LIGHTWEIGHT AND FRAMEWORK-INDEPENDENT COMMUNICATION LIBRARY TO SUPPORT CROSS-PLATFORM ROBOTIC APPLICATIONS AND HIGH-LATENCY CONNECTIONS

Virtual Conference 19–23 October 2020

Leon Danter[1], Steffen Planthaber[1], Alexander Dettmann[1], Wiebke Brinkmann[1], Frank Kirchner[1,2]

[1]Robotics Innovation Center (RIC), German Research Center for Artificial Intelligence (DFKI), Robert-Hooke-Str. 1, 28359 Bremen, Germany, E-mail: forename.surname@dfki.de
[2]Robotics Lab, University of Bremen, Bremen, Germany

ABSTRACT

Modular design of software stacks and hardware components is indispensable for efficient, robust and scalable projects in many fields. Modularity enables independent parallel development and facilitates large cooperative projects. On the downside communication is slowed down extensively, if a number of different frameworks is of use. To overcome this drawback, the paper at hand introduces a lightweight communication library, specifically designed for robotic applications. It is framework and OS independent and intended to provide a unified communication layer. While being generic, the library is also flexible, offering individual channels for reliable command transfer and non-blocking telemetry data. The provided communication sockets are configurable to use any implemented transport protocol, while not being limited to ethernet based ones. By default the transport is using the networking library ZeroMQ[1] (ZMQ), which provides various patterns, e.g. publish/subscribe, and supports many-to-many connections. An additional protocol implementation, UDT[2], is offered for high-latency connections. Conducted benchmark tests demonstrate that the library, especially the UDT transport implementation, provides a faster and more reliable middleware than what is currently used in native ROS[3]. This is particularly the case for network setups with an emulated delay (2 s) or package loss (20%) and large chunks of telemetry data, which emphasizes the suitability of the library for space applications and high-latency scenarios.

1 INTRODUCTION

Most robotic systems implement their control by means of Robot Control and Operating Systems (RCOS), e.g. ROS or ROCK [1]. They provide standards for modeling and implementing control architectures by defining building blocks, interfaces and communication channels. While this approach facilitates and speeds up the development and implementation process, it is bound to limitations in multi-robot cooperation scenarios. Most notably when a variety of project partners is involved, a diversity of RCOS may be of use. Consequently, the communication between involved software stacks is complicated and the often time-consuming type conversion at many communication nodes can result in a significant setback regarding latency and update frequency, which already pose great challenges in space scenarios. Hence it is advantageous to use a common, lightweight and configurable communication library, that transforms the RCOS specific communication messages into a mutual communication protocol. This approach supersedes the need for customized framework-dependent solutions for each connection and enables all components to effortlessly interact with each other, while being adaptive to task specific constrains like high-latency.

The introduced Robot Remote Control library[4] (RRC) is an open-source solution that is specifically designed for robotic multi-agent applications. It offers a fast and reliable, framework-independent interface to connect multiple instances through interchangeable transport protocols. Individual channels for telemetry data and commands enhance the ability to tailor the communication channels with different parameterizations or even completely independent protocols. This allows for reliable transfer of commands, while at the same time offering fast and non-blocking telemetry updates. In general the message transport is implemented as an open interface, where the default can easily be replaced with other protocols, while not being limited to ethernet based implementations. By default the implemented transport layer is based on the high-level networking library ZeroMQ (ZMQ) and Google's language- and plat-

1 zeromq.org
2 udt.sourceforge.io
3 ros.org
4 github.com/dfki-ric/robot_remote_control

form-neutral protocol buffers protobuf[5] for serialization. With special regard to space applications, which by nature have to deal with high signal delays, a reliable UDT based point-to-point protocol option is implemented in order to support high-latency connections.

In this paper, section 2 elaborates on the concept and implementation of the library. Section 3 describes and discusses setup and results of benchmark tests performed with the library in comparison to ROS. Section 4 concludes the paper and briefly depicts a number of use-cases.

2 METHOD AND IMPLEMENTATION

2.1 Concept and Class Structure

The introduced Robot Remote Control library is a lightweight, pure C++ library compiled with CMake. Only OS dependencies are the protobuf, protobuf-compiler and ZMQ libraries, which run on most OS systems and support a broad scope of languages.

In order to communicate using the presented library, two entities have to exist. On the one hand the ControlledRobot class (CR), which is receiving commands and sharing telemetry updates. On the other hand the RobotController class (RC), which is sending commands and receiving telemetry data. While the chosen names might seem limiting, each robot or network node can define an arbitrary amount of both available objects and send as well as receive commands or telemetry to as many entities as required. Thereby commands are getting acknowledged upon retrieval. The receipt of telemetry data on the other hand is not confirmed by default. Both classes are compiled into individual shared libraries during the build process.

Fundamental to these two classes is the transport implementation, which by default uses the ZMQ networking library. During transport initialization the IP address and port number for connecting the sockets are defined. The generated transports are handed to the CR and RC in their constructor. Thus offering endless opportunities to configure the command and telemetry channels to best accommodate the needed requirements for any setup.

Both classes, CR and RC, implement the abstract update function from their UpdateThread base class, which handles the periodic call of the update method in its own thread. Both, commands and telemetry buffers are immediately sent via the respective communication sockets of one class object and are evaluated and processed in the update method of the other class. In an effort to make the last telemetry message available for reliable requests over the command transport channel, the most recent telemetry entry is also stored in a buffer on the CR side.

The library provides a rich set of message types including e.g. Twist, JointState and Wrench. Additional types can easily be defined in the proto3 formatted RobotRemoteControl.proto[6] file. However a more enduring way to extend the libraries functions and message types is offered via inheritance. The library can act as a base library and enables encapsulation of additional features and types in form of another shared library (as shown in the examples on github[7]). This approach allows to comfortably accommodate task specific requirements, while profiting from a seamless integration of developments and updates from the main library. Besides the extended library remains compatible with the base one.

2.2 Message Types

The table below (Tab. 1) lists all currently implemented command and telemetry types. At this point it shall be emphasized that the presented set serves as a generic foundation and can easily be extended with the methods described in the previous sub-section.

Table 1: Available command and telemetry types

Commands	Telemetry
Pose	Pose
JointState	JointState
GoTo	WrenchState
Twist	SimpleSensor
SimpleAction	RobotState
ComplexAction	Map
LogLevel	LogMessage

2.3 Serialization

The afore mentioned .proto file contains the basic message description with a straight forward syntax:

```
message Position {
  double x = 1;
  double y = 2;
  double z = 3;
}
```

5 developers.google.com/protocol-buffers

6 github.com/dfki-ric/robot_remote_control/blob/master/src/Types/RobotRemoteControl.proto

7 github.com/dfki-ric/robot_remote_control/tree/master/examples/extending

Making use of the protobuf compiler library protoc[8], this file is processed into a header and implementation file, handling de-/serialization of all described message types. The generation of these files is triggered by Cmake during the build process of the library.

2.4 Transport

By default the libraries transport layer is using ZMQ. ZMQ is an open-source, high-performance, asyncronous messaging library, which can run without a message broker. It offers a number of messaging patterns like publish/subscribe, request/reply or fan-out/fan-in. Thus offering reliable and non-blocking message transfer as well as the opportunity for many-to-many connections.

As an alternative transport a UDT implementation is included as part of the library. UDT is a reliable UDP [2] based transport protocol, with its own reliability and congestion control algorithm. It establishes a point-to-point connection, is highly configurable and specifically designed for data intensive applications. Furthermore UDT is able to use unlimited bandwidth at least within terrestrial areas [3].

Next to the implemented transport option, any transport, even satellite or radio communication can be added to the library, as long as it provides reliable send() and receive() functions and implements the Transport.hpp interface.

3 PERFORMANCE TESTS

3.1 Experimental Setup

In order to benchmark the library, an experiment is set up to classify its performance in comparison to the commonly used ROS framework. In an effort to minimize disturbances due to general network traffic an independent local wireless network is setup using an Alice Modem WLAN 1421 fabricated by Arcadyan. Two hosts, Dell Inspiron 15 7000 running Ubuntu 18.04, and Samsung R519 running a docker image of Ubuntu 18.04, are connected to the network and each launch an executable in the scope of the experiment. The two testing entities can act as ROS nodes using the Melodic Morenia distribution, but they can also provide independent communication sockets using the presented library. The executables are run as sender or repeater, while the latter is returning the received message (telemetry) or an acknowledgment (commands) back to the sender. Thus the round trip or transmission time can be measured reliably without error-prone time synchronization of the host systems.

Before each experiment the network properties are analyzed as described in 3.2. For each test the round trip time of telemetry data or transmission time of commands is measured over 200 consecutive runs, while the final results are composed by three complete tests, conducted on different days. Communication over the telemetry channel is evaluated by sending a random strings of defined byte size forth and back using topics (ROS) or non-blocking telemetry transport (RRC). Commands, in the form of a random twist message, are send from one executable and receive a uint16 formatted acknowledgment in return. Per default the command transfer implemented by the RRC library gets acknowledged with an uint16 upon retrieval. For ROS this reliability feature is induced as a service call with a twist command as request and an uint16 as response. While UDT communication is always point-to-point, the ZMQ transport is configured to transfer telemetry data via publish/subscribe and commands using the request/reply pattern.

With special regard to space applications in orbital and planetary environments, which by nature have to deal with high signal delays, an additional round of tests is conducted with an artificial delay. The delay is emulated on the senders hosts network interface using netem[9]. This allows to provide a reproducible high-latency scenario. Netem is an enhancement of the linux traffic control facilities and is configured to replicate a fixed delay of 2000 ms. Furthermore it is used to emulate a package loss of 20 % for another set of command transfer time measurements.

3.2 Network properties

To ensure reproducibility the local wireless network is characterized before each conducted test. The network latency and packet loss is measured through an ICMP echo request using the commonly known ping[10] command over 100 consecutive runs. In order to quantify the available tcp bandwidth and latency variation (jitter) the iperf3[11] tool is used. The following table (Tab. 2) lists the mean network properties for each network setting.

8 manpages.ubuntu.com/manpages/focal/en/man1/protoc

9 manpages.ubuntu.com/manpages/bionic/man8/tc-netem

10 manpages.ubuntu.com/manpages/cosmic/man8/ping

11 manpages.ubuntu.com/manpages/focal/en/man1/iperf3

Table 2: Average network properties of the three network configurations

	Default	package loss (20 %)	Delay (2000 ms)
Latency (ms)	9.6	13.1	2012.5
Bandwidth (Mbits/sec)	10.45	0.78	3.35
Jitter (ms)	3.26	25.8	6.4
Loss (%)	0.0	20.8	1.6

3.3 Results

Since the wireless network that was used for testing is a local router the following section will focus on qualitatively comparing round trip and transmission times of the different communication implementations. Since establishing a connection does require some additional time, all initial time measurement values are sliced off the plotted data to enhance visibility.

3.3.1 Command Transfer

The following figure (Figure 1) shows boxplots for the transmission time of randomly generated twist commands averaged over three experiments with 200 runs each. Over the y-axis the ROS communication, using a ROS service call, is compared to two transport implementations of the RRC library, namely ZMQ and UDT.

Over the course of all network configurations (default, package loss and delay) ZMQ consistently achieves the fastest command transfer. UDT clocks in at around twice the speed for all runs without an emulated delay. With the two seconds of latency, the comparably small transfer time of 4-8 x10^{-3} s for ZMQ and UDT respectively is overshadowed. This makes ZMQ and UDT almost indistinguishable in transfer time for the network configuration with artificial delay (bottom row). Using the same network configuration, ROS needs three times as long as ZMQ or UDT to receive an acknowledgement (uint_16) about the successful command transfer.

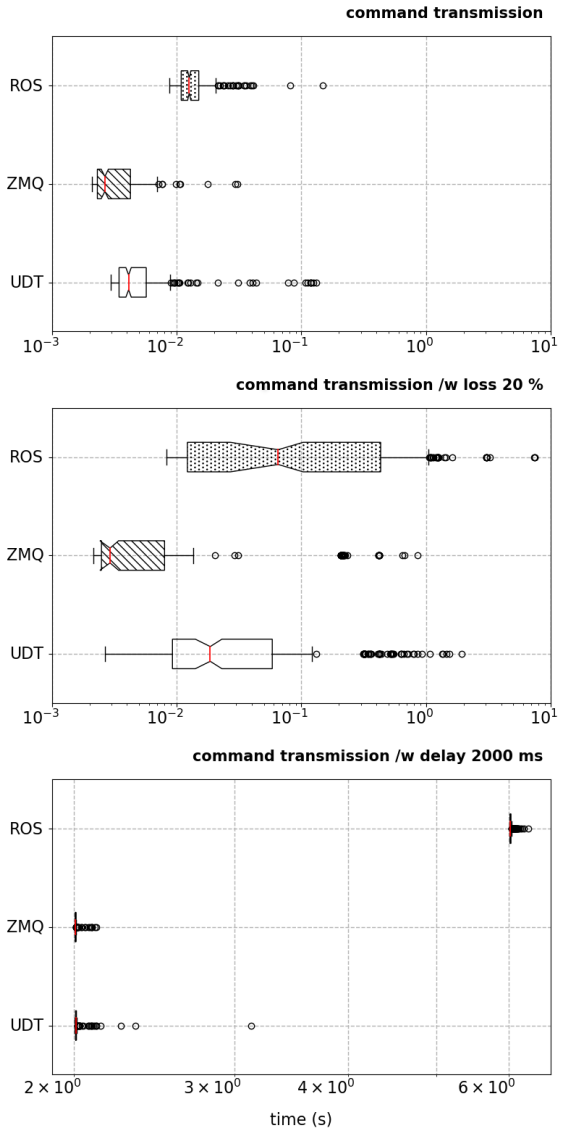

Figure 1: Comparing transmission times (s) for random twist commands using ROS or one of two transport implementations of the RRC library, based on either ZMQ or UDT. Results are obtained on a local wireless network in different configurations: default (top row), emulated package loss of 20% (mid row) and artificial delay of 2 000 ms (bottom row). Results are plotted on a logarithmic x-axis.

3.3.2 Telemetry Transfer

In Figure 2 and 3 measurements of round trip time for telemetry data are displayed. Telemetry data is constructed as a random string of specified size (1 and 10 MB) and transmitted via either ROS communication (topics) or with the ZMQ or UDT transport implementation of the RRC library.

For telemetry data with a size of 1MB all tranport options deliver in a similar round trip time. ROS transfers the random string forth and back in an average of 1.7 seconds, while ZMQ and UDT perform the same task in 1.16 and 1.45 seconds respectively. For telemetry data with 10 MB the difference in round trip time becomes more evident. ROS transport amounts to an average of 18.16 seconds, while ZMQ performs the round trip transfer in 11.4 seconds. UDT takes an average of 3.8 second. Even though the initial and usually longest round trip time is sliced away of all plotted data, ZMQ and ROS show a number of significant outliers, while UDTs outlier are much more contained and not even visible for large telemetry.

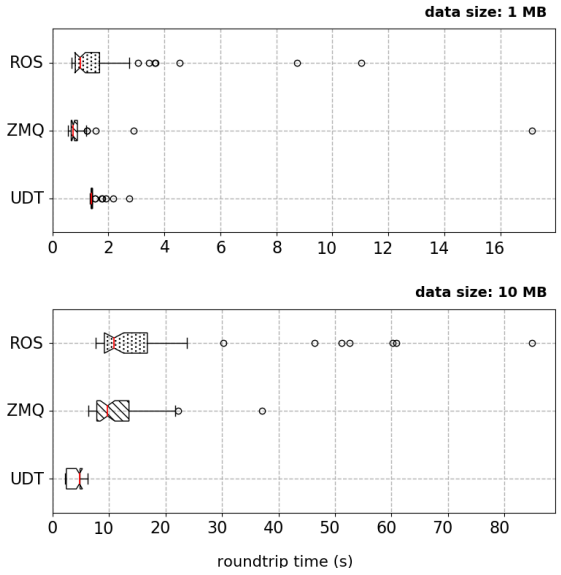

Figure 2: Round trip time of telemetry transfer using ROS topics and the two transport implementations ZMQ and UDT of the RRC library. The two rows compare transfer of 1MB (top row) and 10MB (bottom row) respectively.

From the three transport options only ROS and UDT are able to handle an emulated delay of 2000 ms. ZMQ is not able to successfully transfer data, even though a connection is established. On average ROS (17.6 s) takes more than 3.5 times longer than UDT (4.8 s) to transfer 1 MB of data. For larger telemetry data (10MB) this factor decreases to 1.4. However UDT takes about 15 seconds less (44.7 s) than ROS (61.6 s).

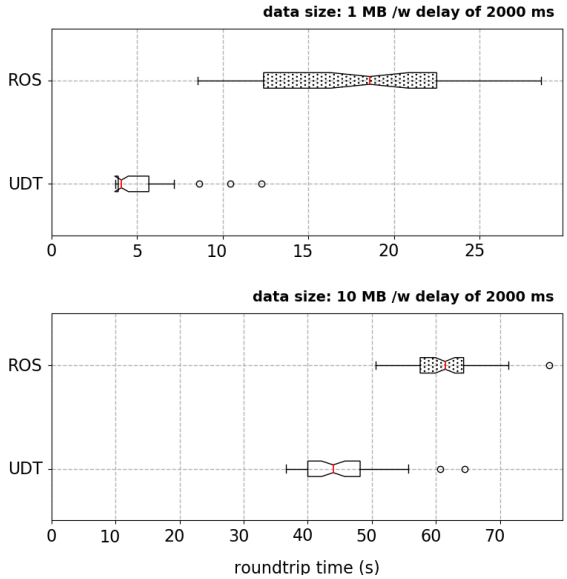

Figure 3: The two plots above show the round trip time of telemetry data (1 MB – top row, 10 MB – bottom row). The performance of ROS is compared to the UDT transport implemented by the RRC library. Both in a network with an emulated delay of 2000 ms. The ZMQ implementation cannot transfer data in networks with such high latency and is therefore excluded.

3.4 Discussion

While all transport protocols show similar performance on the default network, a disadvantage of ROS becomes evident for transfer of commands or data on a network with high delays or package loss. For the acknowledged commands this may be due to the fact that the ROS service calls are based on ROS topics. Therefore two seperate connections are used to transfer the request and response. This may result in the measured delays in comparison to the point-to-point connection of UDT or the immediate request and reply sockets of ZMQ.

In terms of telemetry transport UDT outperforms the other two transport implementations especially for higher loads and for high latency. This explicit advantage of UDT is based on the comparably small over-head and the connection-less character of the underlying UDP protocol in comparison to the connection-oriented heavy-weight TCP protocol. By design the UDT protocol is especially suited for data intensive applications, which proves well in the presented results and becomes even more evident in the experiments with emulated delay.

5 CONCLUSION

The library acts as an alternative communication layer to existing RCOS solutions. It is shown in the scope of this paper that especially the configurable UDT transport implementation of the presented RRC library can serve as a more reliable and fast communication library than what is currently being used in native ROS. While already being able to compete with state of the art middleware, the most prominent advantage of the library is that it provides configurable communication sockets to tailor the communication to task specific requirements and use custom protocols. The configurable and modular character of the communication sockets is further enhanced through the availability of reliable, acknowledged transfer via the command channel or non-blocking data transmission over the telemetry sockets. Moreover the provided transport options are configurable. ZMQ allows for a number of messaging patterns and UDT offers control over reliability and congestion algorithms.

Due to the lightweight and framework independent character of the library, it is already being used extensively in several projects, e.g. the EU funded, space robotics project PRO-ACT [4]. Providing a common communication layer between multiple robots for cooperative manipulation, sockets for independent control of base and manipulators, robust monitoring for mission planning and even a common message definition for sensor fusion between different frameworks. At this point it should be highlighted that control or monitoring instances do not need to be compatible with the framework of the entity that is running the CR. They solely have to wrap the RRC library or even just the RC one. Both being framework and OS independent.

In addition the library can be used as a configurable, generic communication interface for arbitrary purposes not only for actual hardware applications. It could even be used as a common interface for exchanging simulation engines or for realizing computation in distributed setups. Due to the support of C++ libraries in many other programming languages like Java, python or Go the presented library can even serve as an adapter between various programming languages.

Acknowledgement

The work presented is part of several projects. The project Mare-IT (grant no. 01|S17029A) is funded by the German Space Agency (DLR Agentur) with federal funds of the Federal Ministry of Economics and Technology in accordance with the parliamentary resolution of the German Parliament. The project ROBDEKON (grant no.13N14675) is funded by the Federal Ministry of Economics and Technology in accordance with the parliamentary resolution of the German Parliament. PRO-ACT is funded under the European Commission Horizon 2020 Space Strategic Research Clusters - Operational Grants number 821903.

References

[1] Sylvain Joyeux, Jakob Schwendner and Thomas M. Roehr (2014) Modular Software for an Autonomous Space Rover. *In Proceedings of the 12th International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS), Montreal, Québec, Canada:* pp. 1–8.

[2] Postel, Jon (1980) RFC0768: User Datagram Protocol.

[3] Gu, Y. and Grossman, R. L. (2007) UDT: UDP-based data transfer for high-speed wide area networks. *Computer Networks 51*(7):1777-1799.

[4] Brinkmann W., et al. (2020) Enhancement of the six-legged robot Mantis for assembly and construction tasks in lunar mission scenarios within a multi-robot team. In Proceedings: *International Symposium on Artificial Intelligence, Robotics and Automation in Space*.