Ye et al.
Chinese Journal of Mechanical Engineering (2023) 36:144
https://doi.org/10.1186/s10033-023-00976-5

Chinese Journal of Mechanical Engineering

ORIGINAL ARTICLE

Open Access

# ROS2 Real-time Performance Optimization and Evaluation

Yanlei Ye[1], Zhenguo Nie[1,2], Xinjun Liu[1,2*], Fugui Xie[1,2], Zihao Li[1] and Peng Li[1]

**Abstract**

Real-time interaction with uncertain and dynamic environments is essential for robotic systems to achieve functions such as visual perception, force interaction, spatial obstacle avoidance, and motion planning. To ensure the reliability and determinism of system execution, a flexible real-time control system architecture and interaction algorithm are required. The ROS framework was designed to improve the reusability of robotic software development by providing a distributed structure, hardware abstraction, message-passing mechanism, and application prototypes. Rich ecosystems for robotic development have been built around ROS1 and ROS2 architectures based on the Linux system. However, because of the fairness scheduling principle of the default Linux system design and the complexity of the kernel, the system does not have real-time computing. To achieve a balance between real-time and non-real-time computing, this paper uses the transmission mechanism of ROS2, combines it with the scheduling mechanism of the Linux operating system, and uses Preempt_RT to enhance the real-time computing of ROS1 and ROS2. The real-time performance evaluation of ROS1 and ROS2 is conducted from multiple perspectives, including throughput, transmission mode, QoS service quality, frequency, number of subscription nodes and EtherCAT master. This paper makes two significant contributions: firstly, it employs Preempt_RT to optimize the native ROS2 system, effectively enhancing the real-time performance of native ROS2 message transmission; secondly, it conducts a comprehensive evaluation of the real-time performance of both native and optimized ROS2 systems. This comparison elucidates the benefits of the optimized ROS2 architecture regarding real-time performance, with results vividly demonstrated through illustrative figures.

**Keywords** ROS, Real-time system optimization, Preempt_RT, Real-time performance evaluation of ROS2

## 1 Introduction

Developing a ROS2 control system requires careful attention to real-time performance design and assurance. Industrial robots, aerospace equipment, medical robots, service robots, and military robots all impose strict real-time constraints. A real-time system is one that responds to events occurring in the environment within precise timing intervals [1]. Hence, optimizing and evaluating the real-time performance of ROS2 is crucial, as it determines the system's usability for researchers and engineers and how to better utilize ROS2 [2] for related research.

Numerous software concepts and architectures have been proposed in response to the difficulties of developing software for complex robot systems. In recent years, component-based and model-driven development have gradually been introduced into the construction of robot software systems to simplify development and improve quality. Modern robot control systems are typically designed as component-based distributed systems. Examples of well-known systems that use this approach include OROCOS [3], OpenHRP [4], YARP [5, 6], MRDS [7], Director [8] and ROS [9–13]. They all share the idea that complex robot systems should be composed

*Correspondence:
Xinjun Liu
xinjunliu@mail.tsinghua.edu.cn
[1] State Key Laboratory of Tribology in Advanced Equipment, Department of Mechanical Engineering, Tsinghua University, Beijing 100084, China
[2] Beijing Key Lab of Precision/Ultra-Precision Manufacturing Equipments and Control, Tsinghua University, Beijing 100084, China

of software engineering interaction modules based on components.

The robot operating system (ROS) has become popular among researchers and engineers due to its streamlined, message-based, and tool-based design. However, its non-real-time system architecture prevents it from guaranteeing fault tolerance, deadlines, or process synchronization. Karamousadakis et al. [14] designed a quadruped robot based on the ROS1 system architecture using Xenomai patches to optimize the native system. Despite this improvement, ROS still requires significant resources, including CPU, memory, network bandwidth, threads, and kernels. It cannot manage these resources to meet time constraints effectively.

The real-time robot operating system (RT-ROS) [15] creates a non-real-time/real-time task execution environment using the Linux and Nuttx kernels. This improves the real-time performance of ROS, but it does not guarantee real-time constraints for ROS. Using RT-ROS requires modifications to the ROS library and nodes, making it difficult to quickly update and maintain. MICRO-ROS [16] is a variant developed specifically for resource-limited microcontrollers, which is a lightweight ROS client that can run on modern 32-bit microcontrollers like STM32. However, deploying projects on microprocessors for dual-arm robots or large engineering projects is challenging due to limited resources and computing power.

As the demand for translating research results into commercial products becomes urgent, the limitations of ROS1 as a fundamental research platform are becoming apparent, as it was not designed with the needs of real-time systems, small embedded platforms, non-ideal networks, cross platform compatibility, and commercial productization in mind. ROS2, which uses the data distribution service (DDS) [17, 18] for communication, can improve the real-time performance of message passing [19, 20], but this improvement is only targeted at the latency between nodes (usually considered to be several hundred milliseconds). Ding et al. [21] systematically introduced the architecture of the ROS2 system and were among the first to analyze the source code of ROS2. Maruyama et al. [12] have explored the important real-time performance of ROS2 on the native kernel, evaluating the real-time performance of ROS2 relative to ROS1 from multiple perspectives. Choi [22] proposed a priority-driven chain-aware scheduler to optimize the real-time performance of ROS2 from a scheduling strategy perspective, improving end-to-end latency. ROS2.0 itself is built on DDS and some modules to construct distributed and real-time solutions. However, most of the ROS2 ecosystem is currently built around Linux, and the upper limit of real-time performance is determined by the operating system itself. Commonly, ROS2 is built on Ubuntu, which cannot guarantee the real-time performance of the system (such as a robot communication cycle of 1ms with jitter below 200 μs). When the robot's trajectory is finely interpolated and the system cannot deliver data on time, the robot's joint motion becomes less smooth. Therefore, it is urgent to carry out real-time performance analysis under the ROS2 architecture and improve the real-time performance of the system.

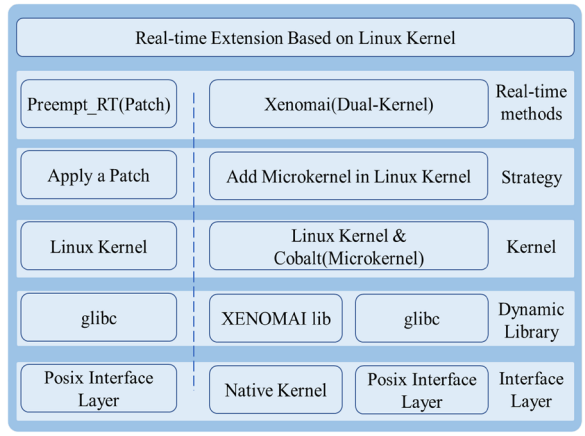


**Figure 1** Real-time extension methods based on the Linux kernel

Currently, several popular commercial real-time systems include QNX Neutrino, ENEA OSE, Integrity, VxWorks, and Windows CE [23–26]. In addition, many open-source real-time systems, including CHAOS, MARS, Spring, ARTS, RK, TIMIX, MARUTI, HARTOS, YARTOS, HARTIK, Erika Enterprise, Shark, Marte OS, RTLinux, and FreeRTOS, are commonly used to handle real-time tasks for single-core and single-task scenarios [1, 27, 28]. However, their capabilities for handling multi-core tasks and compatibility with non-real-time applications are weaker.

Linux is a popular choice among researchers and businesses due to its open-source nature, stability, reliability, fast-update environment, and large community. To leverage the powerful Linux ecosystem, which includes drivers, desktop and human-computer interaction interfaces, and to ensure compatibility with the ROS architecture, modifications to the Linux kernel are required to achieve real-time performance. Two approaches are typically available: the dual-kernel approach (also known as PICO-KERNEL, NANO-KERNEL, DUAL KERNEL) and the real-time patch approach, as shown in Figure 1. The dual-kernel approach includes Xenomai [29, 30] and RTAI [1, 31], while the real-time patch approach includes Preempt_RT [32] (Linux Real-time Patch, Linux Configuration). To maintain a flexible architecture design and minimize changes to the original system code, this article

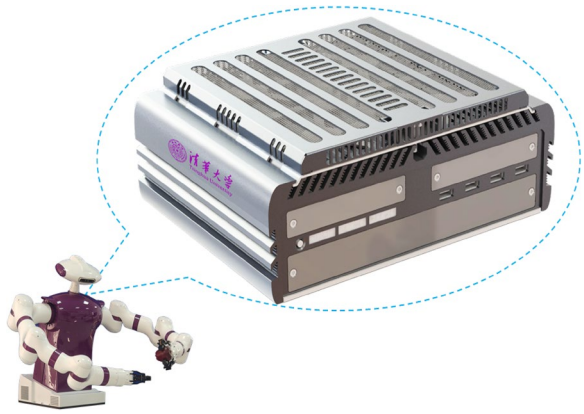

**Figure 2** PC-based control platform

utilizes the Preempt_RT patch approach to optimize the real-time performance of the ROS2 architecture.

This article presents a comprehensive evaluation of the real-time performance of ROS1 and ROS2 data transmission on a Preempt_RT optimized real-time system, which outperforms the native system. The real-time performance of ROS1 and ROS2 is compared from multiple perspectives, including throughput, control frequency, and multi-node subscription. Section 2 introduces the software and hardware operating environment of the system, while Section 3 explains the real-time performance optimization based on Preempt_RT. Section 4 conducts a rigorous evaluation of the real-time performance. Finally, a summary of the results are presented in the last section. This study provides valuable insights for improving the real-time performance of ROS2 systems.

## 2 System Setup

The TH-Dual-Arm robot, developed by the Advanced Mechanism and Roboticized Equipment Lab at Tsinghua University, was utilized as the subject of this study. The control hardware architecture was implemented based on a PC, as depicted in Figure 2. When designing the controller hardware, the requirements for system computing power and storage, as well as the need for platform scalability, universality, and standardization, were taken into account. Table 1 shows some of the software used, while Table 2 lists the hardware. The system utilizes the EtherCAT bus communication protocol. It should be noted that this paper does not analyze the motion performance of the control system but only conducts real-time performance optimization and evaluation under this configuration.

The relevant components and software configurations are shown in Table 1 and Table 2. The Linux system used is Ubuntu 22.04, with a Linux kernel version of 5.15.55 and the Preempt_RT patch applied. The ROS1 version used is Noetic, while the ROS2 version is Humble, which is the latest LTS version supported for the last 5 years.

**Table 1** Software components of the controller

| Item | Description | Version |
|---|---|---|
| Ubuntu | Linux distribution | 22.04 |
| Linux kernel | Linux kernel | 5.15.55 |
| Preempt_RT | Linux kernel patch | 5.15.55-rt48 |
| ROS1 | first-generation robot operating system | Noetic |
| ROS2 | Second-generation robot operating system | Humble |
| EtherCAT master | Industrial Ethernet Fieldbus | acontis |

## 3 Real-time Optimization of ROS2 Based on Preempt_RT

The optimization of the real-time performance of the ROS2 system centers on enhancing the real-time capabilities of the operating system kernel. In this work, we first studied the Xenomai dual-kernel solution. The basic principle of this approach is to run a microkernel and a native Linux kernel simultaneously. Real-time tasks are executed on the microkernel, which takes control of interrupts and directly manages them at the lowest level. When no real-time tasks are running on the microkernel, the Linux kernel can be given an opportunity to run. Xenomai achieves real-time capabilities by running the real-time Cobalt kernel in parallel with the Linux kernel, as illustrated in Figure 3. However, we opted for the Preempt_RT patch approach to optimize the real-time performance of the ROS2 architecture, due to its flexible architecture design and minimization of changes to the original system code.

The Cobalt microkernel manages critical timing activities, such as interrupt handling and scheduling of real-time threads. The Cobalt kernel has a higher priority than the native kernel, and the key to enhancing real-time performance lies in the Adaptive Domain Environment for

**Table 2** Controller hardware system configuration

| Item | Description | Quantity |
|---|---|---|
| Motherboard | Mini ITX motherboard SD103-H110 by Taiwanese manufacturer DFI | 1 |
| CPU | Intel i7 7700 4 cores 3.6 GHz | 1 |
| Solid State Drive | 256 G | 1 |
| RAM | DDR4-3200 32 GB | 2 |
| Network Interface Controller | Intel I211(1 Gbit/s) | 2 |
| Network Interface Controller | Intel I219(1 Gbit/s) | 2 |
| Power Supply | DC 24V | 1 |

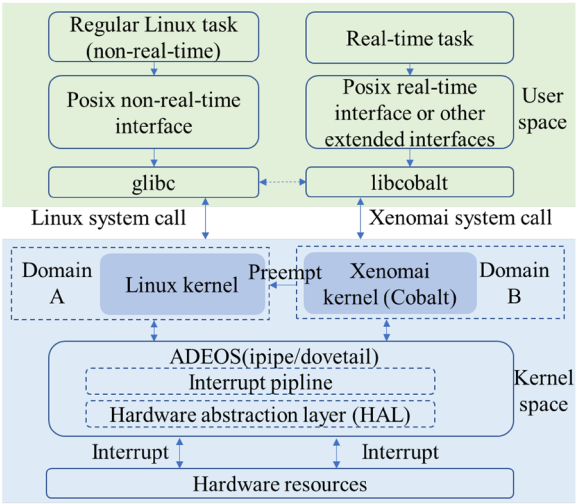


**Figure 3** Xenomai Cobalt kernel architecture

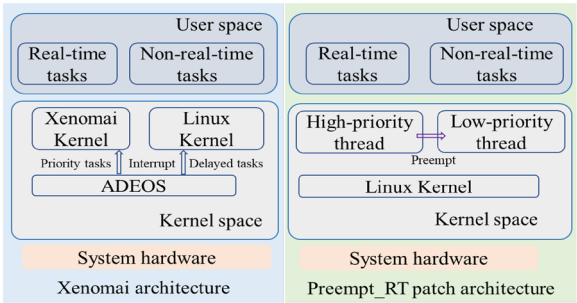


**Figure 4** Architecture of Xenomai and Preempt_RT

**Table 3** Kernel optimization

| Item | Description |
|---|---|
| Preemption model | Fully Preemptible Kernel (Real-Time) |
| Timers' subsystem | High-Resolution Timer Support |
| Timer tick handling | Full dynticks system (tickless) |
| Timer frequency | 1000 Hz |
| Default CPUFreq governor | Performance |
| C-state | Forbid |

Operating Systems (ADEOS). ADEOS enables the sharing of common hardware resources among multiple identical or different kernels on the same system. In ADEOS, the Interrupt Pipeline (I-PIPE) manages and distributes interrupts between Linux and Xenomai, passing them in domain priority order. For registered interrupts in the real-time kernel, direct processing is ensured immediately after their generation, guaranteeing the real-time performance of the system. For interrupts generated by Linux, they are recorded first and then processed only after the real-time task yields the CPU.

To optimize the real-time performance of the native Linux kernel and fully utilize the rich software of the Ubuntu system, this paper uses the Preempt_RT patch. Preempt_RT optimizes the native macro kernel by minimizing the code of non-preemptible kernels and the number of code changes implemented to achieve preemption. In particular, the critical section, interrupt handler, and interrupt disable code sequence are modified to make this section preemptible. The Preempt_RT patch fully utilizes the Symmetrical Multi-Processing (SMP) function of the Linux kernel to add this additional preemption without rewriting the kernel, as shown in Figure 4.

The Preempt_RT patch provides functions such as preemptible critical sections, preemptible interrupt handlers, preemptible "interrupt disable" code sequences, kernel spinlocks, and semaphore priority inheritance, as well as measures to reduce latency. Modifications to the native kernel include high-precision timers, thread interrupt handlers, sleep spinlocks, real-time mutexes, and RCU synchronization mechanisms. To evaluate the performance of the Preempt_RT patch, we installed Ubuntu 22.04 on an Intel x86_64 system with kernel version 5.15.55-generic, applied the Preempt_RT patch (patch-5.15.55-rt48.patch.gz), and optimized Table 3. Some visual modules were trimmed.

To achieve high accuracy timing in the nanosecond range, the clock_gettime(CLOCK_MONOTONIC, &ts_now) function can be utilized. For timed latency requiring precise timing, the clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &ts_nest, NULL) function is recommended.

For scheduling policies in publish-subscribe, client-server, and action-client-action-server designs, we use the CFS scheduler for non-critical nodes in this paper. The CFS scheduler implements scheduling using a red-black tree to adjust running times based on time slices and virtual time, as shown in Eqs. (1) and (2):

$$ime\_slice\_I = sched\_period \times \frac{weight\_i}{weight\_pq'}, \tag{1}$$

$$vruntime\_i = vruntime\_i + \frac{weight\_nice0}{weight\_i} \times real\_runtime. \tag{2}$$

For real-time nodes in the controller design, we use the SCHED_FIFO scheduling policy of the RT scheduler for control. The SCHED_FIFO scheduling policy schedules system tasks using a multi-level priority queue. Among tasks with the same priority, the real-time task based on SCHED_FIFO will execute until completion, relinquish

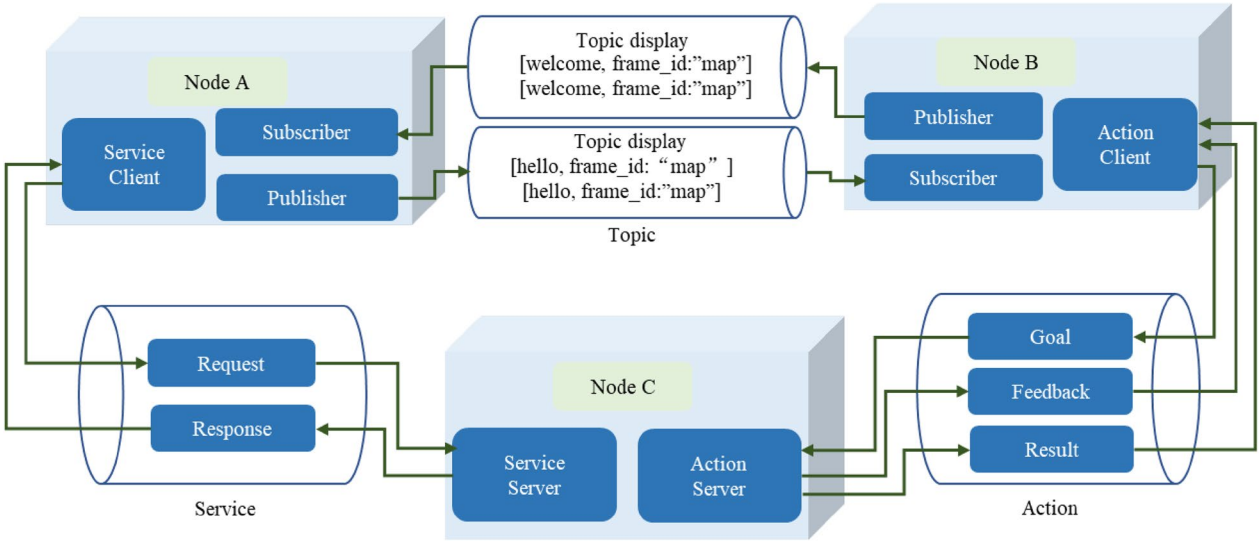


**Figure 5** Node data transfer diagram of ROS2

control voluntarily, or be preempted by a task with a higher priority.

## 4 Real-time Performance Evaluation of ROS2

This study aims to ensure the stability of the control system architecture during operation by maintaining real-time performance across different frequencies and loads. The analysis focuses on the jitter and latency caused by various factors, including frequency, data size, Quality of Service (QoS), and Data Distribution Service (DDS), in both native systems and ROS1 and ROS2 systems optimized using Preempt_RT. Specifically, we investigate the latency characteristics of ROS1 and ROS2 and attempt to identify differences in their performance. The study explores the end-to-end latency of individual nodes as well as the subscription latency of multiple nodes.

Nodes can exchange data through topics, services, and actions, as depicted in Figure 5. Each of these communication methods has its own message structure, which can be nested to enable the exchange of complex data between nodes. Moreover, each node can perform multiple roles, and subscribers can be asynchronously awakened to perform computations. Actions are commonly used in controller design for real-time feedback and execution status computation. ROS2's distinct feature is its decoupling of computation, which facilitates distributed node computing.

### 4.1 Latency Evaluation Method

Cyclictest accurately and repeatedly measures the difference between the expected and actual wakeup times of threads, providing statistical information on system latency. It can measure system latency caused by hardware, firmware, and the operating system, and is commonly used to test the latency of kernel usage to assess real-time kernel performance. The latency measured by Cyclictest refers to interrupt and scheduling delays, as shown in Figure 6, where interrupt delay refers to the latency between the occurrence of an interrupt and the start of the interrupt service routine (ISR), and scheduling latency refers to the time it takes for a task to obtain actual CPU usage after being awakened.

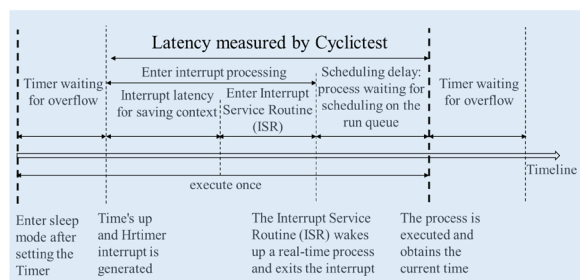


**Figure 6** Measured latency time

To test the real-time performance of the kernel, multiple real-time threads with specified priorities are created in the Master thread, and each real-time thread sets a Timer to periodically wake itself up. When the Timer overflows, an interrupt is generated, and the system enters the interrupt handler. The ISR calls wake_up_process() to wake up the real-time process, and the scheduler performs scheduling and dispatching. The total latency time includes the interrupt handling time and scheduling latency. At the beginning of each loop, the current

```
clock_gettime(&now) //Obtain the current time
next=now+par->interval//Calculate the value of the next period
while(!shutdown)
{
   clock_nanosleep(&next);//Sleep until the next time point
   clock_gettime(&now);//Obtain the value after sleeping
   diff=calcdiff(now,next);//Calculate the delay value
   //Update the minimum value state->min, maximum value state-
>max, total latency state->total latency, cycle period cycles, etc.
   next+=interval; //Calculate the value of the next period
}
```

**Figure 7** Calculating periodic latency

time is calculated, and the value is passed to the Master thread through shared memory for statistics and output. In the while loop, the interval is slept for a few microseconds before waking up and obtaining the current time to calculate the latency time repeatedly. The relevant code snippet is shown in Figure 7.

### 4.2 Real-time Performance of Native-Linux Kernel and Preempt_R-Linux

The present study first evaluated the real-time performance of the native Linux kernel and the kernel optimized with the Preempt_RT patch. For ease of writing, the native Linux kernel is abbreviated as "Native-Linux," while the kernel optimized with the Preempt_RT patch is abbreviated as "Preempt_RT-Linux." Loading tests were performed in the experiment, with Fourier transforms running on four CPUs to bring CPU usage to near 100% (stress-ng -c 4 --cpu-method fft --timerfd-freq 1000000 -t 24h &), as shown in Figure 8. For the Native-Linux system, the test took 242.198 s, with a maximum latency of 6243 μs and an average latency of 3 μs, as shown in Figure 9. This is inadequate for high-precision motion equipment and robots, as the timing jitter for a control cycle of 1 ms is usually required to be less than 200 μs. Similarly, for the optimized Preempt_RT-Linux system, five real-time threads were launched with frequencies ranging from 1000 to 3000 Hz, and a maximum latency of 82 μs and an average delay of 2 μs were observed during the 25.7 h test, as shown in Figure 10.

The comparison between Native-Linux and Preempt_RT-Linux is shown in Table 4. The real-time performance of the optimized Preempt_RT-Linux has been significantly improved. Compared to Native-Linux, Preempt_RT-Linux has smaller minimum and average latency values, and notably, the maximum latency value has significantly decreased.

### 4.3 Real-time Performance Evaluation of ROS1 and ROS2 under Different Data Sizes

The paper discusses the end-to-end latency between publishers and subscribers, with data sizes ranging from 64 bytes to 16 megabytes, using string-type messages for evaluation. The study evaluates the latency characteristics of ROS1 and ROS2. Table 4 lists the hardware and software environment used to measure the latency from the timing publish function of a single publishing node to the callback function of another

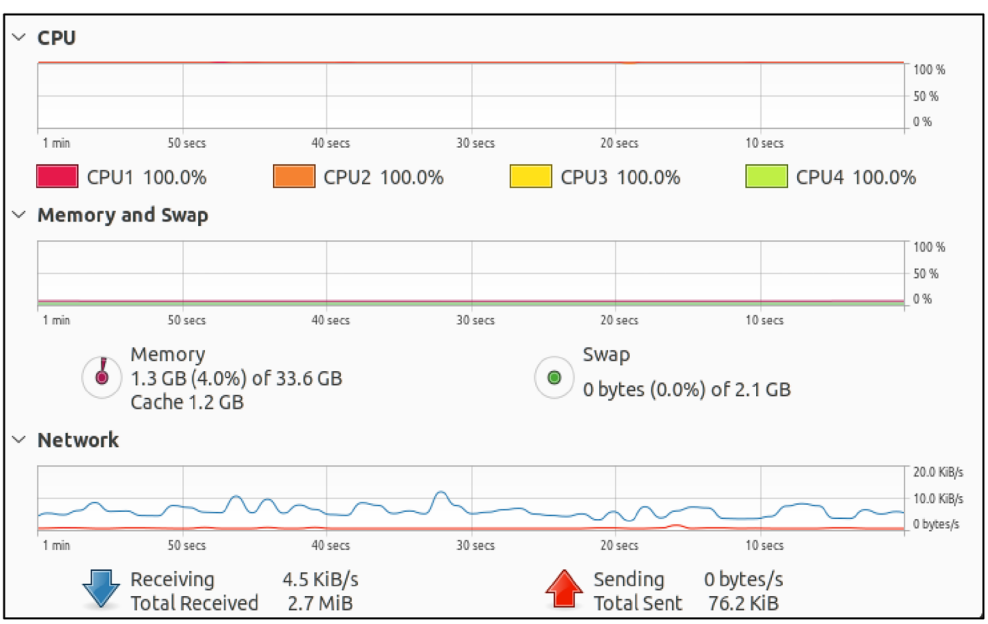


**Figure 8** CPU load status

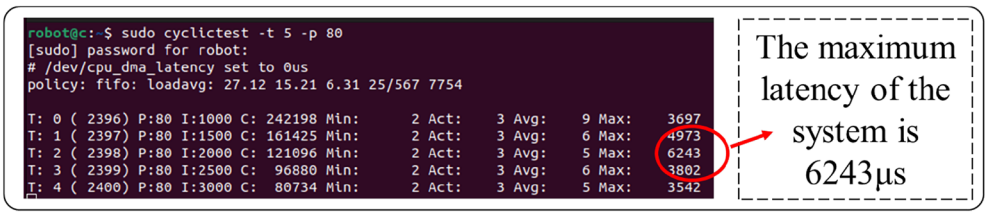


**Figure 9** Timing latency of the native Linux kernel system

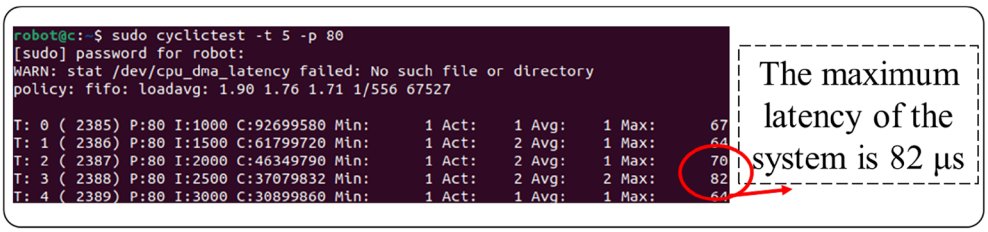


**Figure 10** Real-time performance of Preempt_RT-Linux

subscribing node on the same computer, as illustrated in Figure 11. The nodes are executed at a frequency of 10 Hz, and data of different sizes are evaluated 120 times. Line graphs and the median latency for each group of data are obtained.

ROS1 uses TCPROS for reliable communication, while the corresponding QoS reliable policy is used in ROS2 architecture. Fast DDS is used as the DDS middleware, which is released under the LGPL license. To accurately measure real-time performance, the node design follows the SCHED_FIFO scheduling policy and uses mlockall for memory locking. SCHED_FIFO processes have priority over CFS processes (which are usually used with no specified real-time processes and use the default Linux scheduling policy). The purpose of mlockall is to fix the process's virtual address space in physical RAM, preventing memory from being paged to the swap area and reducing the latency caused by memory allocation. In ROS2, the QoS policy queue size for publishers and subscribers is 100, the history is "keep history", the reliability is "reliable", the persistence is "volatile", and the liveliness, deadline, lifespan, and lease duration are all set to "system default".

Figure 12 illustrates the real-time performance of ROS1 and ROS2 on Native-Linux and Preempt_RT-Linux. The results indicate that Preempt_RT-Linux optimization leads to better real-time performance compared to Native-Linux. Additionally, the curves show that as data size increases (e.g., data size exceeding 512K bytes), the real-time performance of ROS2 outperforms ROS1, mainly because DDS is used as the transmission method in ROS2. However, as data size

**Table 4** Comparison of real-time performance between Native-Linux and PREEMPT-RT-Linux

| Item | Period (μs) | Native-Linux (μs) | PREEMPT-RT-Linux (μs) |
|---|---|---|---|
| Minimum | 1000 | 2 | 1 |
| | 1500 | 2 | 1 |
| | 2000 | 2 | 1 |
| | 2500 | 2 | 1 |
| | 3000 | 2 | 1 |
| Maximum | 1000 | 3697 | 67 |
| | 1500 | 4973 | 64 |
| | 2000 | 6243 | 70 |
| | 2500 | 3802 | 82 |
| | 3000 | 3542 | 64 |
| Average | 1000 | 3 | 1 |
| | 1500 | 3 | 2 |
| | 2000 | 3 | 2 |
| | 2500 | 3 | 2 |
| | 3000 | 3 | 1 |

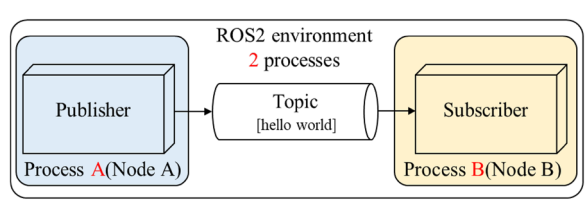


**Figure 11** Inter-process node message transmission and reception

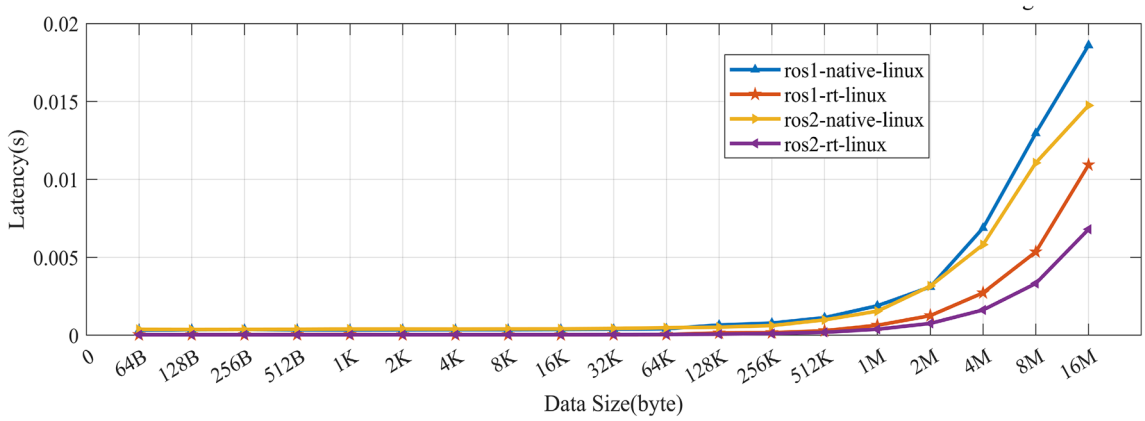


**Figure 12** Comparison of real-time performance of ROS1 and ROS2 before and after optimization

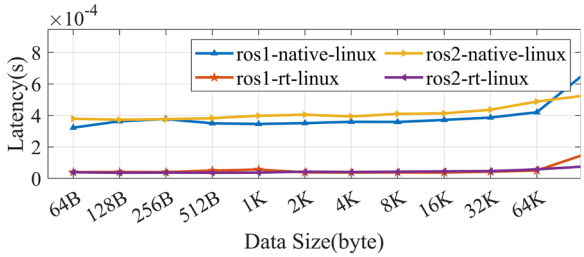


**Figure 13** Real-time performance comparison of ROS1 and ROS2 for small data sizes

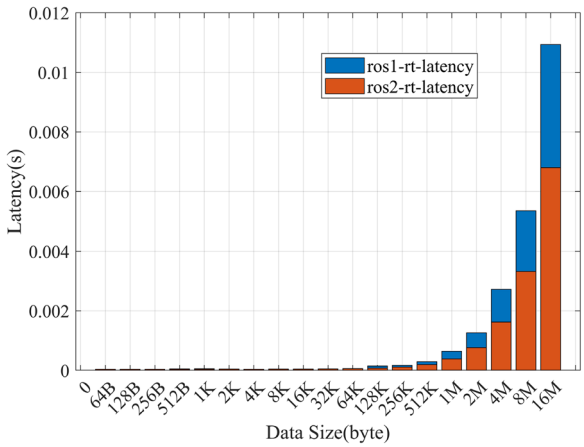


**Figure 14** Comparison of ROS1 and ROS2 real-Time performance bar chart

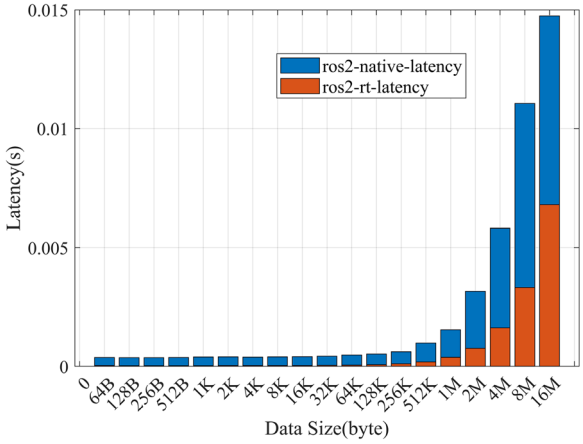


**Figure 15** Comparison of the real-time performance of optimized ROS2

increases, the latency also increases due to the impact of message conversion and DDS processing. DDS has a more significant impact on larger data size transmission. For ROS2 message transmission, two message conversions are required between ROS2 and DDS, with the first conversion from ROS2 to DDS and the second conversion from DDS to ROS2. These conversions consume time, and between them, ROS2 calls the DDS API and sends the message to DDS.

When transmitting small-sized data (ranging from 64 bytes to 64K bytes) in the experiment, the real-time performance of ROS1 and ROS2 was comparable before optimization, and remained so after optimization. However, as shown in Figure 13, the real-time performance of the ROS2 system optimized with Preempt_RT was superior to that of the native ROS2 system. For small data transfers, the conversion and transmission time between nodes and interfaces is relatively small, so the latency remains essentially constant based on the curve observed.

Furthermore, as shown in the bar graph in Figure 14, it can be seen that Preempt_RT-Linux-ROS2 has better real-time performance than Preempt_RT-ROS1 in the case of large data transmission.

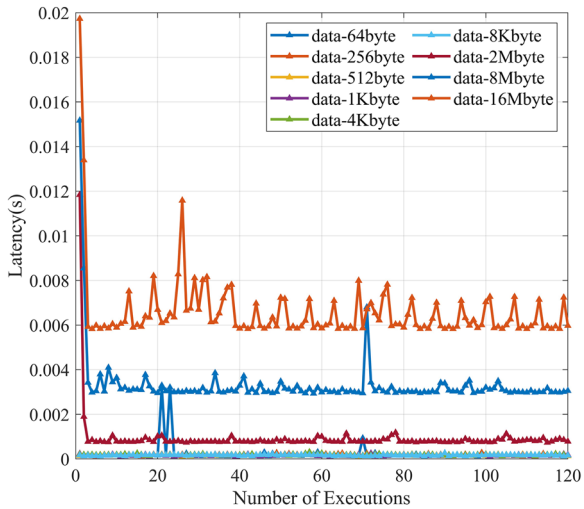


**Figure 16** Real-time performance of Native-Linux-ROS2 with different packet sizes

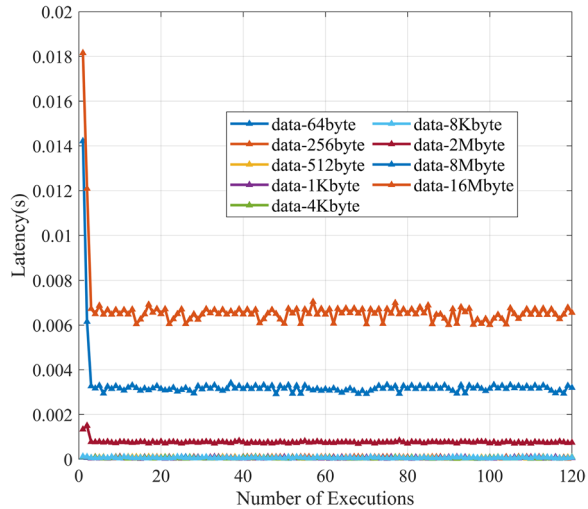


**Figure 18** Real-time performance of Preempt_RT-Linux-ROS2 with different sizes

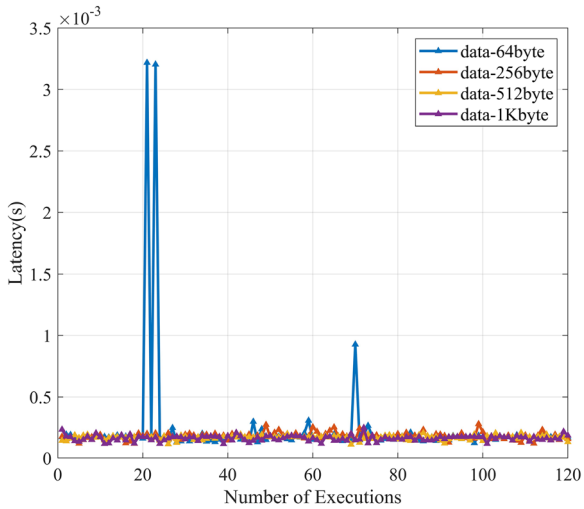


**Figure 17** Real-time performance of Native-Linux-ROS2 on small-scale Data

Figure 15 provides clear evidence that Preempt_RT-Linux-ROS2 outperforms Preempt_RT-ROS1 in real-time performance, particularly when dealing with large data transfers. In fact, as data transfer size increases, the superiority of Preempt_RT-Linux-ROS2 over ROS1 becomes even more pronounced.

Further analysis of the latency of each execution shows that Native-Linux-ROS2 has larger latency fluctuations for different data sizes, as shown in Figure 16. The larger the size of the message-passing data, the more pronounced the latency fluctuations.

Examining small-sized data reveals that, for Native-Linux-ROS2, a smaller data size does not necessarily mean less latency, as shown in Figure 17, where a data size of 64 bytes resulted in a latency of more than 800 μs. Latency also depends on the real-time capabilities of the operating system.

After optimization, Preempt_RT-Linux-ROS2 has smaller real-time fluctuations, and the maximum latency for different data sizes is much smaller than that of Native-Linux-ROS2, as shown in Figures 18 and 19.

### 4.4 Real-time Performance of Different DDS and QoS

ROS2 is built on top of DDS/RTPS middleware, providing discovery, serialization, and transmission. DDS, as an end-to-end middleware, provides message-passing mechanisms and control over different "quality of service" (QoS) options. This section attempts to illustrate the intuitive impact of different DDS and QoS on real-time performance. The study compares eProsima's Fast DDS, Eclipse's Cyclone DDS, and GurumNetworks' GurumDDS, as shown in Figure 20. The curves show that the latency of the different DDS is similar. Specific RMW files and dependencies need to be installed for use. Both C++ and Python nodes support the RMW_IMPLEMENTATION environment variable to select the RMW implementation to be used when running ROS2 applications. This variable can be set to a specific implementation identifier, such as rmw_fastrtps_cpp, rmw_connextdds, or rmw_gurumdds_cpp. For instance, RMW_IMPLEMENTATION=rmw_connextdds ros2 run demo_nodes_cpp talker.

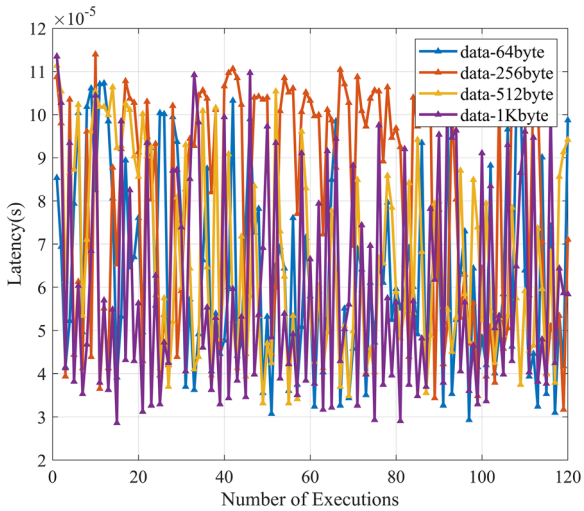


**Figure 19** Real-time performance of Preempt_RT-ROS2 on small-scale data

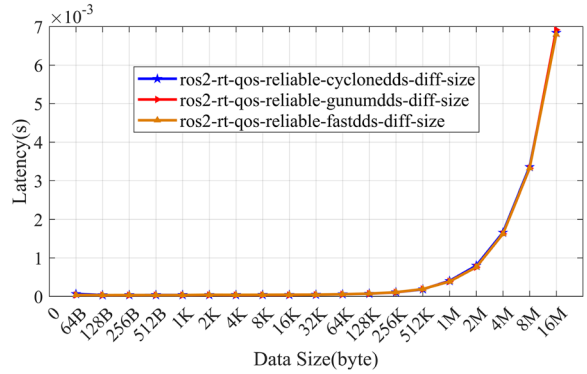


**Figure 20** Real-time performance of different DDS in ROS2

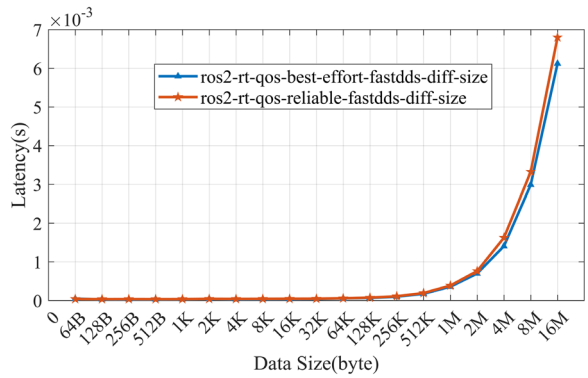


**Figure 21** Real-time performance of different QoS in ROS2

ROS2 provides a rich variety of QoS policies for adjusting communication between nodes. Using the appropriate QoS set, ROS2 can achieve reliable communication, similar to TCP, or best-effort transmission, similar to UDP, and can realize various possible states. Unlike ROS1, which mainly supports TCP communication, ROS2 benefits from the flexibility of the underlying DDS transport. In lossy wireless network environments, the best-effort policy is more suitable. In real-time computing systems, the correct service configuration is needed to meet the final deadline. A set of correct QoS policy combinations form a QoS configuration file. QoS configuration files can be specified for publishers, subscribers, service servers, and clients. QoS configuration files can be applied independently to each instance of the above entities, but if different configuration files are used, they may be incompatible, thus preventing message delivery. Different QoS policies affect the real-time performance of the system. We compared the reliable policy with the best-effort policy using QoS settings. A reliable policy helps ensure reliable communication transmission, while communication in the best-effort policy is unreliable. In the best-effort policy, the subscriber node must be started before the publisher node begins sending messages to avoid "initial value loss." In the test, the subscriber node was started first, followed by the publisher node.

Figure 21 shows the latency under different QoS policies. It can be seen from the figure that for small data sizes, the latency of the best-effort policy and the reliable policy is similar. When the data size increases, the latency of the best-effort policy is smaller than that of the reliable policy. This is because UDP is used in the best-effort policy, while TCP is used in the reliable policy. The QoS history for reliable policy is KEEP_ALL with a depth of 100, and for the best-effort policy, the history is KEEP_LAST with a depth of 1. The specific policy settings are shown in Table 5.

**Table 5** Different QoS policies

| Item | Reliable strategy | Best-effort strategy |
|---|---|---|
| History | KEEP_ALL | KEEP_LAST |
| Depth | 100 | 1 |
| Reliability | Best_effort | Reliable |
| Durability | Transient local | Volatile |
| Deadline | Default | Default |

### 4.5 Real-time Performance of Different Transmission Methods

In design, applications are often composed of individual "nodes" that perform small tasks and are separated from other parts of the system. Such design enables

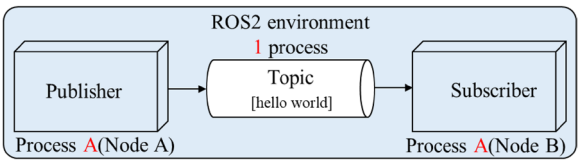


**Figure 22** Inter-process communication through shared memory

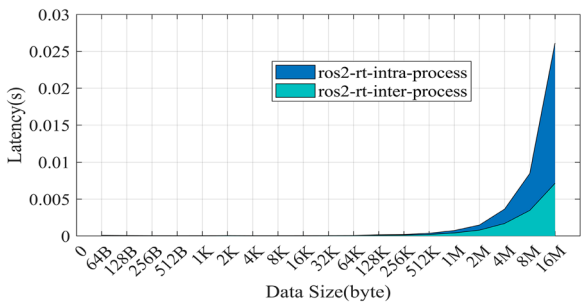


**Figure 23** Real-time performance of different transport methods in ROS2

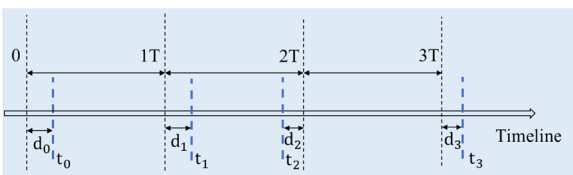


**Figure 24** Periodic execution of tasks

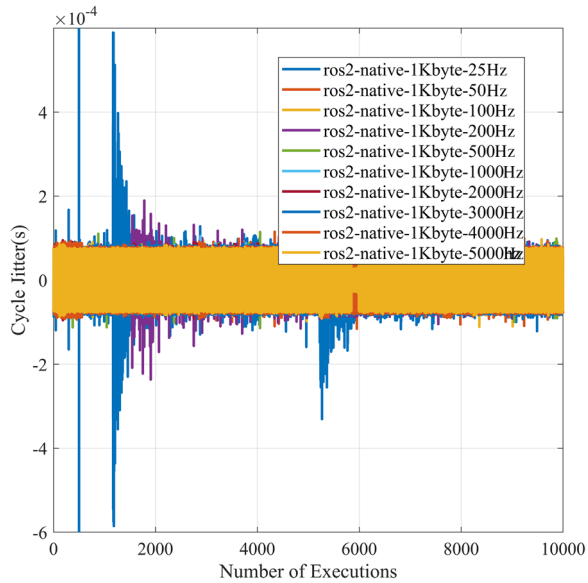


**Figure 25** Periodic jitter at different frequencies on Native-Linux

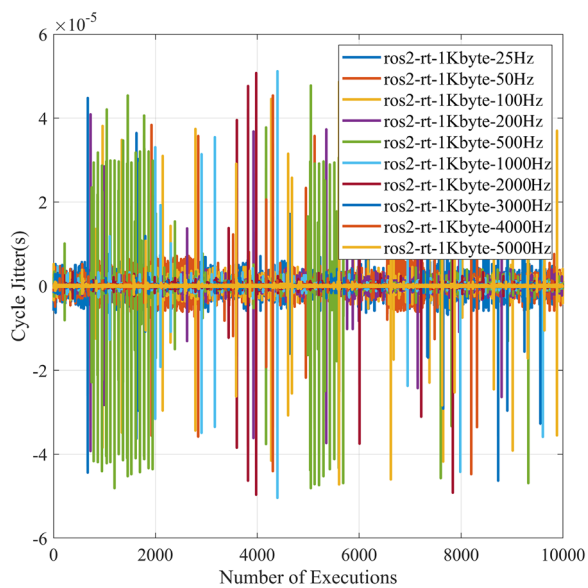


**Figure 26** Different frequency period jitter in Preempt_RT-Linux

fault isolation, faster development, program modularity, and code reuse, but often at the cost of performance. In design, it is also possible to implement multiple nodes within a single process (intra-process), with different nodes implementing message passing, i.e., shared memory transfer, as shown in Figure 22. In this case, DDS is not required. When using std::unique_ptrs for publishing and subscribing, zero-copy message transfer can be achieved through intra-process publish/subscribe connections. DDS requires at least two message translations. The address can be printed to view it: printf("Print out the address of the received message in DDS: 0x%", reinterpret_caststd::uintptr_t(msg.get())). The publishing node and subscribing node have the same address, indicating that the received mail address is the same as the published mail address and not a copy. However, when using const& and std::shared_ptr for publishing and subscribing, multiple copies will be created in this case.

To facilitate the analysis of end-to-end latency characteristics of inter-process transmission (Figure 11) and shared memory transmission (Figure 22), Figure 23 is drawn. The mean latency of each data size is statistically calculated 120 times. For small data sizes, the latency of inter-process and shared memory transmission is similar because the effect of shared memory is hidden by small data sizes. As the data size increases, a significant difference in latency can be observed. Shared memory provides an effective way to transmit large data sizes. It also effectively avoids splitting a message into multiple data packets, reducing end-to-end latency.

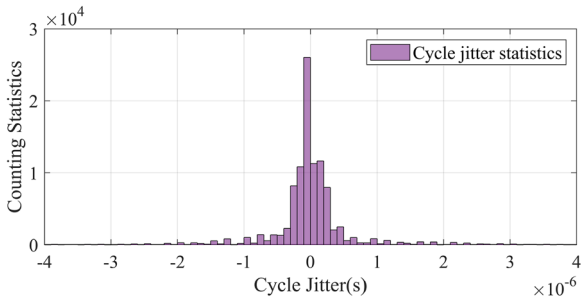


**Figure 27** Histogram-based statistics of periodic jitter

### 4.6 Real-time Characteristics at Different Frequencies

The high and low control frequencies have a significant impact on the effectiveness of control, including trajectory smoothness and computational refinement. This section discusses the impact of frequency on real-time performance. An experiment was conducted to test 10000 cycles, with the horizontal axis representing the number of recordings and the vertical axis indicating cycle jitter. The absolute positioning period, $d_k$, was also measured:

$$d_k = t_k - kT. \tag{3}$$

The cycle jitter, $P_k$ is defined as the deviation between time $k$+1 and time $k$ minus the period $T$, as shown in Figure 24.

$$P_k = t_{k+1} - t_k - T = d_{k+1} - d_k. \tag{4}$$

To better evaluate real-time performance, the CPU was run at full load. Figure 25 shows the cycle jitter for different frequencies on a Native-Linux system, with maximum jitter greater than 400 μs and large fluctuations. The native system is not real-time and cannot be used for multi-axis high-precision motion control.

For the optimized Preempt_RT-Linux system, an analysis was conducted on the cycle jitter for different timing frequencies, which were increased sequentially from 25 to 5000 Hz, with corresponding curves plotted as shown in Figure 26. The jitter fluctuations were smaller, and the maximum cycle jitter was less than 60 μs. The real-time system based on the Preempt_RT patch exhibits good timing performance.

In Figure 27, a histogram of the corresponding cycle jitter is shown, which demonstrates a roughly normal distribution, with the majority of the data points centered around ±10 μs.

### 4.7 Evaluation of Timing Jitter Performance for Multiple Subscribing Nodes

In the previous section, we focused on end-to-end latency between two nodes, analyzed the real-time performance of ROS1 and ROS2, and investigated the impact of different factors on ROS2's real-time performance, such as DDS, QoS, frequency, and throughput. However, in practical applications, there may be a single node publishing messages that are shared and received by multiple nodes. In this section, we conduct further real-time performance analysis by designing one publisher and six subscribers to measure the latency of each receiving node.

Figure 28 shows the latency of ROS1, and it can be observed that there is a significant difference in the latency between the subscribing nodes. Since ROS1 arranges message publishing and receiving in sequence, it is not suitable for real-time systems. For instance, when the data size is 4Mb, the maximum latency of the subscribing node is nearly twice the minimum latency. In contrast, the latency of ROS2 is largely dependent on the packet size, and the latency deviation of all subscribers in ROS2 is small, as shown in Figures 29 and 30. It is evident that the behavior of all subscribers is relatively fair in ROS2. This demonstrates that ROS2 message publishing is fairer for multiple subscribing nodes than ROS1. After real-time optimization, ROS2 shows improved real-time performance for multi-node subscriptions compared to before the optimization.

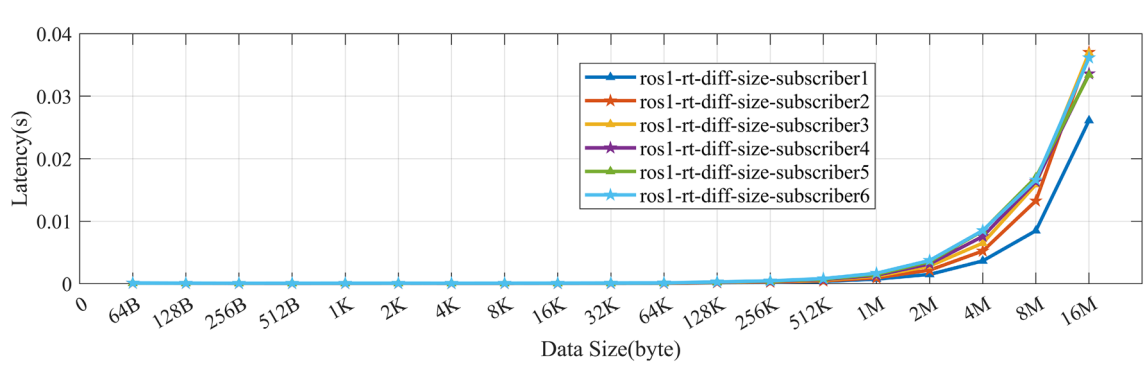


**Figure 28** Real-time performance of multiple subscriber nodes in ROS1

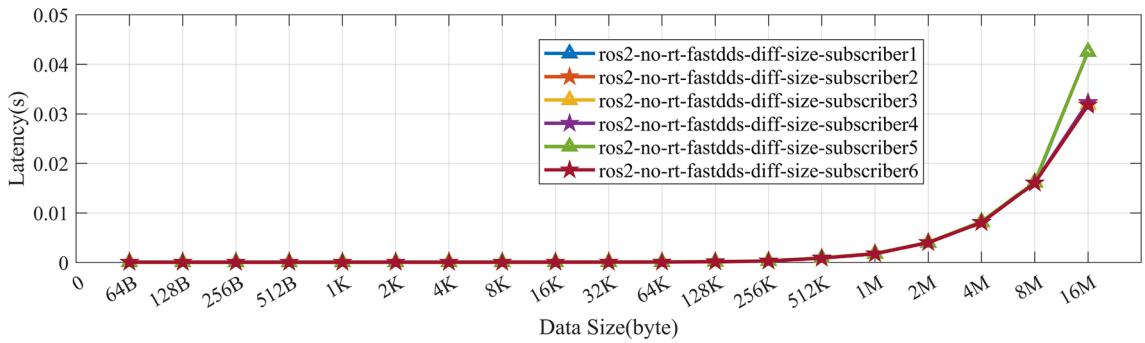


**Figure 29** Real-time performance of multiple subscriber nodes in the native system ROS2

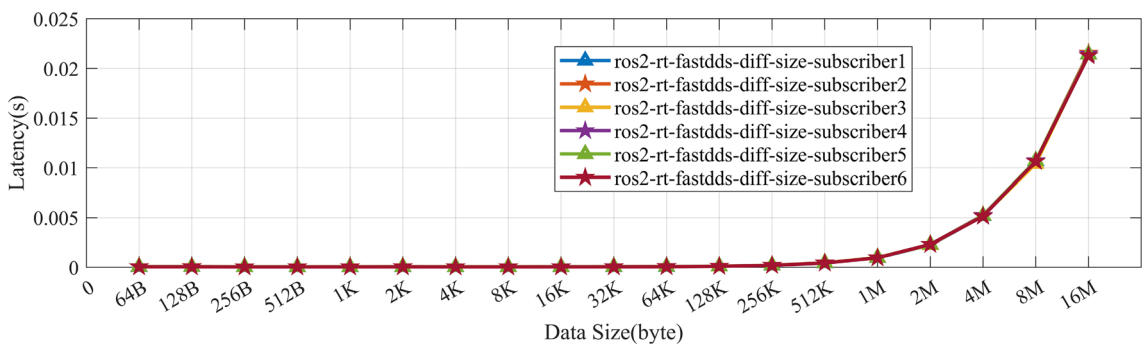


**Figure 30** Real-time performance of the optimized multiple subscriber nodes in ROS2

Furthermore, an in-depth analysis of the latency characteristics of data transmission at various frequencies optimized with Preempt_RT is conducted. Using the default Fast-DDS as the message passing middleware, we measured the data transmission latency of sending and receiving messages at different frequencies with a fixed message size of 1K byte. Each frequency was tested 120 times, and the results were plotted in a 3D graph in Figure 31 and a corresponding curve in Figure 32. It can be observed from the figures that the latency deviation is small at different frequencies under real-time constraints. It should be noted that the latency also depends on the size of the transmitted data, which was fixed at 1K byte in this experiment. The maximum latency was less than 150 μs.

### 4.8 Real-time Performance of EtherCAT Master

The EtherCAT master needs to run on a real-time system to ensure strict real-time performance. The robot system

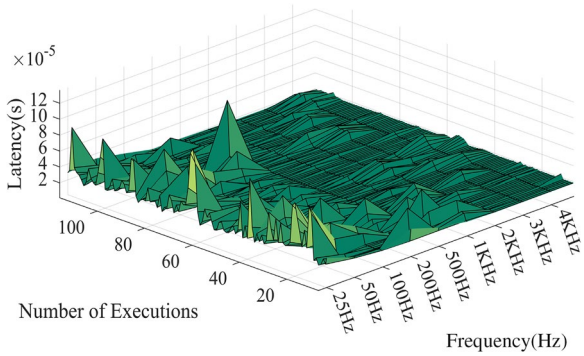


**Figure 31** Latency distribution of ROS2 at different frequencies

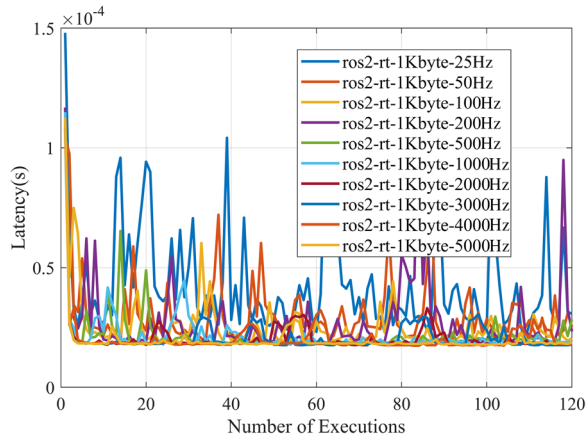


**Figure 32** Real-time performance of Preempt_RT-Linux-ROS2 at different frequencies

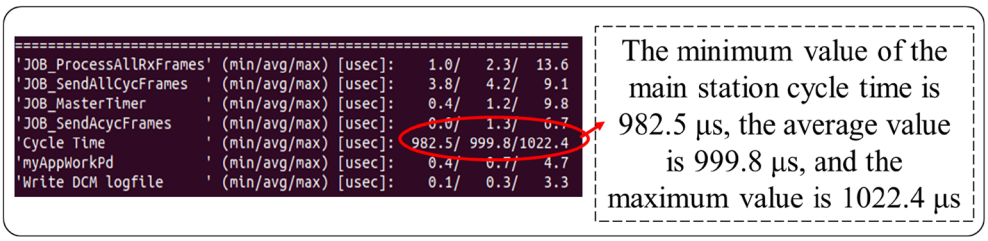


**Figure 33** Real-time performance of EtherCAT master

platform (see Figure 2) has one EtherCAT master and 16 EtherCAT slaves. In the experiment, the master station cycle period was set to 1000 μs, and we obtained the EtherCAT master's latency data for a duration of 1613610 ms, as shown in Figure 33. The minimum period of the master station was 982.5 μs, the average period was 999.8 μs, and the maximum period was 1022.4 μs. It can be seen that the EtherCAT master exhibits good real-time performance and can be applied to robot control.

## 5 Conclusions

This paper proposes an optimization and assessment of ROS2's real-time performance, utilizing a method that melds fair and first-in-first-out scheduling strategies for a robotic control system. This method, predicated on the ROS2's DDS transmission mechanism, adopts the use of Preempt_RT to construct a fully preemptive, event-driven system kernel, thereby improving the timeliness and reliability of ROS2's data transmission.

We engage in both qualitative and quantitative evaluations of the real-time performance of ROS1 and ROS2, considering factors such as throughput, transmission methodology, QoS service quality, frequency, quantity of subscription nodes, and EtherCAT master. Our findings indicate reliable real-time performance of the optimized ROS2 with Preempt_RT implementation. This research intuitively demonstrates that, in large-scale data transmission and multiple node subscriptions, ROS2 outperforms ROS1 in terms of real-time performance.

Specific conclusions of the paper are as follows.

(1) The key to improving the real-time performance of ROS2 lies in optimizing the real-time performance of the operating system. The use of the Preempt_RT patch can reduce the latency in ROS2 message transmission. Preempt_RT improves the real-time computing capability of the native Linux kernel through high-precision timers, thread interrupt handlers, sleep spinlocks, real-time mutexes, and RCU synchronization mechanisms.

(2) The real-time performance of the optimized ROS2 system was systematically and comprehensively evaluated under stringent operating conditions with the CPU running at full load. The system demonstrated stable real-time performance, running for 25.7 h with a maximum latency of 82 μs and an average latency of 2 μs.

(3) This study compares the real-time performance of ROS1 and ROS2, both located in the application layer above the Linux kernel. The real-time performance of the optimized Preempt_RT-Linux-ROS2 is much better than that of the Native-Linux-ROS2. Additionally, for a single publisher node corresponding to multiple subscriber nodes, ROS2 demonstrates fairer real-time performance than ROS1 for multiple subscribers, making ROS2 more suitable for the development of real-time control systems.

(4) The optimized Preempt_RT-Linux maintains stable performance for both average jitter and maximum jitter at different frequencies, with a timing jitter cycle of less than 60 μs. The study also measures the real-time performance of the EtherCAT master, with a timing cycle of 1000 us and a worst-case timing cycle of 1022.4 us, demonstrating the effectiveness of the optimized system.

**Acknowledgements**
Not applicable.

**Authors' Contributions**
YY designed and wrote the paper, XL and ZN completed manuscript revisions, FX provided suggestions and guidance, ZL assisted with programming, and PL provided assistance in device construction. All authors read and approved the final manuscript.

**Authors' Information**
Yanlei Ye born in 1991, is currently a Ph.D. candidate at *Department of Mechanical Engineering (DME), Tsinghua University, China*. His research interests include robot operating systems and compliant motion control.
Zhenguo Nie born in 1983, is currently an associate professor at *DME, Tsinghua University, China*. His research interests include intelligent design and surgical robotics.
Xinjun Liu born in 1971, is currently a professor and a Ph.D. candidate supervisor at *DME, Tsinghua University, China*. His research interests include robotics, parallel mechanisms, and advanced manufacturing equipment.
Fugui Xie born in 1982, is currently an associate professor and a Ph.D. candidate supervisor at *DME, Tsinghua University, China*. His research interests include parallel mechanisms and mobile machining robots.
Zihao Li born in 1992, is currently a Ph.D. candidate at *DME, Tsinghua University, China*. His research interests include cooperative robot and teleoperation.

Peng Li born in 1989, is currently a Ph.D. candidate at *DME, Tsinghua University, China*. His research interests include collaborative robot design and control.

**Funding**
Supported by National Key Research and Development Program of China (Grant No. 2019YFB1309900), and Institute for Guo Qiang, Tsinghua University of China (Grant No. 2019GQG0007).

## Declarations

**Competing Interests**
The authors declare no competing financial interests.

Received: 22 February 2023 Revised: 11 November 2023 Accepted: 12 November 2023
Published online: 04 December 2023

**References**

[1] F Reghenzani, G Massari, W Fornaciari. The real-time linux kernel: A survey on preempt_rt. *ACM Computing Surveys (CSUR)*, 2019, 52(1): 1-36.
[2] S Macenski, T Foote, B Gerkey, et al. Robot Operating System 2: Design, architecture, and uses in the wild. *Science Robotics*, 2022, 7(66): eabm6074.
[3] S Barut, M Boneberger, P Mohammadi, et al. Benchmarking real-time capabilities of ROS2 and OROCOS for robotics applications. *IEEE International Conference on Robotics and Automation*, Xi'an, China, May 30 -June 05, 2021: 708-714.
[4] R Mittal, A Konno, S Komizunai. Implementation of hoap-2 humanoid walking motion in openhrp simulation. *International Conference on Computing Communication Control and Automation*, Pune, India, February 26-27, 2015: 29-34.
[5] G Metta, P Fitzpatrick, L Natale. YARP: yet another robot platform. *International Journal of Advanced Robotic Systems*, 2006, 3(1): 43-48.
[6] T Fietzek, H Ü Dinkelbach, F H Hamker. ANNarchy-iCub: An interface for easy interaction between neural network models and the iCub Robot. *Computational Intelligence and Virtual Environments for Measurement Systems and Applications*, Chemnitz, Germany, June 15-17, 2022.
[7] J Jackson. Microsoft robotics studio: A technical introduction. *IEEE Robotics & Automation Magazine*, 2007, 14(4): 82-87.
[8] P Marion, M Fallon, R Deits, et al. Director: A user interface designed for robot operation with shared autonomy. *Journal of Field Robotics*, 2017, 34(2): 262-280.
[9] D Kortenkamp, R Simmons, D Brugali. Robotic systems architectures and programming. *Springer Handbook of Robotics*, 2016: 283-306.
[10] M Quigley, K Conley, B Gerkey, et al. ROS: an open-source robot operating system. *ICRA Workshop on Open Source Software*, Kobe, Japan, 2009.
[11] T Itsuka, M Song, A Kawamura, et al. Development of ROS2-TMS: new software platform for informationally structured environment. *ROBOMECH Journal*, 2022, 9(1): 1-19.
[12] Y Maruyama, S Kato, T Azumi. Exploring the performance of ROS2. *Proceedings of the 13th International Conference on Embedded Software*, Pittsburgh, PA, USA, October 02-07, 2016.
[13] M Albonico, M Đorđević, E Hamer, et al. Software engineering research on the Robot Operating System: A systematic mapping study. *Journal of Systems and Software*, 2022.
[14] M Karamousadakis. *Real-time programming of EtherCAT master in ROS for a quadruped robot*. National Technical University of Athens, 2019.
[15] H Wei, Z Shao, Z Huang, et al. RT-ROS: A real-time ROS architecture on multi-core processors. *Future Generation Computer Systems*, 2016, 56: 171-178.
[16] K Belsare. Micro-ROS*//A Koubaa. Robot Operating System (ROS)*. Cham: Springer International Publishing, 2023: 3-55.
[17] A Hakiri, P Berthou, A Gokhale, et al. Publish/subscribe-enabled software defined networking for efficient and scalable IoT communications. *IEEE Communications Magazine*, 2015, 53(9): 48-54.
[18] W Sim, B Song, J Shin, et al. Data distribution service converter based on the open platform communications unified architecture publish–subscribe protocol. *Electronics*, 2021, 10(20): 2524.
[19] H Choi, Y Xiang, H Kim. PiCAS: New design of priority-driven chain-aware scheduling for ROS2. *Real-Time and Embedded Technology and Applications Symposium*, Nashville, TN, USA, May 18-21, 2021: 251-263.
[20] T Kronauer, J Pohlmann, M Matthé, et al. Latency analysis of ROS2 multi-node systems. *Multisensor Fusion and Integration for Intelligent Systems*, Karlsruhe, Germany, September 23-25, 2021.
[21] L Ding, M C Qu, Y L Zhang, et al. *Analysis and engineering application of ROS2*. Beijing: Tsinghua University Press, 2019. (in Chinese)
[22] H Choi. *On the design and analysis of autonomous real-time systems*. University of California, Riverside, 2021.
[23] B Akesson, M Nasri, G Nelissen, et al. An empirical survey-based study into industry practice in real-time systems. *Real-Time Systems Symposium*, Houston, TX, USA, December 01-04, 2020: 3-11.
[24] H Kopetz, W Steiner. *Real-time systems: design principles for distributed embedded applications*. Springer Nature, 2022.
[25] A Barbalace, A Luchetta, G Manduchi, et al. Performance comparison of VxWorks, Linux, RTAI, and Xenomai in a hard real-time application. *IEEE Transactions On Nuclear Science,* 2008, 55(1): 435-439.
[26] D Dasari, M Becker, D Casini, et al. End-to-end analysis of event chains under the qnx adaptive partitioning scheduler. *Real-Time and Embedded Technology and Applications Symposium*, Milano, Italy, May 04-06, 2022: 214-227.
[27] C Maiza, H Rihani, J M Rivas, et al. A survey of timing verification techniques for multi-core real-time systems. *ACM Computing Surveys (CSUR)*, 2019, 52(3): 1-38.
[28] D Ramegowda, M Lin. Energy efficient mixed task handling on real-time embedded systems using FreeRTOS. *Journal of Systems Architecture*, 2022: 131.
[29] R Delgado, B You, B W Choi. Real-time control architecture based on Xenomai using ROS packages for a service robot. *Journal of Systems and Software*, 2019, 151: 8-19.
[30] R Delgado, J Park, B W Choi. Open embedded real-time controllers for industrial distributed control systems. *Electronics*, 2019, 8(2): 223.
[31] J Arm, Z Bradac, V Kaczmarczyk. Real-time capabilities of Linux RTAI. *Ifac-Papersonline*, 2016, 49(25): 401-406.
[32] G K Adam, N Petrellis, L T Doulos. Performance assessment of Linux Kernels with PREEMPT_RT on ARM-Based embedded devices. *Electronics*, 2021, 10(11): 1331.