Software Impacts 9 (2021) 100109

Contents lists available at ScienceDirect

Software Impacts

journal homepage: www.journals.elsevier.com/software-impacts

Original software publication

RTCF: A framework for seamless and modular real-time control with ROS

Michael Fennel [*], Stefan Geyer, Uwe D. Hanebeck

Intelligent Sensor-Actuator-Systems Laboratory (ISAS), Institute for Anthropomatics and Robotics (IAR), Karlsruhe Institute of Technology (KIT), Adenauerring 2, 76131 Karlsruhe, Germany

ARTICLE INFO

Keywords:
Robotics
Real-time
Control
Robot Operating System
Linux

ABSTRACT

Owing to the steady progress in the field of Linux kernel development, high-performance control applications are no longer a rarity on general-purpose computing platforms. However, many real-time control libraries lack important properties such as modularity, effortless integration, and encapsulation. These are key design features of the popular Robot Operating System (ROS) that is, however, not real-time capable. We aim to solve this issue by introducing the Real-Time Control Framework (RTCF), which offers high modularity, ROS-related concepts leading to seamless interoperability with ROS, and high performance. To demonstrate the capabilities of the RTCF, we provide several examples and exemplary performance data.

Code metadata

Current code version	v1.1.0
Permanent link to code/repository used for this code version	https://github.com/SoftwareImpacts/SIMPAC-2021-74
Permanent link to Reproducible Capsule	
Legal Code License	MIT license
Code versioning system used	github
Software code languages, tools, and services used	Robot Operating System, OROCOS toolchain, Preempt-RT
Compilation requirements, operating environments & dependencies	Linux with ROS, optionally with Preempt-RT patch
If available Link to developer documentation/manual	https://github.com/KIT-ISAS/RTCF/blob/master/README.md
Support email for questions	michael.fennel@kit.edu

1. Introduction

In recent years, *Robot Operating System* (ROS) [1] has become a de-facto standard in the area of robotics research and development. The reasons for its popularity include the separation of functions through interfaces, resulting in modular and flexible architectures, and a set of integrated tools for standard tasks such as data logging, visualization, and parameterization. When ROS was developed, its main focus was put on performing higher-level algorithms and as a consequence, its asynchronous architecture is neither real-time safe nor does it provide a built-in option for synchronous operation. This becomes a limitation when mid-level or even low-level control algorithms with strict timing requirements are implemented in ROS, as the achievable performance is very limited and depends on external factors. Nevertheless, developments such as the *Xenomai* [2] or the *Preempt-RT* [3] kernel patch have proven that it is possible to deploy applications under real-time constraints on a general-purpose computer running Linux.

While a standalone application with real-time capabilities can be easily implemented in such systems, it is much more difficult to create a complex application that still integrates well into an existing robotic software project. Several attempts to simplify the integration of real-time capabilities into ROS have been made in the past. The library *ros_control* [4] encapsulates the control tasks for a specific hardware device in a ROS node, resulting in limited modularity for complex scenarios (e.g., controller cascades and multiple hardware devices). As an alternative with a large number of features, OROCOS [5] can be used in conjunction with the *rtt_ros_integration*-package [6]. Despite its functional superiority, OROCOS is rarely applied in practice due to a steep learning curve and complex concepts. With the arrival of ROS 2, real-time capabilities are finally taken into account in the design of

* Corresponding author.
E-mail addresses: michael.fennel@kit.edu (M. Fennel), mail@stefan-geyer.org (S. Geyer), uwe.hanebeck@kit.edu (U.D. Hanebeck).

https://doi.org/10.1016/j.simpa.2021.100109
Received 5 July 2021; Received in revised form 9 July 2021; Accepted 14 July 2021

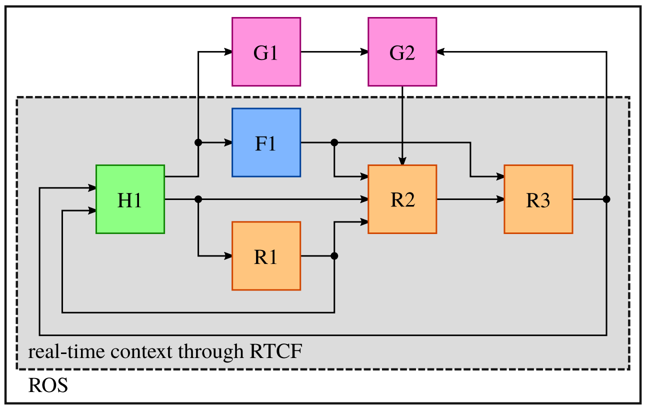


Fig. 1. An exemplary structure of a desired real-time architecture with ROS integration as it might occur in real-world robotic applications. In this example, a hardware interface (H1), a state estimation filter (F1), and three controllers (R1–R3) are real-time components, that are interwoven with some standard ROS nodes (G1–G2).

ROS [7]. However, at the current state, the real-time support of ROS 2 requires significant amounts of boilerplate code and the connections between nodes are hard-coded, which contradicts some of the core ROS principles. Moreover, many users still rely on ROS 1. Beyond that, there exist several less well-known frameworks that combine ROS with real-time control [8–10]. The disadvantages of these are limited modularity and incomplete integration of popular ROS workflows.

For this reason, we introduce the *Real-Time Control Framework* (RTCF) as a new way to seamlessly build modular and high-performance controller architectures within a ROS ecosystem. For robotic systems, this means that complex low- and mid-level controller architectures, e.g., with nested control-loops or controllers that are contributed by different teams, can be realized on a single general-purpose computer without sacrificing simplicity, performance, and reusability. This is achieved by choosing OROCOS as a base and adding the following requirements, which are derived from our experience with the above-mentioned frameworks and the example in Fig. 1.

- *Modularity:* Components are reusable, interfaces are well-defined, and the configuration is conducted at runtime. Regarding Fig. 1, this means that the connections between the blocks are not hard-coded.
- *Interoperability:* Existing ROS tools (e.g., launch files, topics, parameter server, logging) are fully compatible. In the example, the connections between the RTCF and ROS must be handled transparently.
- *Performance*: A real-time safe execution with low overhead is guaranteed. The desired overhead and jitter are in the range of 10 μs to facilitate control frequencies up to several kilohertz.
- *Ease of use:* The framework is easy to learn for existing ROS users due to similar concepts.

2. Design

In the following, the main features and design concepts of the RTCF will be discussed briefly.

2.1. Components

Since the RTCF is built on top of OROCOS, the smallest functional unit is a so-called *component*. Each component has a pre-defined lifecycle as well as input and output interfaces that are called ports. These ports are very similar to the publisher–subscriber mechanism in ROS, and thus components are conceptually very similar to ROS nodes. A minimal working example of such a component is shown in Listing 1.

2.2. Dependency resolution

A major difference between the RTCF, ROS, and OROCOS is the way components are executed. Both ROS and OROCOS execute the loaded nodes or components in concurrent threads by default. This is inappropriate for the targeted control applications, where a deterministic behavior and minimal overhead are desired.

For this reason, the RTCF executes all loaded components sequentially in a single, real-time capable thread. The determination of the order is done through an automated dependency resolution using Kahn's algorithm [11] and the list of predecessors and successors of each component. To break possibly occurring loops (e.g., in hardware interfaces with sensors and actuators), some connections can be manually excluded. The resulting order for the example in Fig. 1 is then H1–F1–R1–R2–R3.

2.3. Interoperability with ROS

A major feature of the RTCF is its seamless interoperability with ROS, which utilizes parts of the rtt_ros_integration [6].

Launch Files: To launch a whole controller architecture with numerous components, two ROS nodes, *rt_runner* and *rt_launcher*, are available. While the first node is responsible for holding, managing, and executing the actual payload similar to a ROS *nodelet_manager* [12], the latter allows the convenient loading of components through standard ROS commands and launch files. As a result, there is no need to learn anything new, such as the OROCOS scripting language, for a ROS developer. Listing 2 shows an exemplary RTCF launch file. Except for the package name and the component type, which are moved to the `args`-attribute, this completely works like any normal launch file.

Topics: The RTCF provides an option to transparently map connections from real-time components to ROS topics and vice versa. The setup of this is achieved through a simple whitelist regular expression. Possible not real-time capable side channels are automatically detected and avoided.

Parameters: In contrast to existing solutions, the RTCF facilitates easy access to ROS parameters from component context at configuration time by providing a standard ROS node handle. Beyond that, a real-time safe wrapper for *dynamic_reconfigure* [13] is provided. This means that real-time controllers can be tuned at run-time with existing ROS tools.

Logging: Logging from a real-time context needs special care due to the required memory allocations. For this reason, the RTCF extends the OROCOS real-time logging system with logging macros similar to ROS. Furthermore, integration into the ROS logging system, including *rqt_console* as well as *rtq_logger_level*, is provided.

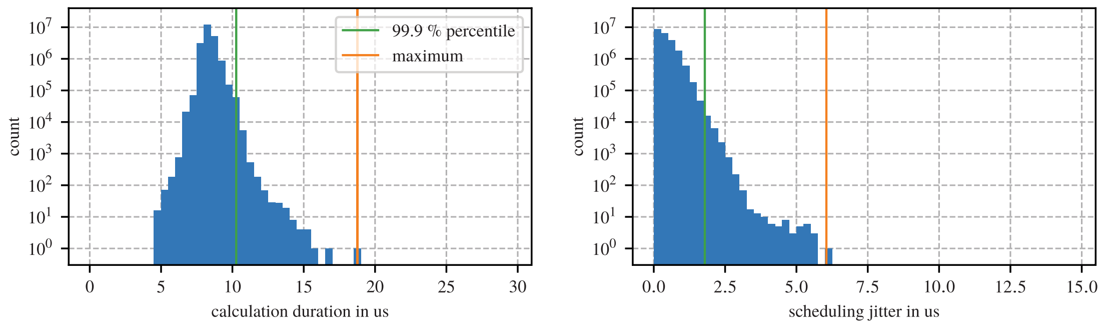
Simulation: The compatibility with simulation tools such as Gazebo is also given as the RTCF correctly handles the `use_sim_time`-parameter.

3. Evaluation

To demonstrate the performance as well as the usability, the example architecture as depicted in Fig. 1 was implemented. All components are internally implemented as a simple sum operation to ensure an evaluation that is decoupled from the actual controller payload.

3.1. Configuration

The example has two loop closures via H1, which means that the incoming connections must be ignored for the dependency resolution. This is reflected by setting the `is_first`-parameter for H1 in the launch file. The three connections between the real-time components and G1/G2 are enabled by setting the `ros_mapping_whitelist`-parameter appropriately. Beyond that, no special configuration is required. The rt_runner will automatically start the control loop including the dependency resolution as soon as all expected components are loaded.



(a) Calculation duration.   (b) Scheduling jitter.

Fig. 2. Performance evaluation of the example scenario running with a frequency of 2 kHz for 3 h.

3.2. *Real-time performance*

The controller architecture was run with a frequency of 2 kHz on an off-the-shelf desktop computer with an Intel Core i5-8600 CPU (kernel version 5.4.44 with Preempt-RT [3]). For reproducible load conditions, *stress-ng* was running in the background with the options `--cpu 48 --io 48`. The scheduling jitter and the calculation duration for each full RTCF controller iteration were captured over 3 h using a built-in performance measurement topic. From the histogram in Fig. 2(a), it can be seen that the 99.9 % quantile of the calculation duration is 10.3 μs, while the maximal duration is 18.8 μs. This means that the proposed framework has low overhead and consistent delays, even for large controller architectures. In Fig. 2(b), the 99.9 % quantile and the maximum of the scheduling jitter are 1.8 μs and 6.1 μs, respectively. This indicates that the Preempt-RT patch and the control loop timing work as intended, allowing control frequencies far beyond 2 kHz.

Listing 1 A minimal working example of an RTCF component.

```cpp
#include <std_msgs/Float32.h>

#include <rtcf/macros.hpp>
#include <rtcf/rtcf_extension.hpp>
OROCOS_HEADERS_BEGIN
#include <rtt/Component.hpp>
#include <rtt/Port.hpp>
#include <rtt/RTT.hpp>
OROCOS_HEADERS_END

class PaperExample : public RTT::TaskContext, public RtcfExtension
{
public:
  PaperExample(std::string const &name) :
      TaskContext(name), port_out_("out_port"), port_in_("in_port") {}

  bool configureHook()
  {
    this->ports()->addPort(port_in_);
    this->ports()->addPort(port_out_);
    double param = this->getNodeHandle().param("/test_param", 0.0);
    NON_RT_INFO_STREAM("Fetched param with value " << param);
    return true;
  }
  void updateHook()
  {
    RT_INFO("Update hook called!");
    port_in_.read(msg_);
    port_out_.write(msg_);
  }

  bool startHook() { return true; }
  void stopHook() {}
  void cleanupHook() {}

private:
  RTT::OutputPort<std_msgs::Float32> port_out_;
  RTT::InputPort<std_msgs::Float32> port_in_;
  std_msgs::Float32 msg_;
};

ORO_CREATE_COMPONENT(PaperExample)
```

4. Impact

To the best of our knowledge, the RTCF is the first framework that enables high-performance control on general-purpose computers in combination with broad compatibility with ROS and outstanding modularity. Today, this already increases the speed and the flexibility of robotic control research and development. For example, researchers often use custom embedded systems for their low-level controllers and hardware interfaces. The need for these is largely eliminated with the arrival of the RTCF, especially if platforms such as the Raspberry Pi are taken into account. In the case of ROS, the availability of a unified framework has led to a large ecosystem of packages for standard tasks. Similarly, we see this potential in the RTCF in the long run.

Listing 2 Example of a ROS-compatible launch file with two real-time components that are loaded into a rt_runner and executed with 1 kHz.

```xml
<launch>
    <node name="rt_runner" pkg="rtcf" type="rt_runner" output="screen">
        <rosparam param="mode">"wait_for_components"</rosparam>
        <rosparam param="ros_mapping_whitelist">".*ros.*"</rosparam>
        <rosparam param="num_components_expected">2</rosparam>
        <rosparam param="frequency">1000.0</rosparam>
    </node>
    <node name="example1" pkg="rtcf" type="rt_launcher"
    ↪   args="rtcf_examples PaperExample">
        <remap from="out_port" to="/tmp"/>
        <remap from="in_port" to="/ros/in"/>
        <rosparam param="is_first">True</rosparam>
    </node>
    <node name="example2" pkg="rtcf" type="rt_launcher"
    ↪   args="rtcf_examples PaperExample">
        <remap from="out_port" to="/ros/out"/>
        <remap from="in_port" to="/tmp"/>
    </node>
</launch>
```

The first complex deployment will take place during the renovation of the ISAS semi-mobile haptic interface [14]. In this case, the aim is to replace the outdated and hard-to-maintain embedded system with a flexible research platform without compromising the overall control performance. Furthermore, the university group KITcar [15], which is active in the field of autonomous driving, has signaled its interest in using the RTCF as a substitute for its current control framework.

5. Outlook

Although the RTCF is already in a mature state some limitations still exist. For example, all controllers need to run with the same frequency and multiple instances of the rt_runner are currently not supported. In addition, all components are executed sequentially even if parallel paths in the controller architecture exist. Another limitation is that only Preempt-RT is supported for the execution in real-time at the moment.

Future work will deal with these issues and examine how the presented solution can be transitioned towards ROS 2. Beyond that, we intend to incorporate user feedback to make the RTCF even more versatile.

Illustrative examples

See Listings 1 and 2.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Ng, ROS: An open-source robot operating system, in: ICRA Workshop on Open Source Software, vol. 3, 2009.

[2] Xenomai Wiki, 2021, https://source.denx.de/Xenomai/xenomai/-/wikis/home. (Accessed 28 June 2021).

[3] Real-Time Linux Wiki, 2021, https://rt.wiki.kernel.org/index.php/Main_Page. (Accessed 28 June 2021).

[4] S. Chitta, E. Marder-Eppstein, W. Meeussen, V. Pradeep, A. Rodríguez Tsouroukdissian, J. Bohren, D. Coleman, B. Magyar, G. Raiola, M. Lüdtke, E. Fernández Perdomo, ros_control: A generic and simple control framework for ROS, J. Open Source Softw. (2017) http://dx.doi.org/10.21105/joss.00456.

[5] H. Bruyninckx, Open robot control software: the OROCOS project, in: Proceedings of 2001 IEEE International Conference on Robotics and Automation, vol. 3, 2001, pp. 2523–2528, http://dx.doi.org/10.1109/ROBOT.2001.933002.

[6] Orocos RTT / ROS integration packages, 2021, https://github.com/orocos/rtt_ros_integration. (Accessed 28 June 2021).

[7] ROS. 2 Documentation: Galactic, Real-time programming in ROS 2, 2021, https://docs.ros.org/en/galactic/Tutorials/Real-Time-Programming.html. (Accessed 28 June 2021).

[8] H. Wei, Z. Shao, Z. Huang, R. Chen, Y. Guan, J. Tan, Z. Shao, RT-ROS: A real-time ROS architecture on multi-core processors, Future Gener. Comput. Syst. 56 (2016) 171–178, http://dx.doi.org/10.1016/j.future.2015.05.008.

[9] R. Delgado, B.-J. You, B.W. Choi, Real-time control architecture based on Xenomai using ROS packages for a service robot, J. Syst. Softw. 151 (2019) 8–19, http://dx.doi.org/10.1016/j.jss.2019.01.052.

[10] micro-ROS, 2021, https://github.com/micro-ROS. (Accessed 28 June 2021).

[11] A.B. Kahn, Topological sorting of large networks, Commun. ACM 5 (11) (1962) 558–562, http://dx.doi.org/10.1145/368996.369025.

[12] ROS Wiki, nodelet, 2021, http://wiki.ros.org/nodelet. (Accessed 28 June 2021).

[13] ROS Wiki, dynamic_reconfigure, 2021, http://wiki.ros.org/dynamic_reconfigure. (Accessed 28 June 2021).

[14] P. Rößler, T. Armstrong, O. Hessel, M. Mende, U. Hanebeck, A novel haptic interface for free locomotion in extended range telepresence scenarios, in: Proceedings of the Third International Conference on Informatics in Control, Automation and Robotics, 2006, pp. 148–153, http://dx.doi.org/10.5220/0001214101480153.

[15] KITcar: Cognitive autonomous racing, 2021, https://kitcar-team.de/. (Accessed 28 June 2021).