RECORDS: a Remote Control Framework for Underwater Networks

Giovanni Toso*, Ivano Calabrese‡, Paolo Casari*‡, Michele Zorzi*‡
*Department of Information Engineering, University of Padova, via Gradenigo 6/B, I-35131 Padova, Italy
‡Consorzio Ferrara Ricerche, via Saragat 1, I-44122 Ferrara, Italy
E-mail: {tosogiov, icalabre, casarip, zorzi}@dei.unipd.it

***Abstract*—We present an open source framework that makes it possible to remotely monitor and control a heterogeneous network of underwater acoustic nodes. The framework exploits acoustic communications to deliver control messages, and thus avoids the need to deploy cabled or wireless connections to control each node. The framework has been developed with the goal to provide a ready-to-use, lightweight, robust and reliable tool for real field trials. The framework is very cheap in terms of hardware resources and is easily portable on several embedded systems because it is not necessary to recompile it. Furthermore, RECORDS embeds features developed to manage in real time network experiments by using the DESERT Underwater framework, based on the well-known ns2/NS-MIRACLE network simulator. RECORDS has been validated successfully in several laboratory experiments and sea trials involving different embedded systems arranged into several network topologies, including one major sea trial conducted in collaboration with the NATO STO CMRE.**

***Index Terms*—Underwater networks; remote control framework; Tcl; expect; embedded systems; tank tests; sea trial.**

I. Introduction and Related Work

During sea trials involving underwater acoustic networks, it is important to acquire real-time information about the status of the network, such as the signal-to-noise ratio of a given link, the residual battery level of a node, or the disk space available in an embedded system. The availability of such information is fundamental to effectively characterize the network setup, as well as to improve the design of the experiments. Gathering such information in real time during the trial makes it possible to quickly explain network events and unexpected behaviors. In addition, such information allows the operator to reconfigure the nodes and the experiments according to the status of the network. This way, the most convenient experiment can be chosen among several options prepared a priori, as a function of the current status of the deployment. This can be easily achieved whenever a reliable cabled or wireless connection to all nodes is available. However, in a large-scale scenario, providing such connections requires a significant effort, when at all feasible. Moreover, it is often impossible to access all the nodes of an underwater network after its deployment, especially in networks of large size. In fact, it may be necessary to have nodes such as buoys or Autonomous Underwater Vehicles (AUVs) run on battery power, and to leave them in the water for the longest time during a sea trial. For all reasons discussed so far, alternative means of remote control must be employed.

In this work, we present and demonstrate a robust, versatile and reliable solution to this issue, that can be employed to control the whole network whenever direct access to at least one node is available throughout the experimental campaign. The solution proposed gives the possibility to send remote commands over multihop networks, to ultimately run underwater networking experiments via the DESERT Underwater framework [1], [2].

RECORDS (short for remote control framework for underwater networks) is the evolution of the framework presented in [3] and successfully employed in two sea trials since then. Both RECORDS and the framework in [3] are written using scripting languages and feature a modular structure with loopback socket communications among different modules, but the latter has several limitations compared to RECORDS: it is based on a master/slave approach, that consequently divides the nodes in two categories and forces the user to be in control of a master node; it provides only a static routing protocol; it was designed as a monolithic solution, which is simple but less flexible than RECORDS's multilayer stack; it offers limited functionalities beyond the remote start/stop of networking experiments. In any event, the master/slave approach in [3] was successfully used in two sea trials, namely the test of the SUN routing protocol in the Werbellin lake, Berlin, in 2012 [4] in collaboration with EvoLogics GmbH [5], and the CommsNet12 sea trial campaign, conducted in the proximity of La Spezia, Italy, in collaboration with the NATO STO Centre for Maritime Research and Experimentation (CMRE). Some results obtained during the latter trial are reported in [6].

The network reprogramming function was also demonstrated during the Sea Trial 2 of the RACUN project, a multi-national EDA initiative aimed at providing robust and reliable communications in underwater networks. A specific command coded via the Generic Underwater Application Language (GUWAL) was broadcast using the GUWMANET routing protocol [7] in order to reconfigure the network layer of the nodes, including switching from a stand-alone network protocol implementation to protocols programmed in the DESERT Underwater framework [1], and vice-versa. Other commands made it possible to change modem-related parameters both locally and remotely.

A third solution to remotely configure and control the ex-

The authors gratefully thank Piero Ruol and Luca Martinelli for kindly providing access to the wave flume of the Maritime Laboratory of the Civil, Environmental and Architectural Engineering Department of the University of Padova. The purchase of the communications equipment used in this paper has been partly supported by an infrastructure fund of the Department of Information Engineering of the University of Padova. The NATO STO CMRE is gratefully acknowledged for involving the authors in the CommsNet'12 and CommsNet'13 campaigns. Many thanks also to the CMRE personnel for their support. The MSUN protocol was developed in the context of the multi-national EDA RACUN project and experimented during CommsNet'13 with the permission of the RACUN consortium.

periments in an underwater network where there is no direct access to all the nodes, called backseat driver in the following, or BD for short, has been presented in [8], but has not been publicly released so far. The BD is programmed using the SUNSET framework [9] and based on ns2/NS-MIRACLE [10]. When the BD receives a control message, it interprets the user command in order to configure and start a second instance of SUNSET, which in turn will run the proper networking experiment. The backseat driver in [8] has similar features compared to RECORDS (e.g., no prior knowledge about the network is required; unicast, multicast and broadcast messages are supported; no master/slave hierarchy is assumed). However, the two approaches are different in several ways. For example, RECORDS is written entirely using scripting languages, which are very easily ported on different embedded systems, whereas the BD requires to cross-compile and install ns2, NS-MIRACLE and SUNSET on every node, including those controlled by embedded systems; our approach makes it possible to develop and debug the software directly on the embedded system; the configuration of RECORDS can be fully or partly modified at run time via remote commands; finally, RECORDS makes it possible to interact with an attached modem via native modem commands, and can remotely execute general purpose shell commands and retrieve their output. In fact, starting networking experiments also requires no more than a shell command, and leverages on the same functionality.

Recently, the SeaLinx framework for underwater acoustic networks was presented in [11]. The framework has a layered architecture including the MAC, network, transport and application layers, which communicate via sockets. An application layer module named Acoustic Remote Control (ARC) runs permanently on the nodes and makes it possible to send remote commands acoustically, similar to what happens in RECORDS. However, [11] does not specify whether Sealinx allows the user to reconfigure the networking protocols. SeaLinx is also unavailable to the community as of today.

To the best of the authors' knowledge, the solutions [3], [7], [8], [11] discussed above, and the one presented in this paper, are the only ones specifically designed to remotely control and command underwater acoustic networks. The RECORDS software is open source, and can be freely downloaded at [12].

The rest of the paper is organized as follows: in Section II we explain the architecture of RECORDS and the functions of each module. In Section III we evaluate the framework by presenting experiments performed in our test bed and at sea. Section IV concludes the paper.

II. The RECORDS Framework

A. Framework Description

RECORDS is composed of four main modules: the remote control module, the startup module, the system profiler and a module to post-process raw log files. The framework has been written entirely using scripting languages, and in particular Tcl/Tk (both version 8.5 and 8.6 were tested), Expect (for automating interactive applications) and `sh` shell scripting. We made this choice mainly for three reasons: the straightforward

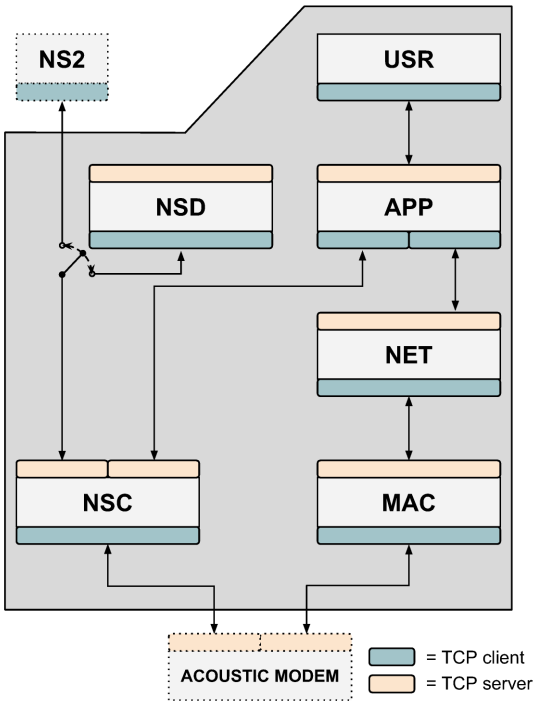

Fig. 1. The RECORDS framework: block diagram, client/server TCP connections and interactions with ns2 and the acoustic modem.

string processing capabilities of such languages; their portability (Tcl interpreters are typically available in distributions for embedded systems); the possibility to test the code directly on embedded systems, which substantially shortens development and debug time.

We proceed by describing the components of RECORDS. The **STARTUP** module can be seen as a simplified version of a *watchdog* daemon. This module reads the parameters passed by the user from the standard input, such as the IP of the modem, the socket ports and the name of the log files folder. The role of this module is crucial in the life cycle of the framework: it checks that the required network ports are available, starts other modules in the correct order, starts the system profiler and, in case any component crashes, restores it along with the other modules that depend on it.

The **PROFILER** module can be enabled for debug purposes. It keeps track, with a predetermined period, of the CPU and RAM consumption of each component of the framework, thereby allowing the user to identify memory leaks or excessively high CPU load.

The core components exploited for the remote control and command functions are graphically shown in Fig. 1, along with the TCP client/server connections opened between them. We start with the network stack modules that implement the medium access control layer (**MAC**), the network layer (**NET**) and the application layer (**APP**). Their main role is to parse, understand and deliver (either locally or remotely, or both) the commands issued by the user.

The MAC interacts directly with the modem to deliver the messages coming from the upper layers. It propagates the status messages of the modem to the upper modules (e.g.,

message sent, error, message canceled, etc.) and provides some basic multiple access interference mitigation via uniformly distributed random backoffs prior to the transmission of any message to the modem. The maximum backoff time can be set remotely by the user. For each data packet received from the modem, the MAC module queries the modem to retrieve the multipath structure of the channel and logs it.

The network layer (NET) implements two network protocols: a static source routing scheme and a flooding scheme. Each packet can be sent (in broadcast, unicast or multicast) using either protocol, and packets sent via different protocols can co-exist simultaneously in the network. The route followed using the static source routing scheme can be configured by the user, and can contain as many hops as can fit in the maximum packet size supported by the modem. The flooding scheme can also be configured by setting a time-to-live (TTL) value that limits the reach of the flooded packets. When the NET receives a packet from the APP, it sets the required fields in the header of the packet (e.g., the next hop), whereas when the packet comes from the MAC, it decides whether to deliver the packet locally to the APP or to forward it according to the protocol used for this packet. RECORDS can be easily extended with other network protocols, either by programming them in the same scripting language used for the framework, or by loading modules written with other languages, e.g., C++ or Python.

The APP module considered in this paper customizes the framework to launch networking experiments via the DESERT Underwater framework [1], although nothing forbids to write other APP modules and plug them into the framework to provide different functionalities. Our APP module keeps track of the instances of DESERT started so far and can inform the user about the status of either of them by telling whether it is running, was stopped or never started. The APP module makes it possible to remotely kill all instances of DESERT, or a subset thereof. It also acts as an abstraction layer for the modem: the user can send any message to the other nodes with no knowledge of the structure of the physical commands understood by the modem itself. It sets the environment variables required to start the DESERT executables and can run a specified experiment, with the possibility to piggyback a completely customizable list of simulation parameters. By interacting with the NET layer, it creates end-to-end acknowledgement messages to notify the user about the reception of a command and/or its output. Finally, the APP interacts with the OS to run any system command and retrieve its output (`tail`, `date` and `reboot` are relevant examples).

The ns-control (**NSC**) module is a layer that resides directly between an instance of the DESERT framework (where it effectively acts as the physical layer) and the acoustic modem. Primarily, NSC forwards messages from the DESERT software to the modem and vice-versa, and logs each message on file. Moreover, it makes it possible to simulate a desired packet error rate (PER) value over a given source-destination link, which is a very useful tool if the user wants to simulate or force a given network topologies. These PER maps are permanent and persist through subsequent instances of DESERT: however, thanks to the socket connection between the NSC and the APP

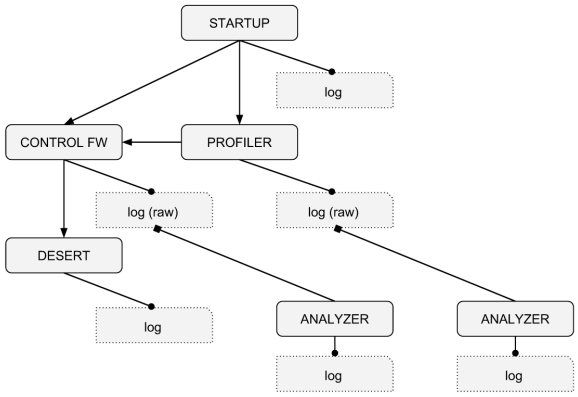

Fig. 2. Starting tree of the modules (arrows) and log files produced (points).

(see Fig. 1), the PER values can be read, set or reset in real time by the user, either locally or remotely. Finally, the NSC retrieves the channel multipath structure perceived by every DESERT packet received from the modem. All logging functionalities can be switched on or off (e.g., to save disk space).

To simulate the behavior of DESERT without actually starting it or, more in general, to provide a random traffic generator, we created an ns-dumb (**NSD**) module. This module can be started at boot, or otherwise started/stopped remotely on demand. It can be configured to send random ASCII strings to a modem; the length of the string and the transmission period can also be programmed remotely. This module can be used to test the behavior of RECORDS under different network loads in order to test the system and modem responsiveness.

The user (**USR**) module mimics the behavior of an actual user. Instead of hard-coding default values for the internal parameters of each module, USR sends them to the framework either locally, remotely, or both. Similarly, USR can query the modem by mimicking the same commands that a human user would have used if he had been connected directly to the modem. The module can ask, for example, to reset the modem, to set the acoustic ID of the modem and to set the source level. The settings can be issued at boot time, after a predetermined period, or also periodically. Among other things, these capabilities make it possible to send periodic heartbeat messages, e.g., in the form of strings.

Every RECORDS module produces a verbose and detailed log file, that can be used as meta-data when post-processing experimental results as well as for debug purposes. Log files can be individually disabled to save disk space. We provide an **ANALYZER** script that, for most modules, processes the logs and produces a human-readable output, in order to obtain statistics on the fly and in real time about the status of the framework and of the DESERT Underwater experiments. We also provide scripts that generate plots. A global overview of the interaction among the modules and the log files produced by RECORDS is reported in Fig. 2.

B. RECORDS Packet Structure

RECORDS exchanges messages among remote nodes by using packets created by the framework itself. The structure of

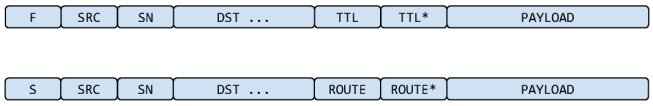

Fig. 3. Structure of the header for the flooding routing scheme (top) and for the static source routing scheme (bottom).

the packets is not fixed and the fields depend on the protocol employed by the packet. A graphical representation of the packets created by RECORDS is reported in Fig. 3.

The first four fields are the same for both protocols: a letter that identifies the protocol (F for flooding and S for static); the ID of the source (SRC) of the packet (if the packet is forwarded, the SRC does not change); the sequence number of the packet (SN), where we note that in the context of any given experiment, the pair SRC–SN uniquely identifies a packet; a list of destinations (DST). The latter field can be set as a *broadcast* address, as a single *unicast* address or as a list of unicast addresses. When a node receives a packet, the NET module checks if the ID of the current node is contained in this list. If so, it sends the packet up to the APP and removes its own ID from the list. After the deletion, if the list is not empty, the packet is forwarded according to the specified protocol; otherwise, as a general rule, the packet is dropped. Optionally, the user can choose to override this behavior so that the flooding scheme keeps retransmitting the packet until the TTL reaches zero, or so that the static source routing scheme keeps forwarding the packet along the specified route.

The fifth field depends on the protocol. In case of flooding, it represents the TTL value, i.e., the maximum number of times that the packet can be forwarded, and is initially set by the source node. Every time the NET protocol processes such a packet, it decrements the TTL value in the header and, if the value reaches zero, the forwarding procedure is stopped. The TTL* field contains a copy of the initial TTL value set by the source, and can be employed to estimate the number of times the packet has been forwarded by subtracting TTL from TTL*. This information is used to reduce the amount of overhead in the network by properly setting the TTL of acknowledgements sent in response to a received command.

The fifth field in the case of the static source routing protocol, namely ROUTE, contains the static route that the packet has to follow, and is set by the packet source. When a node receives the packet, it checks if its own ID is contained in the DST field. If so, it sends the packet to the upper layers and removes its ID from the list of destinations. If other destinations remain in the DST field and there are still hops to be covered in the ROUTE field, the node forwards the packet to the next hop in the list. A node whose ID is in the ROUTE field, but not in the DST field, forwards the packet to the next hop without sending it to the upper layers. The sixth field, ROUTE*, is the length of the ROUTE list. It is worth noting that the ROUTE field can contain the same ID more than once, because in some cases it may be useful to create loops in the path followed by a packet.

The seventh field has the same function for both protocols and it represents the payload of the packet. The payload

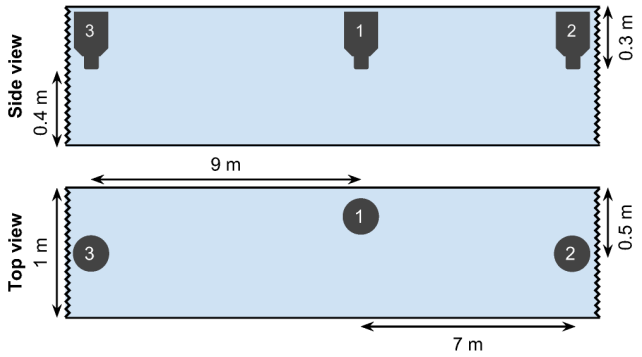

Fig. 4. Node deployment for testbed experiments in the wave flume.

is application-specific, and can be a remote command for RECORDS (e.g., send a packet remotely), a system command (e.g., battery check, node reset) or a command for the DESERT Underwater framework.

III. Software Evaluation

In this section we evaluate the performance and footprint of the RECORDS framework by means of some experiments with hardware in the loop. In particular, we start by describing the hardware setup arranged for the tests, we proceed with some controlled tank experiments in Section III-B, and conclude by describing a sea trial where we successfully employed RECORDS to remotely start experiments in a multihop network with highly asymmetric links.

A. Hardware Description

For all experiments, we employed both the S2CR White Line Science Edition (WiSE) and the S2CR 18/34 modems by EvoLogics GmbH [5]. To control the modems, we employed several different embedded systems: the IGEPv2 DM3730, by ISEE [13], the Pandaboard, by pandaboard.org [14], the Gumstix FIREstorm coupled either with the Tobi or with the Tobi-Duo expansion board, by Gumstix, Inc [15], and the Raspberry Pi Model B, by the Raspberry Pi Foundation [16].

B. Laboratory Testbed

This set of experiments aims at evaluating the RECORDS framework in the controlled environment provided by the 25-m wave flume of the Maritime Laboratory of the Civil, Environmental and Architectural Engineering Department of the University of Padova. Three S2CR WiSE modems were deployed in the tank. Each modem was connected to a different embedded system, namely one Raspberry Pi (acoustic ID 2) and Gumstix+Tobi (acoustic ID 1) and one Gumstix+Tobi-Duo (acoustic ID 3). The node with acoustic ID 1 controls the network and is positioned roughly in the middle of the tank. The other nodes are placed at opposite sides. The scenario is sketched in Fig. 4 for reference. The transducers of the nodes are placed 0.3 m below the surface, or 0.4 m from the bottom of the tank. The distance between nodes 2 and 1 is 7 m, whereas the distance between nodes 1 and 3 is 9 m. We remark that the Raspberry Pi and the Gumstix have very

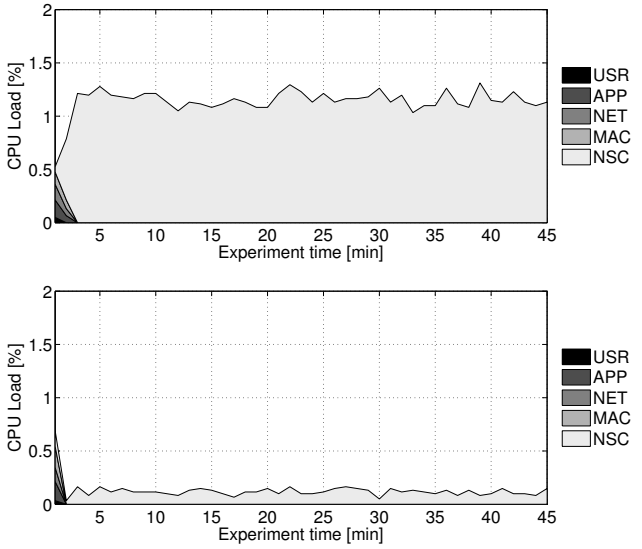

Fig. 5. CPU Load [%] for the Gumstix in the presence of heavy traffic with packet generation period 2 s (top) and light traffic with packet generation period 20 s (bottom).

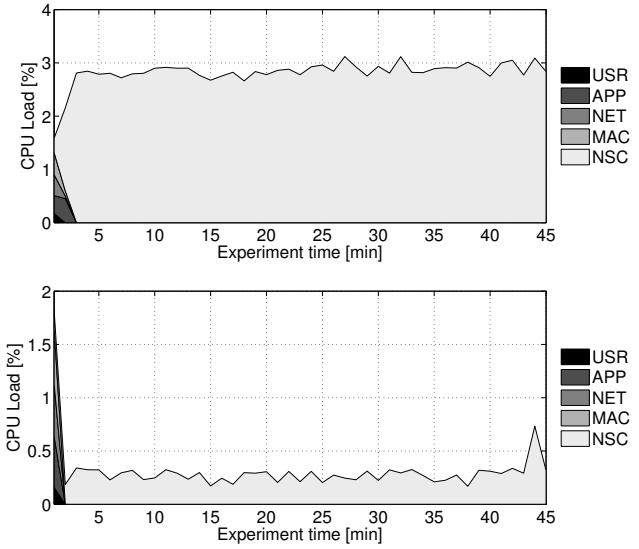

Fig. 6. CPU Load [%] for the Raspberry Pi in the presence of heavy traffic with packet generation period 2 s (top) and light traffic with packet generation period 20 s (bottom). Notice that the y-axis reaches up to 4% in the top panel.

different hardware features and prices.[1] With the following experiment, we prove that RECORDS is lightweight and does not stress the CPU and the RAM of the system; moreover, it behaves similarly on either embedded board despite their hardware differences. For a comparison of the behavior of embedded systems with the DESERT Underwater framework, we refer the interested reader to [17].

Test 1: CPU and RAM usage when running network experiments. In the first experiment we focus on measuring RECORDS's requirements of CPU and RAM on the embedded systems. We carry out the comparison under two realistic work load conditions, reproducing either a heavy or a light traffic, as could be experienced in a real underwater network, depending on the application to be supported. In the heavy traffic configuration, node 1 generates 1 data packet every 2 s with a random payload of 30 bytes, and sends each packet in broadcast. The packets are generated via the NSD module. Nodes 2 and 3 receive the data packets, log them and retrieve the channel multipath structure from the modem. In the low traffic condition, the generation period is relaxed down to 20 s. In the plots we evaluate the CPU and RAM usage for a total duration of 45 min from when the NSD module starts generating traffic. We reset both the modems and the embedded systems before starting each experiment. The CPU and RAM usage is sampled every 1 s, and each value reported in the plots is computed as the mean over one minute.

The plots for the CPU load (in % of the CPU time), for both traffic generation rates, are reported in Fig. 5 for the Gumstix, and in Fig. 6 for the Raspberry Pi. We observe that only the NSC loads the CPU noticeably, whereas the other modules are mostly idle. This is easily explained by recalling that, after

[1]The Gumstix is equipped with a TI OMAP3730 Cortex-A8 Core 1 GHz whereas the Raspberry Pi with a ARM1176JZF-S 700 MHz. The price of the a Gumstix FIREstorm with a Tobi expansion board is roughly 6 times the cost of a Raspberry Pi Model B.

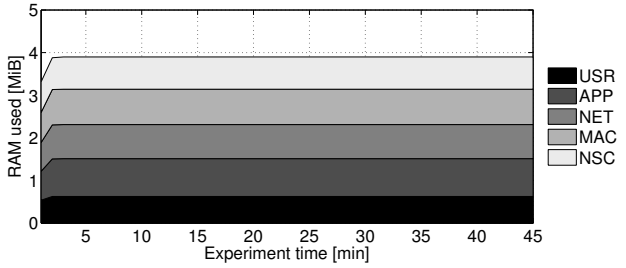

Fig. 7. Amount of RAM used [MiB] for the Raspberry Pi in the presence of heavy traffic with packet generation period 2 s.

the startup phase (when the control framework sends remote commands), the only module that actually processes packets is the NSC. In particular, it reads messages coming from the upper layers and forwards them to the modem, and vice-versa, and logs the multipath structure for each received packet. In the heavy traffic experiments, the overall CPU load is on the order of 1.2% for the Gumstix and 2.8% for the Raspberry Pi. Under light traffic, this value decreases to about 0.2% for the former and 0.4% for the latter. As expected, due to the fact that the Gumstix is equipped with a faster CPU, the CPU load is lower compared to the Raspberry Pi. Nevertheless, the global CPU consumption in the heavy load condition stabilizes to low values for both embedded systems. From the previous analysis we can state that, when the framework is processing and analyzing packets from the *ns2/DESERT* framework, its impact is very low on the CPU, and the impact does not change dramatically depending on the embedded system, thus confirming that it is lightweight.

The plots for the amount of RAM space used (in MiB) is entirely analogous for all experiments. For this reason, we report only the worst case of the Raspberry Pi in the heavy load scenario, shown in Fig. 7. The differences among different modules can be attributed to the number of TCP sockets opened by each module. For example, Fig. 1 shows that APP

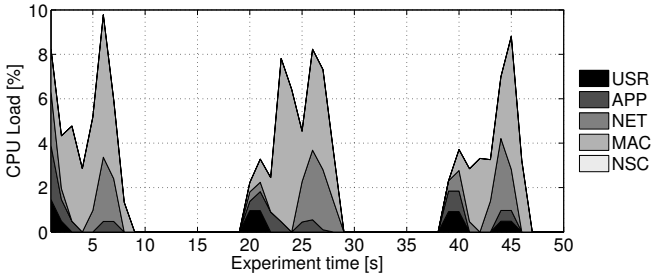

Fig. 8. CPU load for the Raspberry Pi during the static routing experiment.

allocates the largest amount of RAM (3 sockets), whereas USR requires the lowest amount, as it opens only 1 socket. The overall amount of RAM required by RECORDS sums to 4 MiB for the Raspberry Pi and to 3.5 MiB for the Gumstix, approximately 0.75% of the total amount of RAM available on either system. We remark that the amount of RAM used by RECORDS was observed to be stable throughout the entire test, a good sanity check that suggests that the framework is free from memory leaks. As a side note, Tcl 8.5 loaded on the Gumstix board seems to lead to a slightly smaller memory allocation with respect to Tcl 8.6 loaded on the Raspberry Pi. From these tests, we can conclude that the CPU and RAM footprint of RECORDS is very contained, and that it scales well from light to intense communication rates.

Test 2: CPU and RAM usage when employing internal RECORDS networking capabilities. While in the previous analysis we focused on the performance of the framework itself, in the second experiment we evaluate the CPU and RAM usage of RECORDS when its networking functionalities are exploited to route remote commands through a multihop network. This makes it possible to evaluate the behavior of all modules (not only the NSC as in the previous experiment). The setup is the same of the previous experiment, but in this case we configured node 2 to send strings to node 3 by using node 1 as a relay. Upon the reception of a message from node 2, node 3 floods back an acknowledgment message. The USR module in node 2 was configured to send the message `HELLO_I_AM_NODE_2` every 20 s to the APP module:

```
SEND,S,3,1 3,HELLO_I_AM_NODE_2
```

where `S` instructs the framework to create a static source routing message with node `3` as the final destination, with list of hops `1 3`. In this experiment, node 2 (the source) must carry out the largest number of operations. To maximize the stress to which the node is subject, we equipped it with the more performance-constrained Raspberry Pi board, and will focus our analysis on this platform. The RAM usage for this experiment is the same as in the previous experiment, and is not reported for brevity. Fig. 8 reports a plot of the impact of RECORDS on the load of the Raspberry Pi CPU across a windows of 3 data packet transmissions, where we sample the CPU load once every 1 s. For each packet, the initial peak is due to the generation of the packet itself, to the translation of its contents into an accepted modem command, and to the communication with the modem. A second peak is due to the reception of the acknowledgment from node 3. Notice

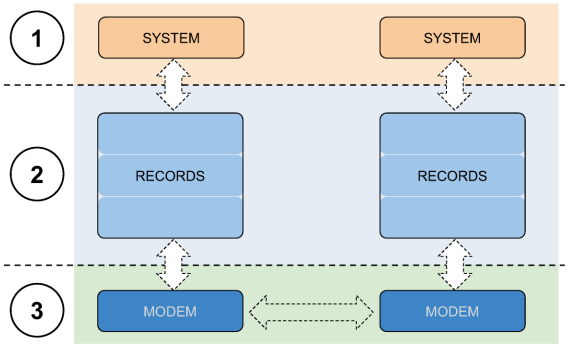

Fig. 9. Components involved in the measurement of the delay to execute remote commands.

that this happens roughly 5 s after the packet transmission. The non-zero CPU use tail between these two peaks is due to the overhearing of forwarded packets, that are not meant for node 2, and are therefore discarded by the MAC or the NET layer. As opposed to the previous experiment, in this one the NSC module does not load the CPU, as we do not send traffic through the NSC.

Test 3: delivery and execution latency for remote commands. In the third tank experiment, we evaluate the time required to execute remote commands. This time depends on several factors like the propagation delay, the processing time within RECORDS, and the time required by the modem to modulate and transmit the packet. We employed the same topology as in Fig. 4 where node 2 (equipped again with a Raspberry Pi) sends commands to node 3 (equipped with a Gumstix). We synchronized the node clocks in advance to within a few ms via the Network Time Protocol (NTP), in order to allow more precise time measurements. We consider three types of remote commands: those that do not require an acknowledgment, those that require an acknowledgment, and those specifically employed to start the DESERT Underwater framework. The last one is especially interesting both because it requires several seconds to execute and because it finds practical use in underwater networking experiments. The distance between nodes 2 and 3 is about 16 meters, which translates into a propagation delay of about 10 ms by assuming a sound propagation speed of 1500 m/s. The processing time required by the modems is undeclared by the manufacturer, so we estimated it by subtracting the processing delays of the components that we could measure from the overall delay measured.

Fig. 9 provides a graphical representation of the components involved in the measurement of the execution time of remote commands, namely (1) the system time required to process the command, (2) the time required by RECORDS to process, parse and forward the command to the modem, (3) the time required by the modem to deliver the message via the acoustic channel.

In the following, we compute each reported value as the average of 10 separate measurements, where every value encompasses the three components listed above. The first is used to remotely reboot a node, and does not require acknowledgments. The command issued by the user is:

```
SEND,F,2,1,SYSTEM 0 reboot
```

where: SEND means that we want to send a remote command, F means that we request the use of the flooding protocol, 2 is the ID of the destination and 1 is the TTL. SYSTEM 0 reboot is the remote command: in particular, SYSTEM means that RECORDS has to execute the command as a system command at the receiver side, 0 is the execution delay, and reboot is the command to be executed. The time required by RECORDS to read, interpret, deliver and remotely execute the user command was 1.33 s, where 0.73 s are due to RECORDS itself, a negligible part is due to the processing time and 0.60 s are due to the acoustic communication chain.

The second command we consider is used to retrieve the list of files contained in a remote folder. The command from the user is:

```
SEND,F,2,1,SYSTEM 0 ls
```

The structure of this command is the same as in the previous case, except that now we want to remotely run the ls system command. The overall time required is 3.75 s, where 1.60 s are due to RECORDS, 0.54 s is the system processing time, and 1.61 s are required by the acoustic communication chain.

The third command we consider starts the DESERT Underwater framework remotely. The command from the user is:

```
SEND,F,2,1,NS 02 M 35 120 3600 3 1 0 10 30 20\
           7 8 0 2
```

where the first part of the command is the same as before, and NS is a special tag used to instruct RECORDS to start DESERT. 02 is the ID that uniquely identifies the experiment being started, and the remaining input parameters are used to configure the experiment. The time required by this command is 4.76 s, where 1.67 s are required by RECORDS, 1.48 s are the system processing time, and 1.61 s are required by the acoustic communication chain. The amounts of time required by RECORDS and by the communication chain are comparable across all experiments presented: this determines that the behavior of RECORDS and of the modem is stable under several conditions, as suggested in Section III-A. In the presence of a larger distance among the nodes, the measured values would increase according to the longer propagation delay. The main difference among the three tested commands, except that two of them required an acknowledgment, lies mainly in the time required to execute the remote command, especially the one related to DESERT. We also measured the time required by the latter special command in both embedded platforms. The mean value is 1.48 s for the Raspberry Pi and 0.75 s for the Gumstix. This is due in part to the faster CPU of the Gumstix, and in part to the different SD cards used by the systems: a Class 2 for the Raspberry Pi and a Class 6 for the Gumstix.

C. Use of RECORDS during the CommsNet'13 Sea Trial

In the period Sep 9–22, 2013, we participated to the CommsNet'13 sea trials in La Spezia, Italy, in the context of a collaboration with the NATO STO CMRE. The purpose of our trial module was, among others, to test and compare four network protocols developed for the DESERT Underwater framework. The network employed for the trial was composed by several types of nodes, including bottom nodes, mobile nodes and floating nodes. Since not all the nodes were reachable via a cable or radio link, we could only reconfigure them and check their status via the RECORDS framework. In particular, we successfully used RECORDS to start, manage and stop more than 30 network experiments. As an example of real-world application of RECORDS, we will now discuss one of the CommsNet'13 experiments (the configuration parameters for this experiment and some measured performance metrics are given in Table I). In this analysis we want to demonstrate how we configured the network experiment remotely, how the configuration message propagated, as well as some numerical results such as the time required to configure each node. An overview of the deployment is depicted in Fig. 10. Compared to the nodes reported in the figure, the Wave Glider was not available, and we assumed to have direct access only to node M2.[2]

Fig. 10. A comprehensive snapshot of the network deployment during the CommsNet'13 sea trials in La Spezia, Italy. The yellow pins delimit the operational area. The network includes bottom-mounted nodes, a ship node, an acoustic/radio gateway buoy, two autonomous underwater vehicles and one autonomous surface system.

Considering that the position of our node was quite central in the deployment, we decided to configure the nodes by using the flooding protocol. The command sent via M2 was:

```
SEND,F,255,4,NS 53 M 60 240 36000 4 1 0 15 48\
           120 2 7 8 3
```

With this command, the modem broadcasts a configuration message with TTL 4, in order to instruct the entire network to start DESERT Underwater to experiment a specific routing protocol named MSUN (whence the M), with experiment ID 53. The nodes were also instructed about the duration of the experiment, 36000 s, about which ones are packet sources (IDs 2, 7 and 8) and which are sinks (node 3), and are provided with a list of parameters internal to the protocol. In Fig. 11 we report the paths followed by the configuration messages before reaching all the nodes. The IDs of the nodes in this image are the acoustic IDs associated to the modems. Compared to the labels in Fig. 10 the association is: M1→1, M2→2, M3→3,

[2]In fact, nodes M1 through M4 and the gateway buoy were directly accessible via radio links. However, we decided to fully exploit RECORDS by not using the radio link option.

TABLE I
CONFIGURATION PARAMETERS AND SAMPLE RESULTS OF THE ROUTING EXPERIMENT PERFORMED DURING THE COMMSNET'13 TRIALS.

Packet gen. rate per node	1 pkt/min
Source node IDs	2, 7, 8
Sink node ID	3
Actual experiment duration	1445 s
Total number of bits sent	115294
APP-layer PDR	0.98
NET-layer PDR	0.98
Overhead	662 bit/pkt
Throughput	2.35 pkt/min

M4→4, Gateway Buoy→5, Wave Glider→6, Ship Node→7, first Folaga AUV→8, second Folaga AUV→9. The message, sent by node 2, reached nodes 1, 3, 5, 7 and 8 directly. The framework is configured so that the source node does not configure itself directly upon sending the message, but only if it receives a forwarded copy of it. The other nodes received a forwarded copy of the message: node 4 received the message from node 3, and nodes 2 and 9 received a copy forwarded by node 5 (the latter despite node 2 was geographically closer).

As reported in Section III-B, the time required to start a command depends on several factors. From the log files we can infer that the reception of the configuration message required roughly 3 seconds for nodes 1, 3 and 5, about 4 s for nodes 7 and 8, and 6 s for nodes 2, 4 and 9 (which received a forwarded copy of the configuration message). Notably, the application layer ACK from node 4 to node 2 was received directly, and not through the 4→3→2 route. This is one among many cases where asymmetric links were experienced, and shows that RECORDS can operate correctly also in this condition. The initial reconfiguration message was sent by node 2, and forwarded 8 times (up to once per node), whereas 15 configuration messages (including replicas) were received in the network. For each unique configuration message received, the nodes sent an acknowledgment message to node 2. All such messages were received.

IV. CONCLUSIONS

In this paper, we presented RECORDS, an open source framework to remotely control underwater modems via acoustic messages. RECORDS is modular and programmed using scripting languages for easier portability. We conducted several field experiments to test the framework, both alone ad along with several network protocols programmed using DESERT Underwater. The results show that RECORDS is a stable, lightweight and robust solution to control underwater networks. The performed testbed and real-world experiments allowed us to measure both the impact of RECORDS on the system resources of different embedded platforms and the latency before commands are actually executed on a remote node. The results confirm that RECORDS can be employed extensively in real world experiments.

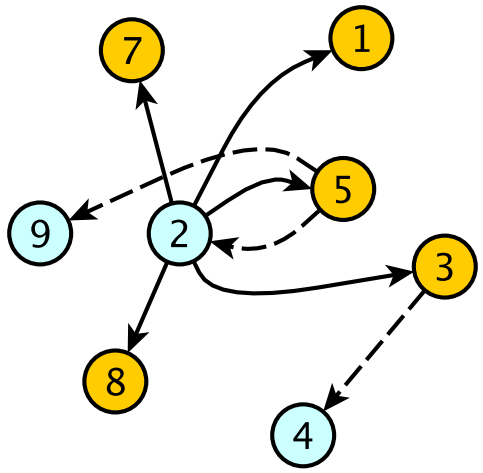

Fig. 11. Configuration message broadcast to start experiment ID 53 during the CommsNet'13 sea trials. The arrows depict the path followed by the message: solid arrows indicate direct reception from the original sender (node 2); dashed arrows indicate the reception of a forwarded message.

REFERENCES

[1] R. Masiero, S. Azad, F. Favaro, M. Petrani, G. Toso, F. Guerra, P. Casari, and M. Zorzi, "DESERT Underwater: an NSMiracle-based framework to DEsign, Simulate, Emulate and Realize Test-beds for Underwater network protocols," in *Proc. IEEE/OES OCEANS*, Yeosu, Korea, Jun. 2012.

[2] R. Masiero, P. Casari, and M. Zorzi, "The NAUTILUS project: Physical parameters, architectures and network scenarios," in *Proc. MTS/IEEE OCEANS*, Kona, HI, Sep. 2011.

[3] R. Masiero, G. Toso, P. Casari, O. Kebkal, M. Komar, and M. Zorzi, "A master/slave approach to command acoustic modems during underwater networking sea trials," in *Demo presented at ACM WUWNet*, Los Angeles, CA, Nov. 2012.

[4] G. Toso, R. Masiero, P. Casari, O. Kebkal, M. Komar, and M. Zorzi, "Field experiments for dynamic source routing: S2C EvoLogics modems run the SUN protocol using the DESERT Underwater libraries," in *Proc. MTS/IEEE OCEANS*, Hampton Roads, VA, Sep. 2012.

[5] "Evologics," Last time accessed: February 2014. [Online]. Available: http://www.evologics.de/

[6] B. Tomasi, G. Toso, P. Casari, and M. Zorzi, "Impact of time-varying underwater acoustic channels on the performance of routing protocols," *IEEE J. Ocean. Eng.*, vol. 38, no. 4, pp. 772–784, Oct. 2013.

[7] M. Goetz and I. Nissen, *GUWMANET–Multicast Routing in Underwater Acoustic Networks*. Warsaw, Poland: Military University of Technology, 2012, vol. 2, ch. 5, pp. 27–44. [Online]. Available: www.wil.waw.pl/art_prac/2012/MCC_vol2.pdf

[8] R. Petroccia and D. Spaccini, "A back-seat driver for remote control of experiments in underwater acoustic sensor networks," in *Proc. MTS/IEEE OCEANS*, Bergen, Norway, Jun. 2013.

[9] C. Petrioli, R. Petroccia, and D. Spaccini, "Sunset version 2.0: Enhanced framework for simulation, emulation and real-life testing of underwater wireless sensor networks," in *Proc. ACM WUWNet*, Kaohsiung, Taiwan, Nov. 2013. [Online]. Available: http://reti.dsi.uniroma1.it/UWSN_Group/index.php?page=sunset

[10] N. Baldo, M. Miozzo, F. Guerra, M. Rossi, and M. Zorzi, "MIRACLE: The Multi-Interface Cross-Layer Extension of ns2," *EURASIP Journal on Wireless Communications and Networking*, Jan. 2010. [Online]. Available: http://www.hindawi.com/journals/wcn/2010/761792/cta/

[11] S. N. Le, Z. Peng, J.-H. Cui, H. Zhou, and J. Liao, "SeaLinx: A multi-instance protocol stack architecture for underwater networking," in *Proc. ACM WUWNet*, Kaohsiung, Taiwan, Nov. 2013.

[12] "RECORDS source code," Last time accessed: February 2014. [Online]. Available: https://github.com/uwsignet/records

[13] "IGEPv2 DM3730," Last time accessed: February 2014. [Online]. Available: https://www.isee.biz/store/product/76-igepv2-dm3730/

[14] "PandaBoard REV A4," Last time accessed: February 2014. [Online]. Available: http://pandaboard.org/content/resources/references

[15] "Gumstix FIREstorm," Last time accessed: February 2014. [Online]. Available: https://store.gumstix.com/index.php/products/267/

[16] "RaspberryPi," Last time accessed: February 2014. [Online]. Available: http://www.raspberrypi.org/

[17] I. Calabrese, R. Masiero, P. Casari, L. Vangelista, and M. Zorzi, "Embedded systems for prototyping underwater acoustic networks: The DESERT Underwater libraries on board the PandaBoard and NetDCU," in *Proc. MTS/IEEE OCEANS*, Hampton Roads, VA, Oct. 2012.