamac*. Eq. 2 gives the additional length of time required by the Centralized variant.

C. Decentralized TDMA with passive auto-discovery

Decentralized TDMA removes the cycle initialization packet and thus reduces the length of each slot and the chance of errors. However, it introduces the constraint of synchronized clocks[4] for all nodes, which can be somewhat tricky to

[4]the accuracy of the clock synchronization can be low relative to other timing needs such as bi-static sonar. Generally, accuracy better than 0.1 seconds is acceptable; higher inaccuracies can be handled by increasing the guard time on both sides of each slot.

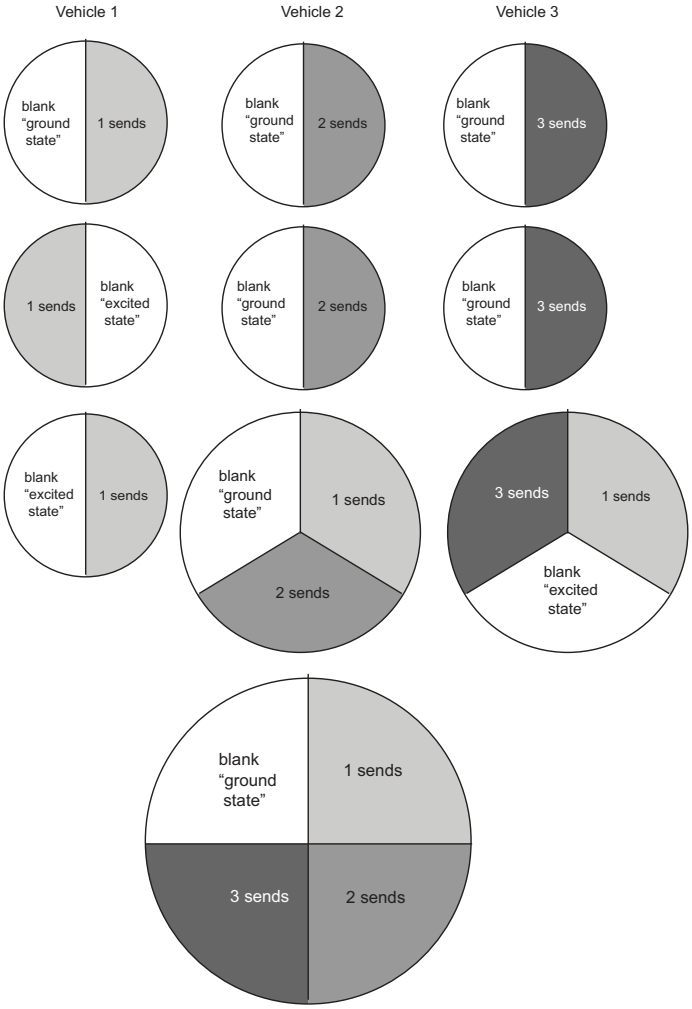


Fig. 9: Graphical example of auto discovery for three nodes launched at the same time. Each circle represents the vehicle's cycle at each time step (represented by horizontal rows) based on the vehicle's current knowledge of the world. In the first row, all vehicles only know of themselves and put the 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, see equation 3) 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.

maintain underwater See Eustice et al. [27] for an example of maintaining synchronity for navigation and communication.

Decentralized TDMA gives each vehicle a single slot in 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 the time of day). All slots are ordered by ascending acoustic MAC address (or "modem identification number"), which is an unsigned integer unique for each network.

During the runtime of the network, it is often desirable to add or remove nodes. Since the MAC is spread throughout the nodes, there is no easy way to change the cycle during runtime. *libamac* supports passive auto-discovery (and subsequent expiration) of nodes to provide a solution to this problem. This auto-discovery is passive because it requires no control messaging beyond the normal communications between nodes.

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

TABLE IV: Example initialization for the Decentralized TDMA with autodiscovery. By 135 seconds, both vehicles have discovered each other and are synchronized. Thus, no more collisions will occur. This scenario assumes that both vehicles always have some data to send during their slot.

Vehicles are discovered by shifting a blank slot in each cycle 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. In the simplified situation (which is really a worst case scenario) discovery is defined by a single vehicle transmitting during a cycle and all the others silent (the current slot is not equal to each vehicle's acoustic MAC address).

The auto-discovery works in a similar manner analogous to excitation of electrons in an atom. The blank slot is typically at the end of the cycle ("ground state"). However, depending on the "temperature" (determined by the *coolness* parameter C) the blank slot may be "excited" and moved one slot away from the end (to the penultimate slot). How often this parameter is moved is pseudorandomly determined from the time of day and the current known world state (as evidenced by the sum of the acoustic MAC addresses of all known nodes). The higher the coolness parameter C, the less likely the blank slot will be "excited" from its position at the end of the cycle. Assuming all collisions are destructive to the data received, no vehicles would ever be discovered without this movement of the blank slot. By moving the blank slot, we improve the chances that two vehicles with dissimilar views of the world will eventually discover each other. Mathematically, this placement of the blank spot in the cycle can be expressed as

$$i_{blank} = \begin{cases} i_{max} & \text{if } \lfloor t_{UTC} / \sum_i \tau(i) \rfloor \pmod{C} = \sum_i a(i) \pmod{C} \\ i_{max} - 1 & \text{otherwise} \end{cases} \quad (3)$$

where i_{blank} is the position of the blank slot in the cycle, t_{UTC} is the number of seconds since midnight Coordinated Universal Time (UTC) of the current day, $\tau(i)$ is the length of the ith slot, C is the "coolness" parameter, and $a(i)$ is the

acoustic MAC address of the node in the ith slot. Put in words, the blank slot is moved from the end of the cycle (position i_{max}) to the penultimate position ($i_{max} - 1$) when the number of cycles since the start of the day is congruent modulo C with the known world (sum of MAC addresses). Therefore, the higher C is, the less often these two values are congruent, and the less often the blank slot is "excited".

V. *libmodemdriver*: ACOUSTIC MODEM DRIVER

A. Motivation

In `goby-acomms`, the physical layer is generally assumed to be an acoustic modem, as the tradeoffs made between efficiency and abstraction are intentionally highly biased towards efficiency in `goby-acomms` because of the very low throughput acoustic channel. However, the remainder of `goby-acomms` is agnostic to the choice of acoustic modem, or even that the physical layer is acoustic at all; other very low throughput channels (e.g. satellite) could also work with the design paradigms of `goby-acomms`. *libmodemdriver* is responsible for communicating with the specific firmware of the acoustic modem of choice and abstracting that interface for the rest of `goby-acomms`. This interface works using basic virtualization in C++ where *DriverBase* provides a superclass that handles generic tasks and provides the abstract (or virtual) interface for `goby-acomms`to use. Various drivers can subclass *DriverBase* to provide the details of a particular physical device. See Fig. 2 for this class structure. Currently, the WHOI Micro-Modem is supported via the *MMDriver* class, but support for the Teledyne Benthos modem and other devices is in progress.

B. DriverBase: Abstract Acoustic Modem Driver

DriverBase provides an virtual interface to a generic acoustic modem. Some requirements are made about the acoustic modems that can be supported:

- The acoustic modem communicates with the host running `goby-acomms` via a serial port (typically RS-232 or RS-485) or an Ethernet port (using TCP/IP). The modem communicates using a delimited format (such as lines of text delimited by the newline character ASCII 0x0A).
- The acoustic modem is configurable over this communication line.
- The modem supports transmission of fixed size datagrams with optional acknowledgement of the message receipt (variable size datagrams can be used to mimic fixed sizes).

DriverBase provides:

- a class that reads serial port or TCP data into a buffer for use by the DriverBase derived class (e.g. MMDriver).
- methods to set all six callbacks provided by the derived class (receive, acknowledgement, data request, parsed incoming message, raw incoming message, raw outgoing message). Typically *libqueue* handles the receive, acknowledgement, and data request callbacks, while *libamac* needs the parsed incoming message callback to discover new vehicles (when in Decentralized TDMA mode). The raw messaging callbacks are optionally provided for the application layer to perform debugging directly on the modem, if desired.
- three virtual functions: for starting the driver, running the driver, and initiating the transmission of a message.
- a method to set configuration values for the acoustic modem. This configuration takes the form of an `std::vector` of `std::string`, the details of the contents depend on the specific modem.

C. MMDriver: WHOI Micro-Modem Driver

The MMDriver extends the DriverBase for the WHOI Micro-Modem acoustic modem. The WHOI Micro-Modem uses a serial RS-232 interface and an NMEA-0183 sentence structure.

MMDriver has been tested to work with revision 0.93.0.30 of the Micro-Modem firmware, but is known to work with older firmware (at least 0.92.0.85). The following commands of the WHOI Micro-Modem as given in [28] are implemented:

- Modem to Control Computer ($CA):
  - $CAACK - acknowledgement of sent message. Will be transformed into a *ModemMessage* and passed to the *DriverBase* acknowledgement callback.
  - $CADRQ - data request. Will be transformed into a *ModemMessage* and passed to the *DriverBase* data request callback.
  - $CARXD - received hexadecimal data. Will be transformed into a *ModemMessage* and passed to the *DriverBase* received data callback.
  - $CAREV - revision number and heartbeat. Used to check for correct clock time and modem reboots.
  - $CAERR - error message. The error message is logged to the `std::ostream` provided to *MMDriver* at instantiation.
  - $CAMPR - ping (two way ranging) response. Gives the one way travel time between nodes computed from a two way message.
- Control Computer to Modem ($CC). Also implemented is the NMEA acknowledge (e.g. $CACYC for $CCCYC):
  - $CCTXD - transmit data. Sent using the returned value from the data request callback (see $CADRQ).
  - $CCCYC - initiate a cycle. Sent on response to a call of DriverBase for initiating transmission of a message.
  - $CCCLK - set the modem clock. The clock is set on startup until a suitable value (within 1 second of the computer time) is reported back. If the modem reboots ($CAREV,...,INIT), the clock is set again.
  - $CCCFG - configure NVRAM value. All values passed to the DriverBase configuration will be passed to $CACFG at startup. For example, to send $CACFG,SRC,3, the string "SRC,3" is placed in the vector passed to DriverBase.
  - $CCCFQ - query configuration values. $CCCFQ,ALL is sent after all the $CCCFG lines to log the NVRAM parameters.

– $CCMPC - initiate a two way ranging request for the travel time between nodes.

1) Prior Work: iMicroModem: Given the number of users of the WHOI Micro-Modem, we believe that a number of special purpose "ad-hoc" Micro-Modem drivers have been written, but the details of which are not reported in the literature. Grund's iMicroModem is one of these that we have had a chance to work with since it was the Micro-Modem driver used with the MOOS automony architecture that preceeded `goby-acomms`. The MOOS is discussed in section VI-A.

MMDriver borrows a number of ideas from iMicroModem in terms of dealing with the specifics of the WHOI Micro-Modem firmware. The major difference is that MMDriver implements the interface provided by *DriverBase* and thus does not have to replicate any of the work done by *DriverBase*. This makes MMDriver shorter and simpler as it only contains details specific to the WHOI Micro-Modem. *DriverBase* handles the communication (RS-232 serial in this case), logging, and interface to the rest of `goby-acomms`.

This is standard object-oriented design, but it is mentioned here since such a design is critical for supporting multiple types of acoustic modems. Given that no standard exists for underwater acoustic telemetry, it appears the need to support various types of hardware through an abstracted generic interface will exist for some time. *MMDriver* with *DriverBase* provides that for the WHOI Micro-Modem.

VI. Field Case Studies

A. MOOS-IvP Autonomy system and middleware

`goby-acomms` was developed with and tested with the MOOS-IvP autonomy architecture [29]. MOOS-IvP is used on marine robots for autonomy level control. We use MOOS-IvP in an abstracted manner such as that different vehicle types from different manufacturers appear the same to the autonomy system and communications network. This model of operations, called Unified Command and Control is presented. The design of Unified Command Control as well as further details of all these trials except GLINT10 can be found in [30]. The results presented here are using `goby-acomms` via pAcommsHandler, an interface process between MOOS-IvP and `goby-acomms`. The experiments referenced are summarized in Table V. Since each experiment has different assets, different environmental conditions, and different objectives, it is difficult to make clear comparisons in performance from one sea trial to another. Thus, what follows is a series of case studies highlighting the development and testing of `goby-acomms`. For successful sea trials with AUVs, two goals are perhaps the most important: saving experimenters' time and improving safety of the vehicles. Any improvement in operations that touches upon these goals improves the productiveness of the experiment. These are often hard to quantify, but these case studies try to emphasize what `goby-acomms` has done on both of these fronts.

B. GLINT08

The three GLINT experiments (2008-2010) were designed to develop and test systems for multi-static active tracking of moving targets. For the 2008 experiment the first part of `goby-acomms`, the code which later became *libqueue*, was developed. We were using three CCL messages to communicate three types of data from the AUV: status (position and speed of the vehicle), contact (possible detection), and track (fused contacts) reports. The status reports were always generated so that we could monitor the health and activity of the vehicles. When an acoustic source was detected, contact reports were generated by the signal processing and then track reports from the tracker. At this point using a basic priority queue (before *libqueue*) we would only receive the higher priority track reports and contact reports and no status reports. This was because the number of contact and track reports exceeded the available throughput of the acoustic channel. This was unacceptable because we knew where the targets were, but no longer received updates as to the position of our AUV. Thus, we needed a way for messages with a lower base value (such as the status message) to occasionally become more valuable than those with higher base values (such as the contact and track messages). To solve this problem, *libqueue*'s dynamic priority queues, as described in section III, were created.

With *libqueue*, the messages received were proportional to the base value (time sensitivity, via the time-to-live (ttl) was introduced later). During a tracking event, track and contact messages were highest priority, but status messages were still occasionally sent.

C. SWAMSI09

SWAMSI09 was another acoustic sensing experiment, this time for sea floor mine-like targets. CCL had no messages for reporting contacts for this type of target. For this reason and the others given in section II-B1, we determined that CCL was no longer sufficient for our needs and developed DCCL and the corresponding encoding library, *libdccl*.

The ease of defining and redefining DCCL messages allows for rapid prototyping of new experimental ideas during the field trial, rather than being rigidly confined to previously defined messages. We wrote five new messages on the experiment to greatly expand the flexibility of vehicle to topside, and vehicle to vehicle communications capability.

We used two AUVs to execute a variety of bistatic acoustic configurations for tracking of proud and buried seabed targets. Both AUVs traversed a circular pattern around the potential target, maintaining a constant bistatic angle (see Fig. 10a). Entering into this collaboration and maintaining the correct angle required handshaking and data transfer between both vehicles. We were able to command the vehicles into this collaborative state with `LAMSS_DEPLOY`, and the `LAMSS_STATUS` message (with additional fields added to support this experiment) was passed between vehicles to maintain the correct positioning autonomously.

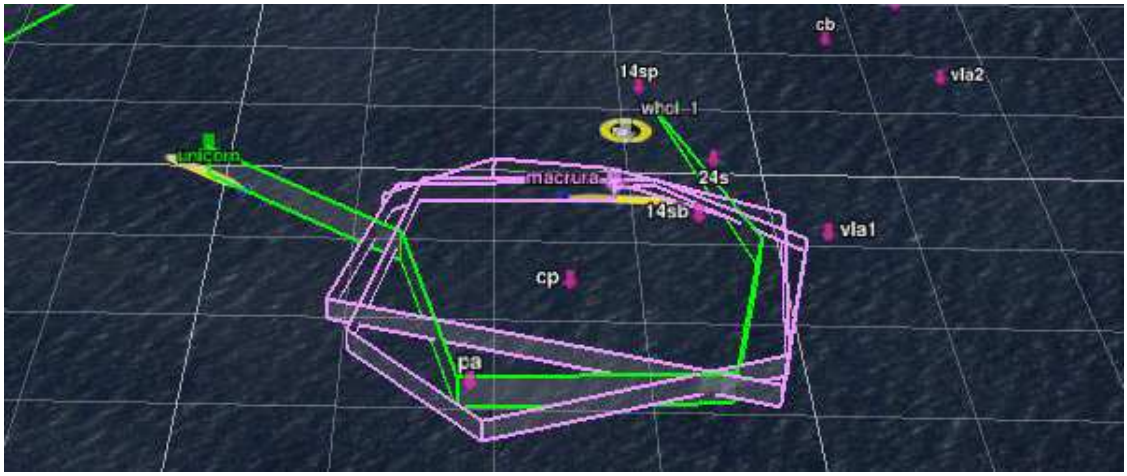
D. GLINT09

For the previous two experiments, we were using iMicro-Modem (section V-C1) as the driver for the WHOI Micro-Modem. Concerns about the robustness and extensibility of that software led to the development of *libmodemdriver*. While

TABLE V: Summary of field trials.

Name	Summary	Assets (vehicles all have WHOI Micro-Modem)	Experiment Datum[a]
GLINT08	Interoperability of marine vehicles for passive acoustic target detection	1 Bluefin 21 AUV, 1 NURC OEX AUV, 1 OceanServer Iver2 AUV, 2 Robotic Marine Kayaks, 1 WHOI Comm Buoy, 2 Ship-deployed WHOI Micro-Modems	42.5°N, 10.08333°E
SWAMSI09	Detection and tracking og seabed objects using bistatic acoustics.	2 Bluefin 21 AUVs, 1 WHOI Comm Buoy	30.045°N, 85.726°W
GLINT09	Interoperability of marine vehicles for multi-static acoustic target target tracking	1 NURC OEX AUV, 1 OceanServer Iver2 AUV, 2 Robotic Marine Kayaks, 2 Ship-deployed WHOI Micro-Modems	42.47°N, 10.9°E
CHAMPLAIN09	Thermocline gradient following.	1 OceanServer Iver2 AUV, 1 Ship-deployed WHOI Micro-Modem.	42.2511°N, 73.3612°W
GLINT10	Interoperability of marine vehicles for passive and active acoustic target tracking. Collaborative acoustic communications and environmental sampling.	1 Bluefin 21 AUV, 1 WHOI Comm Buoy, 2 NURC OEX AUVs, 2 Ship-deployed WHOI Micro-Modems.	42.47°N, 10.9°E

[a] The experiment datum is a location in the southwest corner of the operation region from which all vehicle positions are referenced using the Universal Transverse Mercator projection with the WGS 84 ellipsoid [31].



(a) During SWAMSI09, the two AUVs “Macrura” and “Unicorn” perform a synchronous circular pattern with a constant angle of separation. However, due to the sporatic updates from the acoustic modem, it is hard to visualize the performance of the vehicles in excecuting this maneuver at runtime.

(b) A snapshot of the runtime visualization of the AUV “Unicorn” performing a sinusoidal depth excursion while performing a pentagon shape. While full, updates are delayed, the `LAMSS_STATUS_FILLIN` and `LAMSS_CTD` messages give a detailed history of the vehicle’s track when the communication environment permits.

Fig. 10: Comparison of the Google Earth interface for Ocean Vehicles (GEOV) [30] visualization available to the vehicle operator during runtime using data transmitted via `goby-acomms` early in its design at SWAMSI09 (a) and in the form `goby-acomms` is presented in this paper during GLINT10 (b). Vertical lines indicate acoustic position updates via the `LAMSS_STATUS` message and horizontal lines connect these updates.

the features provided by *libmodemdriver* for the WHOI Micro-Modem did not vary much from iMicroModem, we saved significant time debugging.

Furthermore, we expanded our usage of *libdccl*. DCCL messaging made another collaborative experiment possible. We had a mobile acoustic gateway (an autonomous surface craft with a WHOI Micro-Modem) available to stream high rate environmental and other data messages. By virtue of the surface craft staying near the AUV (made possible by the AUV's `LAMSS_STATUS` message), the AUV had a short acoustic propagation path to the surface craft. From there, the surface craft relayed data to the operators via IEEE 802.11 wireless ethernet. Also, the depth of the modem was controlled by a winch that the surface vehicle could command autonomously. Using the `WINCH_CONTROL` message, the AUV commanded the surface craft a depth at which to set the modem to improve communications. The AUV was performing a bistatic acoustic detection of a mid-water column depth target. The source, mounted on a buoy, was autonomously turned on and off by the AUV using the `SOURCE_ACTIVATION` message. The AUV, which was towing an acoustic array, was the receiver. None of this multi-robot collaboration would have been possible without the ability to define new messages quickly and with a high degree of confidence in their syntactical correctness provided by *libdccl*.

E. CHAMPLAIN09

The third case study is the CHAMPLAIN09 adaptive environmental experiment. In this experiment, a small AUV outfitted with a Conductivity-Temperature-Depth (CTD) instrument was deployed to study the thermocline structure of Lake Champlain. The AUV was commanded, using a updated `LAMSS_DEPLOY` message, on the task of adaptively surveying the thermocline. The vehicle accomplished this task by performing series of sinusoidal ("yoyo") depth maneuvers and streamed its samples back using the delta-difference encoded `LAMSS_CTD` message. In this manner, the environmental data was made available in near realtime (i.e. delayed by no more than a few minutes) to the AUV operator.

The key feature used for this work and later in GLINT10 was delta-differencing, originally applied only to CTD messages and later added as a general feature to *libdccl*. Delta-difference encoding can be applied to <`float`> DCCL fields (and <`int`> since they are derived from <`float`>). It gives an even more compact way to losslessly encode correlated data. In this case, due to the AUVs finite speed and continuity of salinity and temperature values, CTD values are correlated in time. By estimating upper and lower bounds on this correlation, data can be compressed further than DCCL normally allows by sending the first sample in its entirety (still bounded by the usual DCCL <`max`> and <`min`> "global key") and sending the remaining samples in a frame by their difference to this first sample. The bound on the maximum that this difference can be (Δ_{max}) must be given in <`max_delta`> tag, using physical knowledge of the data to be sampled. See Table VI for the corresponding formulas for the field size and encoded values. Diagrammatically, the process is explained in Fig. 11 and an example of the data available to the operator during runtime is shown in Fig. 12.

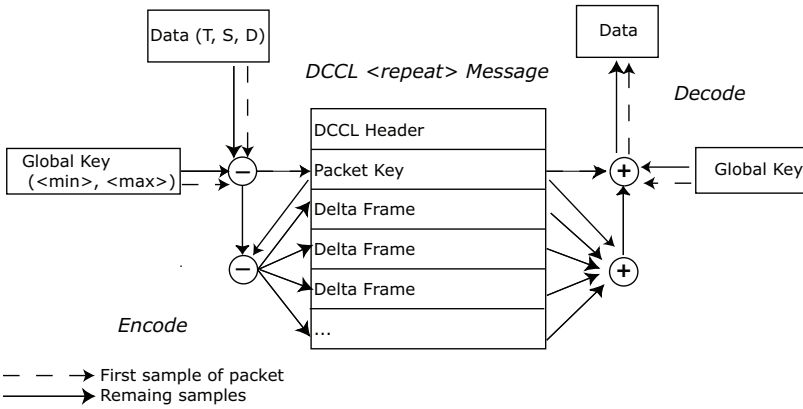


Fig. 11: Schematic of encoding and decoding a DCCL message using delta-difference encoding. The DCCL header is diagrammed in detail in Fig. 4.

For example, perhaps it is known *a priori* by means of historical data or a ship CTD cast that the maximum temperature gradient in a given area is $0.05°C/m$ and the dive rate of our glider is $0.2m/s$. We also know that the water in the operation region does not exceed $(10, 30)°C$. Furthermore, we feel that tenths of a degree Celcius is sufficient precision. Finally, we want to sample the thermistor on our CTD at 1 Hz and we are using a 256 byte WHOI Micro-Modem frame. Putting this all together, we use for temperature (in $°C$) a DCCL <`float`> with a <`min`> of 10, a <`max`> of 30, a <`precision`> of 1. The <`max_delta`> must be calculated iteratively (such as using the Newton-Raphson method), as making a smaller <`max_delta`> creates a smaller message, increasing the number of samples that can be fit in a frame. This increases the window of sampling for a given frame, thus increasing the <`max_delta`>. That is, the optimum Δ_{max} for given bounds is the solution closest to equality to

$$l_{key} + l_{delta}(\Delta_{max}) \cdot (\Delta_{max}/r_{max} * f_s - 1) <= l_{frame} \quad (4)$$

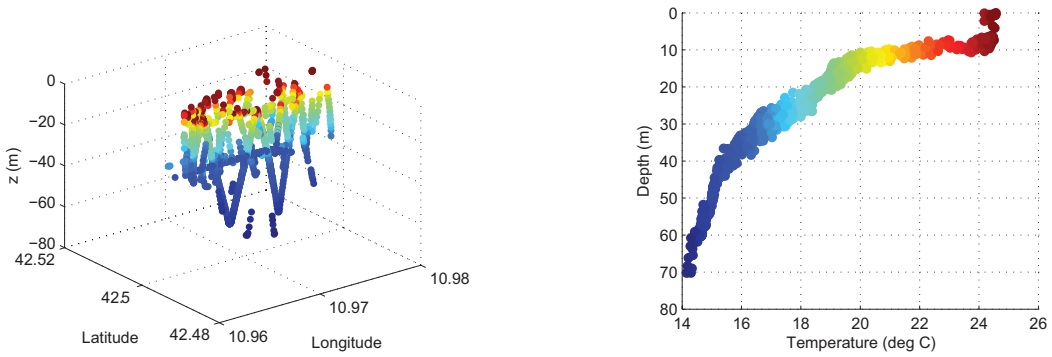
where l_{frame} is the total message frame size, l_{key} and $l_{delta}(\Delta_{max})$ are the sizes for key and delta frames given in Table VI, r_{max} is the maximum expected rate of change of the physical parameter being encoded, and f_s is the sampling frequency. For this example $l = 2048$ bits, $r_{max} = 0.05°C/m \cdot 0.2m/s = 0.01°C/s$, $f_s = 1Hz$, and $l_{key} = 8$ bits, so solving for the smallest Δ_{max} (which provides the largest number of samples in the frame) is $3.1°C$, providing 310 samples per frame. This is a 21% improvement over the 256 (l/l_{key}) samples that would fit if the message was not delta-difference encoded.

Δ_{max} is a function of the size of the frame (l_{frame}), so l_{frame} can also a parameter for optimization based on the expected maximum rate of change (r_{max}) of the physical parameters to be sent if the physical layer supports a variety of frame sizes.

TABLE VI: Formulas for delta difference encoding the DCCL <float> type.

DCCL Type	Size (bits)	Encode[a]
<float> (key)	$l_{key} = \lceil \log_2((x_{max} - x_{min}) \cdot 10^{prec} + 2) \rceil$	$x_{key} = \begin{cases} \text{nint}((x - x_{min}) \cdot 10^{prec}) + 1 & \text{if } x \in [x_{min}, xmax] \\ 0 & \text{otherwise} \end{cases}$
<float> ($delta$)	$l_{delta} = \lceil \log_2(2\Delta_{max} \cdot 10^{prec} + 2) \rceil$	$x_{\Delta} = \begin{cases} \text{nint}((x + \Delta_{max} - x_{key}) \cdot 10^{prec}) + 1 & \text{if } x - x_{key} \in [-\Delta_{max}, \Delta_{max}] \\ 0 & \text{otherwise} \end{cases}$

· x_{min}, x_{max}, $prec$, Δ_{max} are the contents of the <min>, <max>, <precision>, and <max_delta> tags, respectively.
· nint(x) means round x to the nearest integer.
· The key value x_{key} is the same as the normal <float> type encoding given in Table III.



(a) Temperature variation with depth and position

(b) Temperature-Depth profile

Fig. 12: Temperature data from the GLINT10 experiment a CTD instrument mounted on AUV Unicorn available at runtime via goby-acomms using delta-differenced encoding.

F. GLINT10

All the features and implementation of goby-acomms as presented in the rest of this paper were in place for the GLINT10 sea trial. The DCCL messages and corresponding V_{base} and ttl used for dynamic priority queuing are given in Table VII.

The key new items for GLINT10 were

- Expansion of delta-difference encoding mentioned in Section VI-E to support any arbitrary <float> field, not just those from a CTD instrument. This enabled the new "back-fill" LAMSS_STATUS_FILLIN message which keeps a history of vehicle positions regularly sampled (in this case twice per minute). These were queued with a low V_{base} of 0.4 relative to the other messages (see Table VII). Thus, in cases of low throughput due to unfavorable environmental conditions other data messages would be sent. However, when the throughput went up the queued up LAMSS_STATUS_FILLIN messages would be sent, giving the topside operators a somewhat delayed but still relevant history of the vehicle's manuevers. When developing complex adaptive autonomy, this is critical for debugging and understanding the vehicles' performance. Furthermore, all LAMSS_CTD messages were also decoded as status messages since they contain the three dimensional location of the vehicle at the time of the sample.
- The auto-discovery decentralized TDMA MAC described in section IV was tested using two vehicles, one gateway buoy and one ship deployed WHOI Micro-Modem. Due to the lack of a cycle initialization packet that could be lost, transmissions from ranges of up to 4 km were successfully made. Using the standard centralized TDMA that we have used for the previous experiments, we saw transmissions up to 2 km. It is difficult to quantify in-water performance of MAC schemes without a robust understanding of the environmental effects on propogation.

VII. Conclusion

goby-acomms provides an acoustic networking suite that combines high usability at sea with techniques intended to make the most out of the very low throughput provided by acoustic telemetry. It is comprised of four modules that could be interchanged with a suitable replacement as research advances in a particular areas:

- *libdccl*: provides encoding and decoding. The major contribution from DCCL is the ability to create custom objects that can be serialized to very short messages, with an emphasis on message size efficiency over features and abstraction.
- *libqueue*: deals with the common problem in acoustic networks of having too many messages. *libqueue* provides a way to prioritize messages based both on the time sensitivity and the overall value of the message.
- *libamac*: implements a simple TDMA scheme with auto-discovery requiring no control messages to be sent and thus not using any bandwidth that might be better used for mission data.
- *libmodemdriver*: provides an abstract interface for an acoustic (or other low-bandwidth carrier) modem and

TABLE VII: Summary of DCCL Messages used in the GLINT10 Experiment

Message Name	Category	<size> (bytes) [a]	Estimated Equivalent CCL Size (bytes) [b]	ttl [c]	V_{base} [c]	Description
LAMSS_DEPLOY	Command	31	40	300	1000	Underwater vehicle command message.
LAMSS_PROSECUTE	Command	31	40	300	1000	Underwater vehicle command message: prosecute detected target.
ACOUSTIC_MOOS_POKE	Command	32	31	300	10000	Underwater debugging / safety message.
LAMSS_STATUS	Data / Collaboration	27	35	300	1.5	Vehicle Status message (position, speed, Euler angles, autonomy state)
LAMSS_STATUS_FILLIN	Data	29	52	1800	0.4	Vehicle Status message historical "back-fill" (delta-difference encoded).
LAMSS_CONTACT	Data	29	34	600	2	Passive acoustic contact report message.
LAMSS_TRACK	Data	29	34	300	4	Passive acoustic track report message.
LAMSS_BTR	Data	64	63	7200	1	Beam-Time Record Data from a towed passive acoustic array.
LAMSS_CTD	Data	256	496	1800	1	Salinity, temperature, depth data from a CTD instrument (delta-difference encoded).

[a] For DCCL: see section II.
[b] Since CCL does not implement these messages, these size estimates are based on the closest available message in the existed CCL message set.
[c] For Priority queuing: see section III.

an implementation of this interface for the widely used WHOI Micro-Modem.

`goby-acomms` emphasizes robustness through object-oriented design to provide a communications architecture that can support real field operations with underwater robots. The hope is that `goby-acomms`, or at least some of the ideas within, can move the field of collaborative underwater robotics and artificial intelligence forward. `goby-acomms` is freely available with the Goby Underwater Autonomy Project from http://launchpad.net/goby. The Goby project is licensed under the GNU General Public License and gladly accepts contributions from members of the marine acoustic networking and robotics community.

VIII. Glossary

Some terms are subject to ambiguity due to the various disciplines (robotics, programming, acoustics, networking) this paper draws from. These terms are defined here in the context we intend them to mean in this paper:

- *class*: the schema for a an object, the term is used in the standard way for Object Oriented Programming (C++, Java, Python, etc.).
- *frame*: the smallest quantity of data that is either accepted or rejected in whole by the acoustic hardware layer's error correction. The WHOI Micro-Modem uses frames of 32, 64 and 256 bytes depending on the bit rate.
- *message*: a sequence of bytes to be transmitted over some channel. Usually refers to an instantiation of a DCCL type, as defined in section II.
- *object*: an instantiation of a class.
- *schema*: used here to refer to W3C XML Schema. Confusion here can occur since DCCL provides a schema language in its own right, with an XML schema to validate it. Thus, we choose to refer to DCCL in this paper as providing a message structure, reserving the term *schema* for the W3C XML Schema.

Acknowledgment

We thank Roger Stokey for CCL (which formed the inspiration for DCCL), Matt Grund for iMicroModem and his general ideas on acoustic networking. Also, we appreciate the help of Lee Freitag and the rest of the WHOI Acoustic Communications group with understanding and supporting the WHOI Micro-Modem. We thank the MOOS-IvP user community for their use and critiques on this project. Since `goby-acomms` would not have happened without field support, we thank all those who made the sea trials possible, from the officers and crew to the science team to the cooks. We would like to especially thank the NATO Undersea Research Centre in La Spezia, Italy, for their generous contributions of resources to help test and develop the `goby-acomms` project.

References

[1] C. Clay and H. Medwin, *Acoustical oceanography: Principles and applications*. John Wiley & Sons, Inc., 1977.
[2] A. Baggeroer, "Acoustic telemetry–An overview," *IEEE J. Ocean. Eng.*, vol. 9, no. 4, pp. 229–235, 1984.
[3] D. Kilfoyle and A. Baggeroer, "The state of the art in underwater acoustic telemetry," *IEEE J. Ocean. Eng.*, vol. 25, no. 1, pp. 4–27, 2000.
[4] J. Preisig, "Acoustic propagation considerations for underwater acoustic communications network development," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 11, no. 4, p. 210, 2007.
[5] M. Stojanovic, "Recent advances in high-speed underwater acoustic communications," *IEEE J. Ocean. Eng.*, vol. 21, no. 2, pp. 125–136, 1996.
[6] J. Partan, J. Kurose, and B. N. Levine, "A survey of practical issues in underwater networks," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 11, no. 4, pp. 23–33, 2007. [Online]. Available: http://portal.acm.org/citation.cfm?id=1347372
[7] W. Dow, "A telemetering hydrophone," *Deep Sea Research (1953)*, vol. 7, no. 2, pp. 142, IN19 – IN20, 143–146, IN21–IN22, 147, 1960. [Online]. Available: http://www.sciencedirect.com/science/article/B757G-48BCXY5-2B/2/87a7f20e94d6a78564a2d0531cd2a48f

[8] L. Freitag, M. Grund, S. Singh, J. Partan, P. Koski, and K. Ball, "The WHOI Micro-Modem: an acoustic communications and navigation system for multiple platforms," in *IEEE Oceans Conference*, 2005.
[9] A. Bahr, J. Leonard, and M. Fallon, "Cooperative localization for autonomous underwater vehicles," *The International Journal of Robotics Research*, vol. 28, no. 6, p. 714, 2009.
[10] T. Schneider, H. Schmidt, T. Pastore, and M. Benjamin, "Cooperative Autonomy for Contact Investigation," in *IEEE Oceans*, 2010.
[11] A. Shafer, "Autonomous cooperation of heterogeneous platforms for sea-based search tasks," Master's thesis, Massachusetts Institute of Technology, 2008.
[12] E. Fiorelli, N. Leonard, P. Bhatta, D. Paley, R. Bachmayer, and D. Fratantoni, "Multi-AUV control and adaptive sampling in Monterey Bay," *Oceanic Engineering, IEEE Journal of*, vol. 31, no. 4, pp. 935–948, 2007.
[13] H. Zimmermann, "OSI reference model–The ISO model of architecture for open systems interconnection," *Communications, IEEE Transactions on*, vol. 28, no. 4, pp. 425–432, 2002.
[14] G. Booch, J. Rumbaugh, and I. Jacobson, "The unified modeling language," *Unix Review*, vol. 14, no. 13, p. 5, 1996.
[15] R. P. Stokey, L. E. Freitag, and M. D. Grund, "A compact control language for AUV acoustic communication," *Oceans 2005-Europe*, vol. 2, p. 11331137, 2005.
[16] R. P. Stokey, "A compact control language for autonomous underwater vehicles," WHOI, Tech. Rep. Public Release 1.0, 2005. [Online]. Available: http://acomms.whoi.edu/ccl/
[17] M. Chitre, S. Shahabudeen, and M. Stojanovic, "Underwater acoustic communications and networking: Recent advances and future challenges," *The State of Technology in 2008*, vol. 42, no. 1, pp. 103–114, 2008.
[18] T. Welch, "Technique for high-performance data compression." *Computer*, vol. 17, no. 6, pp. 8–19, 1984.
[19] D. Huffman, "A method for the construction of minimum-redundancy codes," *Resonance*, vol. 11, no. 2, pp. 91–99, 2006.
[20] J. Larmouth, *ASN.1 Complete*. Elsevier, 2000. [Online]. Available: http://www.oss.com/asn1/larmouth.html
[21] Apache, "Xerces-C++ XML parser." [Online]. Available: http://xerces.apache.org/xerces-c/
[22] J. Daemen and V. Rijmen, "AES proposal: Rijndael," 1999. [Online]. Available: http://www.cryptosoft.de/docs/Rijndael.pdf
[23] "Secure hash signature standard," NIST, Tech. Rep. FIPS PUB 180-2, 2002. [Online]. Available: http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf
[24] W. Dai, "Crypto++ library 5.6.0." [Online]. Available: http://www.cryptopp.com/
[25] S. Webster, R. Eustice, C. Murphy, H. Singh, and L. Whitcomb, "Toward a platform-independent acoustic communications and navigation system for underwater vehicles," in *OCEANS 2009, MTS/IEEE Biloxi - Marine Technology for Our Future: Global and Local Challenges*, 2009, pp. 1–7.
[26] L. Freitag, M. Grund, C. von Alt, R. Stokey, and T. Austin, "A shallow water acoustic network for mine countermeasures operations with autonomous underwater vehicles," *Underwater Defense Technology (UDT)*, 2005.
[27] R. Eustice, L. Whitcomb, H. Singh, and M. Grund, "Experimental results in synchronous-clock one-way-travel-time acoustic navigation for autonomous underwater vehicles," in *Robotics and Automation, 2007 IEEE International Conference on*. IEEE, 2007, pp. 4257–4264.
[28] W. A. C. Group, "Micro-modem software interface guide," WHOI, Tech. Rep. 401040-SIG, 2010. [Online]. Available: http://acomms.whoi.edu/documents/uModem%20Software%20Interface%20Guide.pdf
[29] 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, November/December 2010.
[30] T. Schneider and H. Schmidt, "Unified command and control for heterogeneous marine sensing networks," *Journal of Field Robotics*.
[31] NIMA, "Department of defense world geodetic system 1984: Its definition and relationships with local geodetic systems. second edition, amendment 1," NIMA, Tech. Rep. TR8350.2, 2000. [Online]. Available: http://earth-info.nga.mil/GandG/publications/tr8350.2/wgs84fin.pdf

Toby Schneider (M'05) received a B.A. in physics at Williams College in Williamstown, MA, USA in 2007.

He is currently a graduate student working towards a Ph.D. in ocean engineering in the Joint Program in Oceanography/Applied Ocean Science and Engineering between the Massachusetts Institute of Technology in Cambridge, MA, USA and the Woods Hole Oceanographic Institution in Woods Hole, MA, USA (http://mit.whoi.edu/).

Mr. Schneider is also a member of the Acoustical Society of America. Further professional details about Mr. Schneider are available on his website: http://gobysoft.com/.

Henrik Schmidt is Professor of Mechanical & Ocean Engineering at the Massachusetts Institute of Technology. He received his MS degree from The Technical University of Denmark in 1974, and his Ph.D. from the same institution in 1978. Following a post-doctoral fellowship at the Risoe National Laboratory in Denmark, he joined the NATO Undersea Research Centre in Italy in 1982, where he worked until he joined the MIT faculty in 1987. Professor Schmidt's research has focused on underwater acoustic propagation and signal processing, and most recently on the development of environmentally adaptive acoustic sensing concepts for networks of autonomous underwater vehicles. Prof. Schmidt is a Fellow of the Acoustical Society of America, and he is the 2005 recepient of the ASA *Pioneers of Underwater Acoustics Medal*