Felix Frank, Alexandros Paraschos, and Patrick van der Smagt
"ORC—a lightweight, lightning-fast middleware".
IEEE International Conference on Robotic Computing (IRC), 2019
https://doi.org/10.1109/IRC.2019.00061

ORC—a lightweight, lightning-fast middleware

Felix Frank, Alexandros Paraschos, Patrick van der Smagt
Machine Learning Research Lab, Volkswagen Group, Munich, Germany
Email: {felix.frank,paraschos}@argmax.ai

***Abstract*—Robotic tasks are commonly solved by integrating numerous different software and hardware modules into one working application. The necessary integration work typically contributes a considerable share of the total work required for a project, which is why past research on robotics computing has pushed towards generating higher-level abstraction layers, like middlewares. However, the current state-of-the-art cannot provide reliable, low-latency communication performance as we will show in the experimental evaluation. In this paper we propose the Open Robot Communication framework (ORC). Compared to previous middlewares, ORC is lightweight and geared towards applications with high-performance requirements. We consider ORC especially useful for applications with Human Robot Interaction or collaborative tasks involving multiple robots. In the paper, we compare the runtime performance of ORC to the robot operating system (ROS). We can show that ORC enables message transfer with delays far below one millisecond and we demonstrate the real-time capabilities of ORC in a force-control task implemented in Python.**

I. Introduction

The field of robotics is characterized by a very complex and multidisciplinary nature. Robotic tasks typically have to be solved by combining control theory with a multitude of sensor inputs, like force sensors or vision, and a decision-making system. Many tasks require the integration of a variety of hardware parts, where it is necessary to combine custom low-level code. Therefore, robotics is a domain that can benefit from modular and reusable code. In the past, this has led to the development of numerous software tools tailored towards providing higher-level abstractions, effectively reducing the time required to implement an application. One example of software, promoting the development of modular functionality, are middleware solutions. Middlewares provide application developers with a communication mechanism, enabling the coordination of distributed programs through information exchange. Prominent examples are the robot operating system (ROS) [1], the open robot control software project (OROCOS) [2], the *OpenRTM-AIST* [3] and *Yet Another Robot Platform* (YARP) [4]. Other research projects, like the player project [5] have worked towards creating hardware abstraction layers or are implementing model-driven methods for robotic behavior specification [6], [7]. For a detailed overview of available solutions the reader is referred to [8].

Previous to developing ORC, we evaluated YARP and ROS for our applications. YARP offers two main features. First, it enables communication between different software modules through a protocol similar to publish–subscribe. Second, YARP provides developers with a Hardware Abstraction Layer (HAL) for commonly used hardware. Combined, those two features can significantly reduce the development time for a novel application. However, compared to other middlewares, the communication setup in YARP is non-trivial, making it hard for new users to benefit from the message exchange functionalities. Additionally, YARP does not offer real-time performance. For most applications that is fine, however, it eliminates YARP as a candidate for tasks with critical safety functions.

ROS, on the other hand, is the de facto standard for middlewares in the robotics community. ROS offers a publish–subscribe communication paradigm in which programs, called nodes, can publish messages through separate communication channels, called topics. Topics are uniquely identified by name. Messages in ROS are strongly typed. Developers have to specify the exact message structure in an interface description file, which can then be used with the ROS build system to generate the required code for serialization and deserialization of the message type. Afterwards developers can use the generated source files to incorporate the message type into their application. The ROS ecosystem offers a great amount of functionality, however, the core of ROS, the underlying message passing system, can often not fulfill the high performance requirements, especially when used for low-level control. We show that communication through ROS can introduce high latencies in the range of tens of milliseconds, especially for messages sized around one to ten kilobyte. This can introduce an excessive delay into the system, preventing the use of ROS in low-level control applications.

In this paper, we propose a novel, lightweight and lightning-fast middleware, the Open Robot Communication (ORC) framework. ORC is an alternative middleware for applications with high-performance requirements. We demonstrate the performance of ORC with message latencies well below one millisecond and we establish the utility of ORC in real applications by showing a real-time force controller implemented in Python. In general direct force-control enables robots to adapt to the environmental constraints rather than modeling those in detail beforehand. Many interaction tasks such as mechanical part mating or polishing use robotic force-control to prevent failures due to excessive buildup of

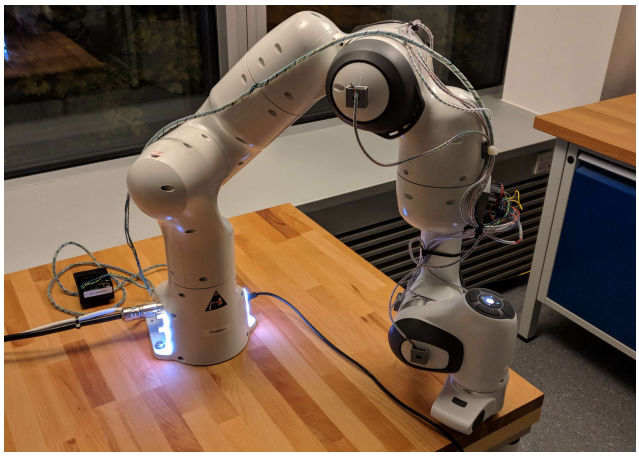


Figure 1. Franka Panda robot that we used to evaluate the performance of our approach in a force-control task

the contact force over the mechanical limits [9]. Compared to the encoders used in robotic motion control, force–torque sensors have a higher noise level, which in turn requires a low-latency control systems to perform accurate reference tracking of the contact force. For the experiments we use a Panda robot from FRANKA EMIKA. The setup is depicted in Figure 1.

The remainder of the paper is organized as follows: Section II presents ORC's software architecture and the essential criteria influencing the design of the middleware. Afterwards, section III highlights important implementation details and shows the interface available to developers. Section IV first presents a round-trip latency comparison between ORC and ROS. Second, we show that ORC can be used for robotic force-control with contact while using a Python controller. The paper ends with a summary of the contribution in section V.

II. Software Architecture

ORC is designed with the several major requirements in mind. First, to deliver a high-performance communication mechanism that enables message transfer with minimal delays below one millisecond, when handling messages up to the size of a megabyte. Second, ORC shall be close to real-time capable. TCP, due to the retransmissions, cannot guarantee an upper bound on latency, however, ORC shall be optimized for the use with a Linux RT patched kernel [10], [11]. Therefore it shall deliver low-latency communication even if the system is under heavy load. Third, we propose a minimal set of dependencies to guarantee compatibility with a wide range of hardware devices. Fourth, in the near future ORC shall make it considerably easier to combine high-level machine learning methods with low-level control functions. As such ORC provides client libraries in C++ and in Python. Finally, it should be easy to develop application code using ORC. As such, we aim to provide a high-level interface, enabling application developers to use ORC's functionality with a few lines of code.

The communication pattern offered by ORC is a one-to-many publish–subscribe pattern. A communication participant can register topics, which are uniquely identified by name. Other participants can subscribe to these topics. As in ROS, topics are separate communication channels which can be used for message transfer. The software is split into a backend application, called the *Address Broker* and a library, *liborc*, providing access to the communication mechanism. Compared to a classic broker approach, where every message is routed through the backend, ORC's *Address Broker* is designed such that it is only necessary for setting up the communication channels between the individual participants. After setup, the participants use direct one-to-one connections between the individual participants. Additionally, one-to-many multicast support is currently investigated. The inner working and structure of the communication in the case of three distinct participants is shown in Figure 2. The *Address Broker* keeps track of the individual participants and the topics they can offer in a special data structure called the *Topic Mapping*. The participants use a custom request–reply protocol to coordinate topics and corresponding addresses with the backend. This is visualized in the top half of the Figure. Participants can declare and request topics from the *Address Broker* by sending messages to the backend. The *Address Broker* stores and distributes all information necessary for participants to open communication channels with each other. These channels, visualized in the bottom half of the Figure, follow a publish-subscribe pattern. A publisher can have an arbitrary number of subscribers. Equally, participants can subscribe to as many topics as they wish. The upper limit for available topics is given solely by the availability of TCP ports in the system. All the communication is handled by the middleware, so application developers can be ignorant to where their messages come from or go to.

III. Software Implementation

Numerous communication libraries and data distribution services have been proposed in the recent years. For developing ORC we evaluated nanomsg[1], eProsima Fast RTPS[2], nanomsg next generation[3] [12] and zeroMQ[4] [13]. The core part of ORC is implemented entirely in C++, where we use zeromq to build both our publish–subscribe and the request–reply protocols. ZeroMQ offers a broader functionality and a bigger user base, compared to the nanomsg successors. Fast-RTPS offers a much more high-level data distribution mechanism, which we found to be too constraining for our application. ORC is implemented as a non-

[1]https://nanomsg.org/
[2]https://github.com/eProsima/Fast-RTPS
[3]https://nanomsg.github.io/nng/
[4]http://zeromq.org/

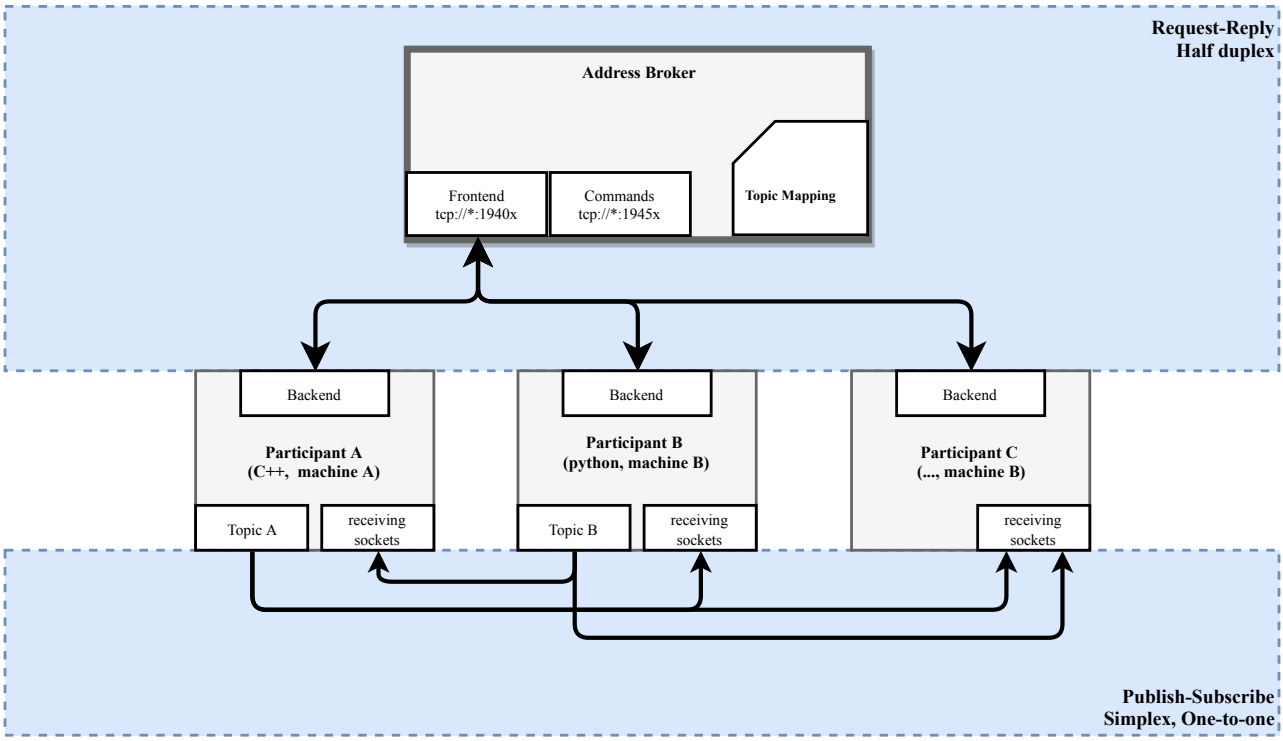


Figure 2. ORC's internal structure. The *Address Broker* backend application handles the communication setup visualized in the top, whereas individual participants can exchange messages with a publish-subscribe pattern shown in the bottom half

blocking message queue with only one message. Sending and receiving messages are equivalent to enqueueing or dequeuing messages into the queue, respectively. Both of these actions never block unless an application wants to. The actual message delivery is handled asynchronously in the background. The queue size of one guarantees that a subscriber will always have the latest delivered message. In this way, the middleware takes care of potential issues with slow consumers. Slow consumers are a common appearance in robotics because typically low-level control and high level task reasoning run at very different frequencies.

ORC uses three distinct communication pathways depending on how the participants are physically connected to each other. For message transfers between two participants within the same process, the zeromq INPROC protocol is used. It is based on classic operating system pipes and delivers constant latencies independent of the message size. If two participants are not within the same process, but are located on the same host, fast linux interprocess sockets are used. Finally, if the two participants can only communicate through a network, ORC currently uses TCP to deliver messages. A single publisher can have separate one-to-one connections through all three communication channels at the same time and for the same topic. Referring back to Figure 2, Participant B will use TCP to communicate with Participant A, while at the same time, the connection to Participant C is established through a linux interprocess socket.

Another feature of ORC is its robustness to network failures. The *Address Broker* keeps track of past connections and any application program which is restarted after a critical failure will reconnect to lost connections. Temporary network outages are handled by the underlying zeromq framework automatically.

A. Application interface

ORC offers a very simple interface for any application code. The core functionality is available through four distinct functions. Two of them are used for setting up the communication channels. With these functions, applications can declare and request topics by name. Internally, the request–reply protocol is used to update the *Topic Mapping* managed by the *Address Broker*.

After the communication setup, an application can use two additional functions for sending and receiving messages. Messages in the ORC framework are considered to be byte arrays. Consequently, users have to take care of serialization and deserialization of the transmitted information. This has the advantage that ORC is compatible with any serialization format. Internally, for the request–reply protocol, ORC uses Google's flatbuffers technology [14], however, users may use protocol buffers [15], JSON or any other format which can be serialized into a byte array. The send method requires only the topic name, a pointer to contiguous memory and

the size of the memory region, to transfer a message. The receive method on the other hand, has a timeout parameter, additional to the topic name. The timeout allows applications to explicitly wait until new messages arrive in the message queue. This enables application developers to write callback-like functionality to message delivery events.

Both the send and the receive function are designed to be nonblocking. Without a given timeout, in the case of the receive method, the function calls return immediately. Internally sending is equivalent to queuing a message for delivery, while a call to receive simply fetches the latest available message out of the queue. In Listing 1 we show a small template application code written for ORC. This small program could be a robot controller as indicated by the comments. The important lines however are the lines 1, 6, 9, 13, 20 and 26. They show how the entire functionality of ORC can be accessed within a few lines of code.

Listing 1. **Template application code for ORC**

```
1  #include "orc.h"

   int main(int argc, char const *argv[])
   {
     /* Communication Setup */
6    orc::broker b;

     /* Declare publishing topic */
     if (!b.register_topic("robot_control")) {
       return EXIT_FAILURE;
11   }

     /* Subscribe to a topic */
     if (!b.request_topic("robot_position")) {
       return EXIT_FAILURE;
16   }

     /* Main execution loop */
     while(1) {
       auto msg = b.recv_msg("robot_position", -1);
21
       /* Deserialize message */
       /* Use the information to compute control */
       /* Serialize reply */

26     if (!b.send_msg(
                       "robot_control",
                       reply, reply_size) {

         /* Error handling */
31     }
     }
     return 0;
   }
```

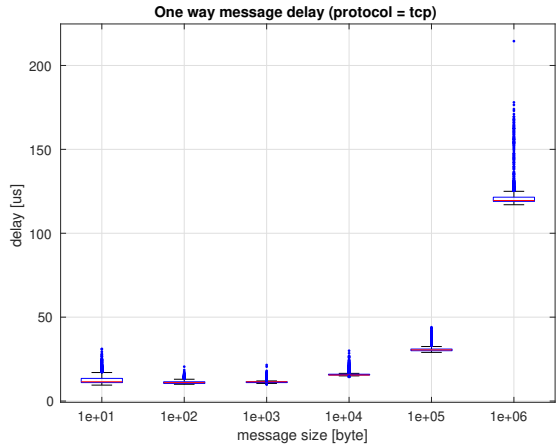
IV. Experiments

A. Performance Evaluation

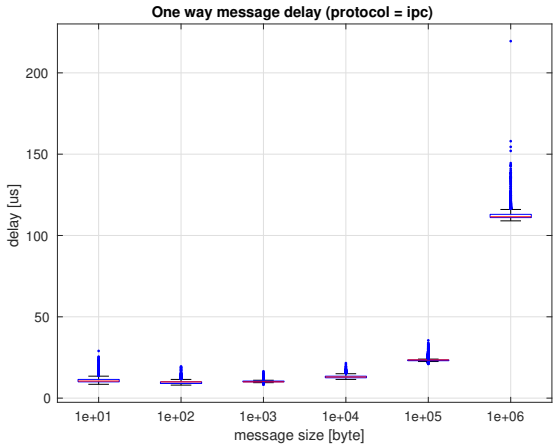
For evaluation of the delay introduced by sending messages a round-trip latency test was performed. The experiment includes two participants exchanging messages. Participant A sends out a message and waits for a reply. Participant B waits for a message and sends back the exact same message to the original sender. The time between sending a message and receiving a reply is measured by participant A. This yields the round-trip delay, which is halved to acquire the one-way delay. The experiment is repeated with different message sizes ranging from 10 bytes to 100 megabytes. We evaluate the performance of our approach by computing the average latency between the two communication participants for exchanging one million messages in a row. We tested all three communication channels available to ORC. For the messages sent through TCP, the two participants are on two separate machines which are directly connected through their own private network.

We compare the performance of ORC to the ROS middleware by performing the same experiment, where two communicating ROS nodes were instantiated on the same computer. The results of the experiments are visualized in Figure 3 and Table I. Figure 3 shows a box plot for the latency distribution of the experiment. Outliers are indicated by a blue dot, while the median is shown as a red line. The whiskers represent the range which includes 99.3% of all measurements. The Figures 3a to 3c show the latency when sending messages through the different communication channels. The Figures demonstrate that ORC can handle message transfer with latencies well below one millisecond with minimal fluctuation. The Figures 3d and 3e, show the experimental results for the ROS experiments. They are split up into two separate Figures for better visualization. We experienced issues with ROS when the message payload is around one kilobyte, and therefore it is difficult to show the exact statistics of all message sizes together in one plot. For a message holding 1 kilobyte of data, the delay increases significantly and lies at an average of 20 milliseconds. The problem was confirmed on three separate machines with different linux operating systems, namely Ubuntu 16.04, 18.04 and Arch Linux. For other message sizes ROS introduces an average delay between 180 microseconds, for 10 byte messages, and 1.5 milliseconds for messages with 1 megabyte of size. However, in contrast to ORC, the experiments with ROS show significant fluctuation. Even for small messages, we measured outliers with delays in the millisecond range. Furthermore, the problem with messages of a size around 1 kilobyte raises the potential delay between two communication participants up to 20 and more milliseconds. A delay of that order, combined with the unreliability of the communication system, renders ROS unusable for any high-performance applications in robotics.

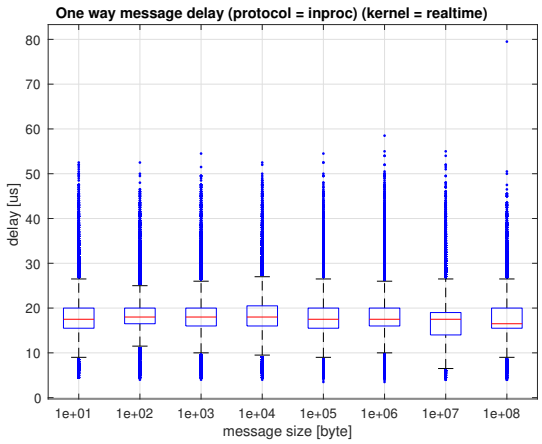
Table I shows the average latencies introduced by sending a message with ORC. We can observe that ORC performs at least a factor 10 faster than ROS independent of the message size. While we were unable to use ROS for messages larger than 1 megabyte, we can do so with ORC.With large messages, we see a linear increase in message latency.



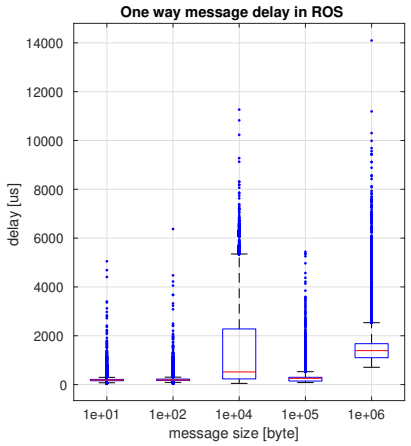
(a) TCP



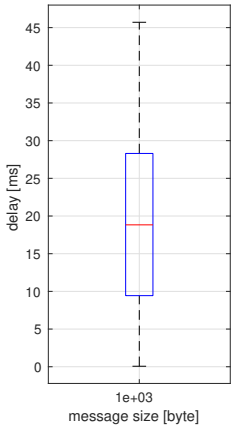
(b) IPC



(c) INPROC



(d) ROS without 1 kilobyte



(e) ROS 1 kilobyte

Figure 3. Latency distribution of our approach compared to ROS on sending one million messages of different sizes

B. *Robot force-control*

For this experiment we implemented a force-controller for a Panda robot from FRANKA EMIKA. The Panda is a seven degrees-of-freedom robotic manipulator with a torque sensor in every axis. Closing the control loop on contact force requires the controller to use torque sensors instead of relying on the joint encoders. Torque sensors have a higher noise level compared to the encoders, which requires a high-bandwidth controller to perform accurate reference tracking. Even though Python is not real-time capable and has to be considered as introducing a considerable delay by itself, we show that a force controller implemented in python is feasible with low-latency communication.

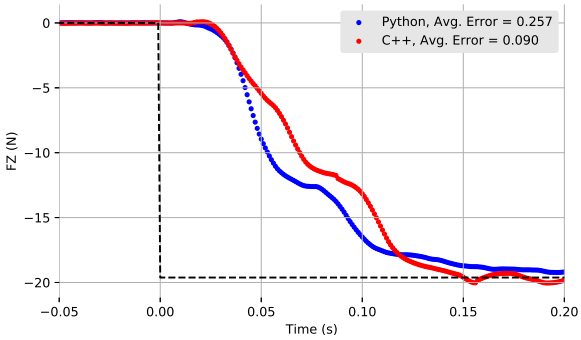
The Panda robot requires a controller running at a 1 kHz frequency, and it can be interfaced through a C++ library called *libfranka*[5]. For our setup we use a small C++ wrapper to directly interface with the robot and provide a control signal at a reliable rate. Simultaneously this wrapper constantly publishes the robot state information and listens to a topic for torque commands, which are directly passed on to the robot. Additionally, the wrapper attenuates the desired torque in case of a communication failure to ensure a safe behavior during the experiments. The actual control loop is implemented in a callback-like fashion with Python. The controller waits for an update about the state of the robot and uses the robot kinematics to map a desired end-effector force to a desired joint space torque. Afterwards, the desired torque is used as a feed-forward signal, whereas a PI controller complements the setup to ensure disturbance rejection.

We compare the performance of this controller, implemented in Python, to one which is implemented directly in C++ and does not use ORC to exchange messages. In the experiment the robot end-effector is faced flat towards a table

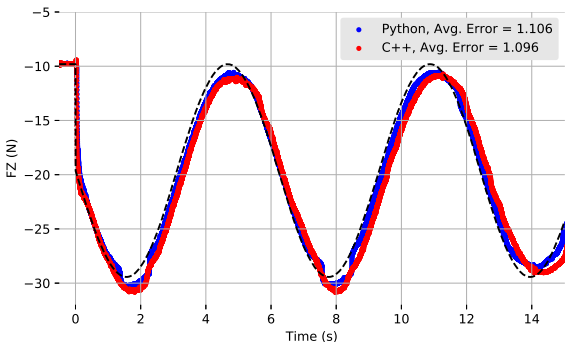
[5]https://github.com/frankaemika/libfranka

Table I
AVERAGE ONE WAY MESSAGE DELAY IN MICROSECONDS FOR EACH TRANSPORT PROTOCOL DEPENDING ON THE MESSAGE SIZE

Message size	10 b	100 b	1 kb	10 kb	100 kb	1 mb	10 mb	100 mb
Transport protocol								
INPROC	18.40	18.24	18.41	18.45	17.98	18.35	17.46	17.73
IPC	10.94	9.74	10.09	13.08	23.43	112.20	1760.63	17160.96
TCP	12.43	11.10	11.48	15.87	30.69	120.49	2068.06	19876.30
ROS	181.14	199.48	19034.53	1455.61	242.01	1494.74	-	-



(a) System step response



(b) Sinusoidal reference trajectory

Figure 4. System response comparison for a controller implemented in Python using ORC (blue) and a controller implemented directly in C++ (red)

and applies a reference contact force. An example system response to a step input is shown in Figure 4a, whereas Figure 4b shows one example of the system's response for a sinusoidal reference trajectory. The blue line depicts the controller using ORC, the red line is the controller using *libfranka* directly and the dashed black line is the reference trajectory. The Figures show that the control performance is barely influenced by constant communication between the low-level robot interface and the controller itself. The experiments demonstrate that a high performance communication mechanism, which provides message transfers with minimal delay, can enable controllers written directly in high-level abstraction languages like Python, without incurring significant penalties with respect to control performance. Table II summarizes mean and standard deviation of the tracking error during the experiments. Every experiment was repeated six times.

Table II
MEAN AND STANDARD DEVIATION OF THE TRACKING ERROR IN NEWTON FOR TWO CONTROLLERS AND DIFFERENT REFERENCE TRAJECTORIES (N = 6)

Controller	Python	C++
Reference Trajectory		
Step Response	0.599 (+/- 0.449)	0.455 (+/- 0.395)
Sinusoidal Trajectory	1.079 (+/- 0.093)	0.968 (+/- 0.136)

V. CONCLUSION

Modular software development has proven to be a viable way to deal with the high complexity present in robotic tasks. However, the combination of such modules requires middleware applications like ROS, OROCOS or YARP. While these solutions provide a multitude of functionality, applications with high performance requirements cannot utilize these tools. In this paper, we presented a novel middleware geared specifically towards these robotic applications with high performance requirements. Therefore, the Open Robot Communication middleware is presented as a lightweight alternative to the current state-of-the-art. ORC can perform message exchange with latencies far below one millisecond, enabling novel applications to use the benefits of modular software design. We demonstrated that a low-latency communication enables controllers to be written directly in higher level languages. Specifically our middleware has the following advantages over other middleware solutions:

- Minimal delay introduced by the message transfer
- Compatibility with many available serialization libraries
- High-level interface to the middleware functionality

Furthermore, ORC was optimized with respect to a preempt RT patched kernel, which ensures high performance even under heavy load. This advantage is relevant for tasks with strict safety requirements. Examples are collaborative tasks with robot and human in the loop.

Future work will investigate the use of multicasting and UDP based protocols. Especially for messages with a bigger payload, like images, the publisher, internally, has to copy the message and send it to every subscriber individually, whereas multicasting would allow ORC to use the specialized network devices for the duplication. Additionally we are planning on releasing a small state machine library, which allows developers to specify modular functionalities using ORC, and enables convenient construction of entire applications by combining and scheduling these modules.

Finally, we would like the research community to benefit from ORC. Therefore, the Open Robot Communication middleware is available as open source code on our github repository[6]. Currently, we have tested and verified ORC on multiple linux operating system with Intel and ARM processors, which are only constrained by the necessity of a compiler supporting the C++11 standard.

REFERENCES

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

[2] H. Bruyninckx, "Open robot control software: the orocos project," in *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, vol. 3. IEEE, 2001, pp. 2523–2528.

[3] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, and W.-K. Yoon, "Rt-middleware: distributed component middleware for rt (robot technology)," in *Intelligent Robots and Systems, 2005.(IROS 2005). 2005 IEEE/RSJ International Conference on*. IEEE, 2005, pp. 3933–3938.

[4] G. Metta, P. Fitzpatrick, and L. Natale, "Yarp: yet another robot platform," *International Journal of Advanced Robotic Systems*, vol. 3, no. 1, p. 8, 2006.

[5] B. Gerkey, R. T. Vaughan, and A. Howard, "The player/stage project: Tools for multi-robot and distributed sensor systems," in *Proceedings of the 11th international conference on advanced robotics*, vol. 1, 2003, pp. 317–323.

[6] A. Paraschos, N. I. Spanoudakis, and M. G. Lagoudakis, "Model-driven behavior specification for robotic teams," in *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*. International Foundation for Autonomous Agents and Multiagent Systems, 2012, pp. 171–178.

[7] D. Calisi, A. Censi, L. Iocchi, and D. Nardi, "Openrdk: A modular framework for robotic software development." in *IROS*, 2008, pp. 1872–1877.

[8] A. Elkady and T. Sobh, "Robotics middleware: A comprehensive literature survey and attribute-based bibliography," *Journal of Robotics*, vol. 2012, 2012.

[9] B. Siciliano and L. Villani, *Robot force control*. Springer Science & Business Media, 2012, vol. 540.

[10] Real-Time Linux (RTL) Collaborative Project, "Fully-preemptible linux kernel," https://wiki.linuxfoundation.org/realtime/rtl/start, 2015, [Online, accessed: 10.06.2018].

[11] P. McKenney, "A realtime preemption overview," *http://lwn.net/Articles/146861/*, 2005.

[12] G. D'Amore, "Nng reference manual," 2018.

[13] P. Hintjens, *ZeroMQ: messaging for many applications*. O'Reilly Media, Inc., 2013.

[14] Google, "Flatbuffers, an efficient cross platform serialization library," https://google.github.io/flatbuffers, [Online, accessed: 08.06.2018].

[15] K. Varda, "Protocol buffers: Google's data interchange format," *Google Open Source Blog, Available at least as early as Jul*, vol. 72, 2008.

[6]https://github.com/argmax-ai/orc