目 录

第 1 章 Overview of uROS

1.1 Structure of uROS

1.2 Basic Concepts

The major concepts (publishers, subscriptions, services, timers, ...) are identical with ROS2. They even rely on the same implementation, as the micro-ROS C API is based on the ROS 2 client support library (rcl), enriched with a set of convenience functions by the package rclc[1]. That is, rclc does not add a new layer of types on top of rcl (like rclcpp and rclpy do) but only provides functions that ease the programming with the rcl types. New types are introduced only for concepts that are missing in rcl, such as the concept of an executor.

1.2.1 Nodes

Each node in ROS should be responsible for a single, module purpose (e.g. one node for controlling wheel motors, one node for controlling a laser range-finder, etc). Each node can send and receive data to other nodes via topics, services, actions, or parameters.

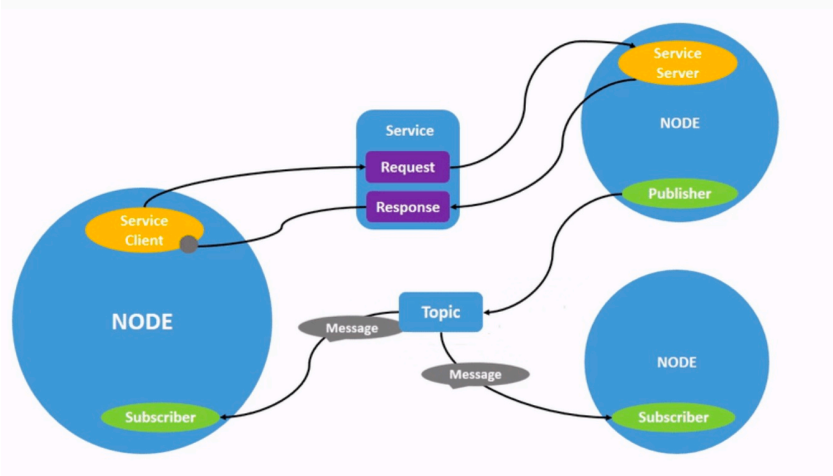

图 1.1 Nodes

1.2.1.1 Initialization

1. Create a node with default configuration:

```
// Initialize micro-ROS allocator
rcl_allocator_t allocator = rcl_get_default_allocator();
```

[1](https://github.com/ros2/rclc/)

```
3
    // Initialize support object
5   rclc_support_t support;
    rcl_ret_t rc = rclc_support_init(&support, argc, argv, &allocator);
7
    // Create node object
9   rcl_node_t node;
    const char * node_name = "test_node";
11
    // Node namespace (Can remain empty "")
13  const char * namespace = "test_namespace";

15  // Init default node
    rc = rclc_node_init_default(&node, node_name, namespace, &support);
17  if (rc != RCL_RET_OK) {
        ... // Handle error
19      return -1;
    }
```

Listing 1 Python example

2. Create a node with custom options: The configuration of the node will also be applied to its future elements (Publishers, subscribers, services, …). The API used to customize the node options differs between ROS2 distributions:

Foxy: The rcl_node_options_t is used to configure the node

```
// Initialize allocator and support objects
2 ...

4 // Create node object
  rcl_node_t node;
6 const char * node_name = "test_node";

8 // Node namespace (Can remain empty "")
  const char * namespace = "test_namespace";
10
  // Get default node options and modify them
12 rcl_node_options_t node_ops = rcl_node_get_default_options();

14 // Set node ROS domain ID to 10
  node_ops.domain_id = (size_t)(10);
16
  // Init node with custom options
18 rc = rclc_node_init_with_options(&node, node_name, namespace, &support, &
      node_ops);
```

```
if (rc != RCL_RET_OK) {
... // Handle error
return -1;
}
```

Listing 2 Python example

Galactic: In this case, the node options are configured on the rclc_support_t object with a custom API

```
// Initialize micro-ROS allocator
rcl_allocator_t allocator = rcl_get_default_allocator();

// Initialize and modify options (Set DOMAIN ID to 10)
rcl_init_options_t init_options = rcl_get_zero_initialized_init_options();
rcl_init_options_init(&init_options, allocator);
rcl_init_options_set_domain_id(&init_options, 10);

// Initialize rclc support object with custom options
rclc_support_t support;
rclc_support_init_with_options(&support, 0, NULL, &init_options, &allocator)
    ;

// Create node object
rcl_node_t node;
const char * node_name = "test_node";

// Node namespace (Can remain empty "")
const char * namespace = "test_namespace";

// Init node with configured support object
rclc_node_init_default(&node, node_name, namespace, &support);

if (rc != RCL_RET_OK) {
... // Handle error
return -1;
}
```

Listing 3 Python example

3. Cleaning up: To destroy a initialized node all entities owned by the node (Publishers, subscribers, services, …) have to be destroyed before the node itself. This will delete the node from ROS2 graph, including any generated infrastructure on the agent (if possible) and used memory on the client.

```
// Destroy created entities (Example)
```

```
rcl_publisher_fini(&publisher, &node);
...

// Destroy the node
rcl_node_fini(&node);
```

Listing 4 Python example

1.2.2 Publishers and subscribers

ROS 2 publishers and subscribers are the basic communication mechanism between nodes using topics. Ready to use code related to this concepts can be found in micro-ROS-demos/rclc/int32_publisher and micro-ROS-demos/rclc/int32_subscriber folders. Fragments of code from this examples are used on this tutorial. ROS 2 breaks complex systems down into many modular nodes. Topics are a vital element of the ROS graph that act as a bus for nodes to exchange messages.

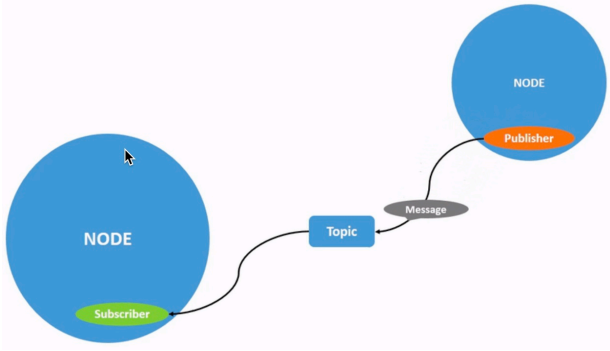

图 1.2 Publishers and subscribers

A node may publish data to any number of topics and simultaneously have subscriptions to any number of topics. Topics are one of the main ways in which data is moved between nodes and therefore between different parts of the system.

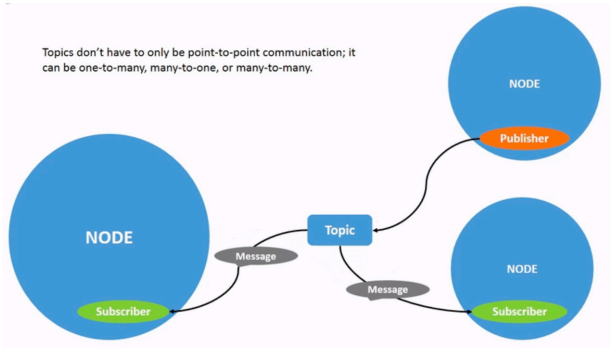

图 1.3 Publishers and subscribers

1.2.2.1 Publisher

1. Initialization: Starting from a code where RCL is initialized and a micro-ROS node is created, there are tree ways to initialize a publisher depending on the desired quality-of-service configuration. For a detail on the available QoS options and the advantages and disadvantages between reliable and best effort modes, check the QoS tutorial.

```
// Publisher object
rcl_publisher_t publisher;
const char * topic_name = "test_topic";

// Get message type support
const rosidl_message_type_support_t * type_support =
    ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, Int32);

// Creates a reliable rcl publisher
rcl_ret_t rc = rclc_publisher_init_default(
    &publisher, &node,
    &type_support, &topic_name);

if (RCL_RET_OK != rc) {
    ...  // Handle error
    return -1;
}
```

Listing 5 Reliable (default)

```
// Publisher object
rcl_publisher_t publisher;
const char * topic_name = "test_topic";

// Get message type support
const rosidl_message_type_support_t * type_support =
    ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, Int32);

// Creates a best effort rcl publisher
rcl_ret_t rc = rclc_publisher_init_best_effort(
    &publisher, &node,
    &type_support, &topic_name);

if (RCL_RET_OK != rc) {
    ...  // Handle error
    return -1;
}
```

Listing 6 Best effort

```
// Publisher object
rcl_publisher_t publisher;
const char * topic_name = "test_topic";

// Get message type support
const rosidl_message_type_support_t * type_support =
    ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, Int32);

// Set publisher QoS
const rmw_qos_profile_t * qos_profile = &rmw_qos_profile_default;

// Creates a rcl publisher with customized quality-of-service options
rcl_ret_t rc = rclc_publisher_init(
    &publisher, &node,
    &type_support, &topic_name, qos_profile);

if (RCL_RET_OK != rc) {
    ...  // Handle error
    return -1;
}
```

Listing 7 Custom QoS

2. Publish a message: Use rcl_publish to publish messages to the topic. For periodic publications, rcl_publish[2] can be placed inside a timer callback. Check the Executor and timers section for details.

```
// Int32 message object
std_msgs__msg__Int32 msg;

// Set message value
msg.data = 0;

// Publish message
rcl_ret_t rc = rcl_publish(&publisher, &msg, NULL);

if (rc != RCL_RET_OK) {
    ...  // Handle error
    return -1;
}
```

Listing 8 publish messages

[2]Note that rcl_publish is thread safe and can be called from multiple threads.

1.2.2.2 Subscriber

1. Initialization: The subscription initialization is almost identical to the publisher one:

```
// Subscription object
rcl_subscription_t subscriber;
const char * topic_name = "test_topic";

// Get message type support
const rosidl_message_type_support_t * type_support =
    ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, Int32);

// Initialize a reliable subscriber
rcl_ret_t rc = rclc_subscription_init_default(
    &subscriber, &node,
    &type_support, &topic_name);

if (RCL_RET_OK != rc) {
    ...  // Handle error
    return -1;
}
```

Listing 9 Reliable (default)

```
// Subscription object
rcl_subscription_t subscriber;
const char * topic_name = "test_topic";

// Get message type support
const rosidl_message_type_support_t * type_support =
    ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, Int32);

// Initialize best effort subscriber
rcl_ret_t rc = rclc_subscription_init_best_effort(
    &subscriber, &node,
    &type_support, &topic_name);

if (RCL_RET_OK != rc) {
    ...  // Handle error
    return -1;
}
```

Listing 10 Best effort

```
// Subscription object
rcl_subscription_t subscriber;
```

```
    const char * topic_name = "test_topic";

    // Get message type support
    const rosidl_message_type_support_t * type_support =
        ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, Int32);

    // Set client QoS
    const rmw_qos_profile_t * qos_profile = &rmw_qos_profile_default;

    // Initialize a subscriber with customized quality-of-service options
    rcl_ret_t rc = rclc_subscription_init(
        &subscriber, &node,
        &type_support, &topic_name, qos_profile);

    if (RCL_RET_OK != rc) {
        ...  // Handle error
        return -1;
    }
```

Listing 11 Custom QoS

2. Callbacks: The executor is responsible to call the configured callback when a message is published. The function will have the message as its only argument, containing the values sent by the publisher:

```
    // Function prototype:
    void (* rclc_subscription_callback_t)(const void *);

    // Implementation example:
    void subscription_callback(const void * msgin)
    {
        // Cast received message to used type
        const std_msgs__msg__Int32 * msg = (const std_msgs__msg__Int32 *)
    msgin;

        // Process message
        printf("Received: %d\n", msg->data);
    }
```

Listing 12 Sub-callback decalaration

Once the subscriber and the executor are initialized, the subscriber callback must be added to the executor to receive incoming publications once its spinning:

```
    // Message object to receive publisher data
    std_msgs__msg__Int32 msg;
```

```
    // Add subscription to the executor
    rcl_ret_t rc = rclc_executor_add_subscription(
        &executor, &subscriber, &msg,
        &subscription_callback, ON_NEW_DATA);

    if (RCL_RET_OK != rc) {
        ... // Handle error
        return -1;
    }

    // Spin executor to receive messages
    rclc_executor_spin(&executor);
```

Listing 13 Sub-callback registration

1.2.2.3 Message initialization

Before publishing or receiving a message, it may be necessary to initialize its memory for types with strings or sequences. Check the Handling messages memory in micro-ROS section for details.

1.2.2.4 Cleaning Up

After finishing the publisher/subscriber, the node will no longer be advertising that it is publishing/listening on the topic. To destroy an initialized publisher or subscriber:

```
// Destroy publisher
rcl_publisher_fini(&publisher, &node);

// Destroy subscriber
rcl_subscription_fini(&subscriber, &node);
```

Listing 14 To destroy an initialized publisher or subscriber

1.2.3 Services

Services are another method of communication for nodes in the ROS graph. Services are based on a call-and-response model, versus topics'publisher-subscriber model. While topics allow nodes to subscribe to data streams and get continual updates, services only provide data when they are specifically called by a client. Ready to use code related to this concepts can be found in micro-ROS-demos/rclc/addtwoints_server and micro-ROS-demos/rclc/addtwoints_client folders. Fragments of code from this examples are used on this tutorial.

TODO: https://micro.ros.org/docs/tutorials/programming_rcl_rclc/service/

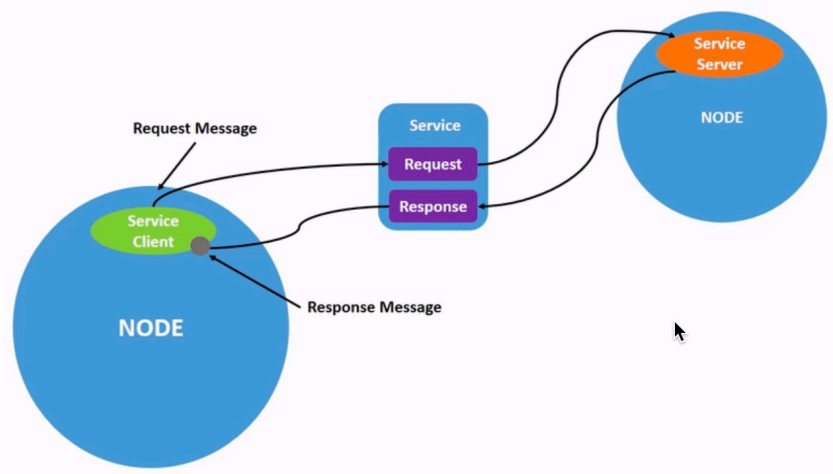

图 1.4 Service

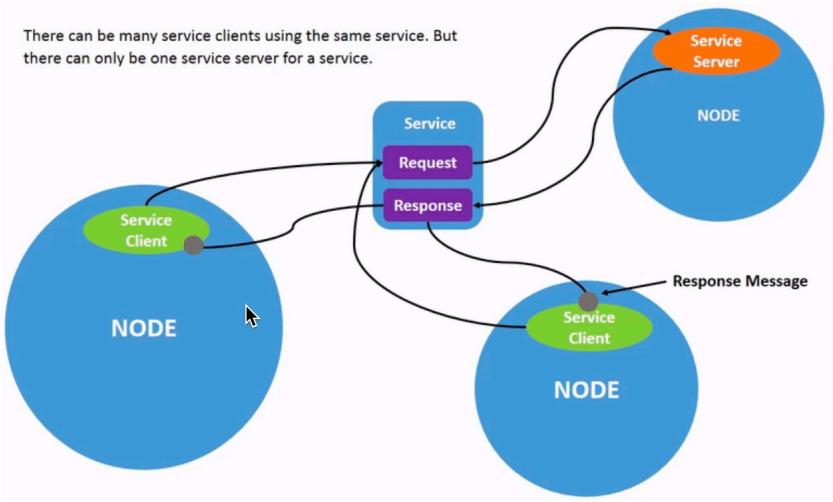

图 1.5 Service2

1.2.3.1 Service server

1.2.3.2 Service Client

1.2.3.3 Message initialization

1.2.3.4 Cleaning Up

1.2.4 Parameter server

TODO: https://micro.ros.org/docs/tutorials/programming_rcl_rclc/parameters/

1.2.5 Executor and timers

1.2.5.1 Timer

Timers can be created and added to the executor, which will call the timer callback periodically once it is spinning. They are usually used to handle periodic publications or events.

1. Initialization:

```
    // Timer period on nanoseconds
    const unsigned int timer_period = RCL_MS_TO_NS(1000);
    // Create and initialize timer object
    rcl_timer_t timer;
    rcl_ret_t rc = rclc_timer_init_default(&timer, &support, timer_period,
    timer_callback);
    // Add to the executor
    rc = rclc_executor_add_timer(&executor, &timer);
    if (rc != RCL_RET_OK) {
        ... // Handle error
        return -1;
    }
```

Listing 15 Timer Initialization

2. Callback: The callback gives a pointer to the associated timer and the time elapsed since the previous call or since the timer was created if it is the first call to the callback. During the callback the timer can be canceled or have its period and/or callback modified using the passed pointer. Check rcl/timer.h for details.

```
    void timer_callback(rcl_timer_t * timer, int64_t last_call_time)
    {
        printf("Last callback time: %ld\n", last_call_time);

        if (timer != NULL) {
            // Perform actions
            ...
        }
    }
```

Listing 16 Timer Callback

3. Cleaning Up: To destroy an initialized timer, This will deallocate used memory and make the timer invalid.

```
    // Destroy timer
    rcl_timer_fini(&timer);
    This will deallocate used memory and make the timer invalid
```

Listing 17 Timer Cleaning up

1.2.5.2 Executor

1.2.6 Quality of service

1.2.7 micro-ROS utilities

1.3 Data Distribution Service (DDS)

DDS itself is an API specification and the actual underlying protocol is the Real-Time Publish-Subscribe Protocol (RTPS)[1]. Multiple open-source implementations of DDS are available for full-fledged computers running Linux or Windows. Few implementations are suitable for resource-constrained embedded systems.

eProsima FastRTPS is a popular RTPS implementation that is also the underlying middleware for ROS 2. OpenDDS is another widespread implementation that is a DDS implementation on top of the RTPS protocol[2]. Real-Time Innovations (RTI) offers ConnextDDS, a commercial DDS implementation[3]. All of the implementations mentioned above target Linux, Windows or MacOS and do not run on embedded platforms. RTI also offers microDDS, an implementation suitable for microcontrollers. In contrast to embeddedRTPS[4], microDDS is not open source and only commercially available. Lastly, freeRTPS 2 is an implementation originally developed for microcontrollers in the context of ROS 2. This implementation explicitly targets STM32 microprocessors and is not under active maintenance.

The DDS Extremely Resource Constrained Environments (XRCE) standard is intended for integration of embedded systems into DDS networks[5]. The resource-constrained platform is integrated into the DDS network by a more powerful server, that acts as a gateway. **In this architecture, the microcontroller is not an independent, first-class participant in the communication network, as it's ability to communicate depends on server.** This also introduces a single point of failure, rendering this architecture potentially unsuitable for safety-critical applications.

1.3.1 Background of RTPS

This section provides a brief overview of the basic concepts behind RTPS. We cover the basic entities, the discovery procedure and data exchange mechanisms. An exhaustive description is provided in the standard maintained by the OMG[1].

1.3.1.1 Entities

As depicted in Fig. 1.6, the basic actors in RTPS are *participants*, which in turn have a variable number of readers or writers. Readers or writers (summarized as *end-*

points in the following) can be identified with an id composed of a unique participant prefix and an entity id which is unique within this participant. They communicate through a publish-subscribe mechanism, a concept also common in other messaging frameworks such as ROS or Message Queuing Telemetry Transport (MQTT). **Endpoints are loosely coupled only by a topic name and data type definition.** From a user perspective, messages published by writers are addressed to a topic and not to specific recipients. Respectively, readers subscribe to a specific topic and join the network without knowledge about the specific writer(s). For this, participants become aware of each other during runtime in the course of a decentralized discovery process. Thus, readers and writers form a loosely coupled communication network that is suitable for runtime integrated systems.

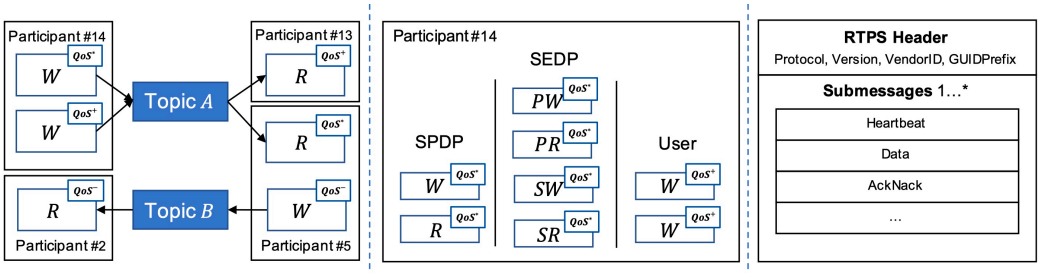

图 1.6 Left: Participants Exchange Messages via User-defined Reader and Writer Endpoints. **Middle**: Built-in RTPS endpoints for Participant (SPDP) and Endpoint Discovery (SEDP) alongside user-defined Endpoints. **Right**: Components of RTPS Messages encapsulated in UDP Packets.

Various Quality of Service (QoS) parameters can be used to customize the behavior of endpoints. For example, writers and readers communicate either in a *reliable* or *best-effort* mode. **Reliable writers keep track of already transmitted packages through sequence numbers and are able to resend messages in case of a transmission failure. Best-effort writers do not provide this capability.** The user interacts with endpoints by passing serialized data to writers or receiving data from readers. RTPS is specified for TCP/UDP communication.

In general, endpoints send UDP packets that are structured as depicted in Fig. 1.6. Each message starts with a header that contains, among other fields, the RTPS version or vendor identification. The header is followed by a variable amount of sub-messages that are used for data exchange and discovery. `Data` sub-messages are used to exchange generic, serialized data between readers and writers. `Heartbeat` and `AckNack` sub-messages are used to provide a reliable communication. The `Heartbeat` sub-message allows writers to announce the available sequence numbers in its storage. Reliable readers send `AckNack` sub-messages to communicate missing as well as received packets.

Based on those acknowledgments, the reliable writer can decide when a packet can be safely removed from its history. While one or multiple participants are addressed by a specific port, the target endpoint is identified by the entity id passed with the aforementioned messages.

1.3.1.2 Discovery Protocols

Entities in RTPS discover each other through a decentralized two-step discovery process, using a set of six built-in endpoints as depicted in Fig. 1.6. Technically, these are regular endpoints that function just like user-defined endpoints but communicate specific messages for the discovery process and have a fixed entity id. In the first step, the Simple Participant Discovery Protocol (SPDP) uses a reader/writer pair of best-effort endpoints for discovery of participants and their properties in the network. Each participant periodically sends messages through the SPDP-writer containing information about the participant on standardized IP broadcast addresses. The transmitted UDP packets are made up of RTPS packets containing a `Data` sub-message and are received by remote SPDP-readers. Discovery of those endpoints is not required because the messages are send on addresses which are relevant for all participants and the destination is known as the built-in endpoints have fixed entity ids. SPDP messages contain, among others, the unicast or multicast locators (i.e. IP-address and port) for meta (e.g. discovery) and user traffic to the sending participant.

Once the participants become aware of each other, they exchange information about user-defined endpoints using the Simple Endpoint Discovery Protocol (SEDP). For this, reliable writers and readers are used. The publication writer PW sends information about user-defined writers to all known participants by addressing the publication readers PR of all known participants. For each user-defined endpoint, information about it's QoS parameters, locators and their entity id are transmitted. Analogously, user-defined readers are announced through the subscription-writer SW and participants receive these respective messages through the subscription-reader SR. After discovering a new endpoint, a participant tries to match it with one if its own endpoints. Endpoints are matched if the topic name, data type name and QoS parameters are compatible.

Once local endpoints have been matched with remote endpoints, every message that is handed to the writer will be send to all known, matching readers in the network. For this purpose, the user hands over a serialized byte array, that is inserted into RTPS packet containing a `Data` sub-message. Note that SPDP and SEDP messages are sent periodically, therefore participants and endpoints can dynamically join the RTPS net-

work.

第 2 章 Design of uROS

2.1 RCLC

2.1.1 RCLC-Executor

Here we introduce the rclc Executor, which is a ROS 2 Executor implemented based on and for the rcl API, for applications written in the C language. Often embedded applications require real-time to guarantee end-to-end latencies and need deterministic runtime behavior to correctly replay test data. However, this is difficult with the default ROS 2 Executor because of its complex semantics, as discussed in the previous section.

First, we will analyse the requirements for real-time embedded applications and, secondly, derive simple features for an Executor to enable deterministic and real-time behavior. Then we will present the API of the RCLC-Executor and provide example usages of the RCLC-Executor to address these requirements.

2.1.1.1 Requirement Analysis

TODO: Real-time embedded application use-case and LET[1]

The described embedded use case relies on the following concepts:

(1) periodic execution of processes
(2) assignment of fixed priorities to processes
(3) preemptive scheduling of processes
(4) co-operative scheduling of tasks within a process (sequential execution)
(5) data synchronization with LET-semantics

While periodic activation is possible in ROS 2 by using timers, preemptive scheduling is supported by the operating system and assigning priorities on the granularity of threads/processes that correspond to the ROS nodes; it is not possible to sequentially execute callbacks, which have no data-dependency. Furthermore data is read from the DDS queue just before the callback is executed and data is written sometime during the time the application is executed. While the spin_period function of the rclcpp-Executor allows to check for data at a fixed period and executing those callbacks for which data is available, however, with this spin-function does not execute all callbacks irrespective whether data is available or not. So spin_period is not helpful to periodically execute a number of callbacks (aka tasks within a process). So we need a mechanism that triggers the execution

[1] https://github.com/ros2/rclc/tree/master/rclc

of multiple callbacks (aka tasks) based on a timer. Data transmission is achieved via DDS which does not allow to implement a LET-semantics. To summarize, we derive the following requirements:

(1) trigger the execution of multiple callbacks
(2) sequential processing of callbacks
(3) data synchronization with LET semantics

2.1.1.2 RCLC-Executor Features

Based on the real-time embedded use cases as well as the software architecture patterns in mobile robotics, RCLC-Executor are proposed with the following main features:

User-defined sequential execution of callbacks: At configuration, the user defines the order of handles and whether the handle shall be called only when new data is available (ON_NEW_DATA) or whether the callback shall always be called (ALWAYS). At runtime, all handles are processed in the user-defined order: if the configuration of handle is ON_NEW_DATA, then the corresponding callback is only called if new data is available; if the configuration of the handle is ALWAYS, then the corresponding callback is always executed. In case, no data is available from DDS, then the callback is called with no data (e.g. NULL pointer).

Trigger condition to activate processing: Given a set of handles, a trigger condition based on the input data of these handles shall decide when the processing is started. Available options include: ALL operation, fires when input data is available for all handles; ANY operation, fires when input data is available for at least one handle; ONE, fires when input data for a user-specified handle is available; User-defined function, user can implement more sophisticated logic.

LET-Semantics:

- Assumption: time-triggered system, the executor is activated periodically
- When the trigger fires, reads all input data and makes a local copy
- Processes all callbacks in sequential order
- Write output data at the end of the executor's period (Note: this is not implemented yet)

Additionally the rclcpp Executor semantics (RCLCPP) is implemented and chosen as the default configuration:

- waiting for new data for all handles (rcl_wait)
- using trigger condition ANY

- if trigger fires, start processing handles in pre-defined sequential order
- request from DDS-queue the new data just before the handle is executed (rcl_take)

2.2 RCL

2.3 RMW

2.4 XRCE-DDS

eProsima Micro XRCE-DDS is an open-source wire protocol that implements the OMG DDS for e**X**tremely **R**esource **C**onstrained **E**nvironment standard (DDS-XRCE). The aim of the DDS-XRCE protocol is to provide access to the DDS Global-Data-Space from resource-constrained devices. This is achieved thanks to a client-server architecture, where low resource devices (called XRCE Clients), are connected to a server (called XRCE Agent), which acts on behalf of its clients in the DDS Global-Data-Space.

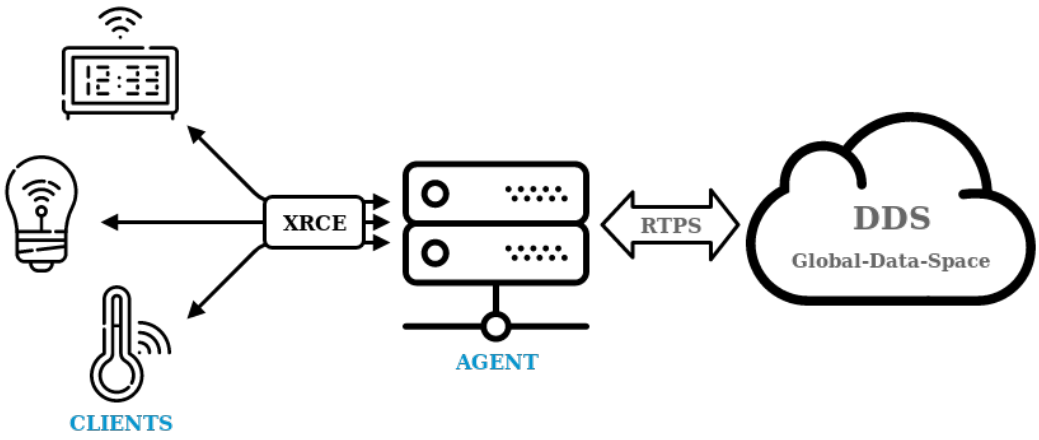

图 2.1 structure

Micro XRCE-DDS is composed by two main elements:

- **Micro XRCE-DDS Agent**: a C++11 out-of-the-box application which implements the XRCE Agent functionality
- **Micro XRCE-DDS Client**: a C99 library which implements the XRCE Client side functionality

In addition, Micro XRCE-DDS uses other two components:

- **Micro CDR**: a de/serialization engine used in the Client library
- **Micro XRCE-DDS Gen**: a code generator tool used for generating Micro CDR de/serialization functions and Client apps examples from IDL sources.

2.4.1 Main Features

1. Low Resource Consumption: As it was mentioned above, Micro XRCE-DDS is focused on microcontroller applications. Therefore, the design and implementation of this middleware have been carried out taking into account the memory constraints of this kind of devices. A proof of this is the fact that the XRCE Client is completely dynamic memory free. From the point of view of the memory footprint, the latest version of this library has a memory consumption of less than 75 KB of Flash memory and around 3 KB of RAM for a complete publisher and subscriber application handling messages sizes on the order of 512 B. For more detailed information on the memory consumption as a function of message size, entity number and internal memory management of the middleware library, please refer to the Micro XRCE-DDS memory profiling section. Moreover, this library is highly configurable thanks to a profile concept that enables to choose, add or remove some features in configuration time. That allows customizing the XRCE Client library size, if there are features that are not used. There are several definitions for configuring and building the Client library at compile-time. These definitions allow to create a version of the library according to the application requirements, and can be modified in the client.config file. For incorporating the desired configuration, it is necessary to run the cmake command every time the definitions change.

2. Multi-Transport Support: As part of the profiles discussed in the previous section, the user can choose between several transport layers to communicate the Clients with the Agent. Indeed, in contrast to other IoT middleware such as MQTT and CoaP, which work over only a particular transport layer, XRCE supports multiple transport protocols natively. In particular, the latest version of Micro XRCE-DDS support: UDP, TCP and a custom Serial transport protocol. Apart from this, Micro XRCE-DDS has a transport interface for both Agent and Client which allows to implement custom transports in a straightforward manner. This makes the port of Micro XRCE-DDS to different platforms and the addition of new transports a seamless task that any user can undertake.

3. Multi-Platform Support: The XRCE Client supports FreeRTOS, Zephyr and NuttX as embedded RTOS. Moreover, it also runs on Windows and Linux. On the other hand, the XRCE Agent supports Windows and Linux.

4. QoS support: The XRCE Client library allows the user to use two different approaches for creating DDS entities in the XRCE Agent: By XML (the default option), By reference. When using the default option, users are enabled to create entities either

in Reliable or Best-Effort mode, with the XML files written and stored on the Client side. But these QoS configurations may not fit some users'requirements. For these cases, Micro XRCE-DDS allows to create entities directly on the Agent, where the user can write custom XML QoS as in DDS. Each entity available on the Agent will be associated to a label, so that the Clients can to create the entities they need for the communication by just referring to these labels. Additionally, using references will also reduce the memory consumption of the Client inside the MCU. This is because the reference approach allows avoiding to build the parts of the code where XMLs are stored.

2.4.2 Micro XRCE-DDS Client

The Micro XRCE-DDS Clients[2] are lightweight entities meant to be compiled on eXtremely Resource Constrained Environments.

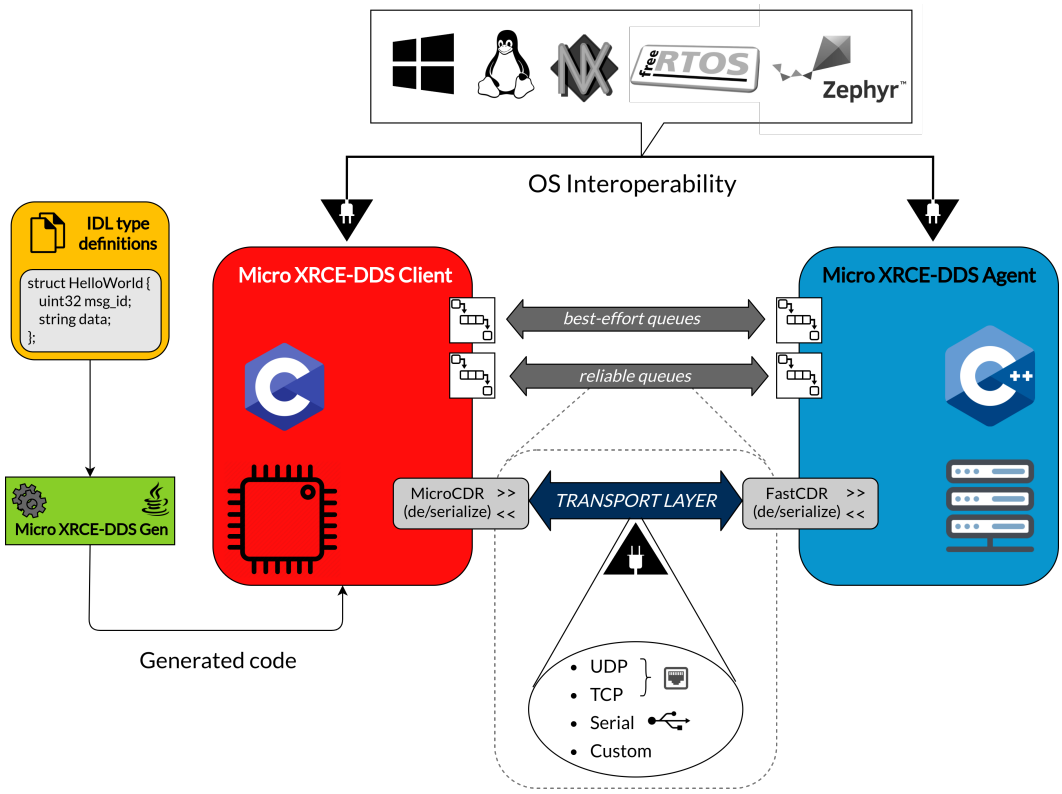

图 2.2 structure

The Micro XRCE-DDS Clients request operations to the Agent to publish and/or subscribe to topics in the DDS global dataspace. Remote procedure calls, as defined by the DDS-RPC standard, are also supported, allowing Clients to communicate in the DDS dataspace according to a request/reply paradigm. The Agents process these requests and send back a response with the operation status result and with the requested

[2]Github: https://github.com/eProsima/Micro-XRCE-DDS-Client

data, in the case of subscribe/reply operations. The communication in the DDS world is mediated by a dedicated ProxyClient in charge of creating the DDS Entities requested by the Clients, such as Participants, Topics, Publishers, and Subscribers, which can interact with the DDS Global dataspace. eProsima Micro XRCE-DDS provides the user

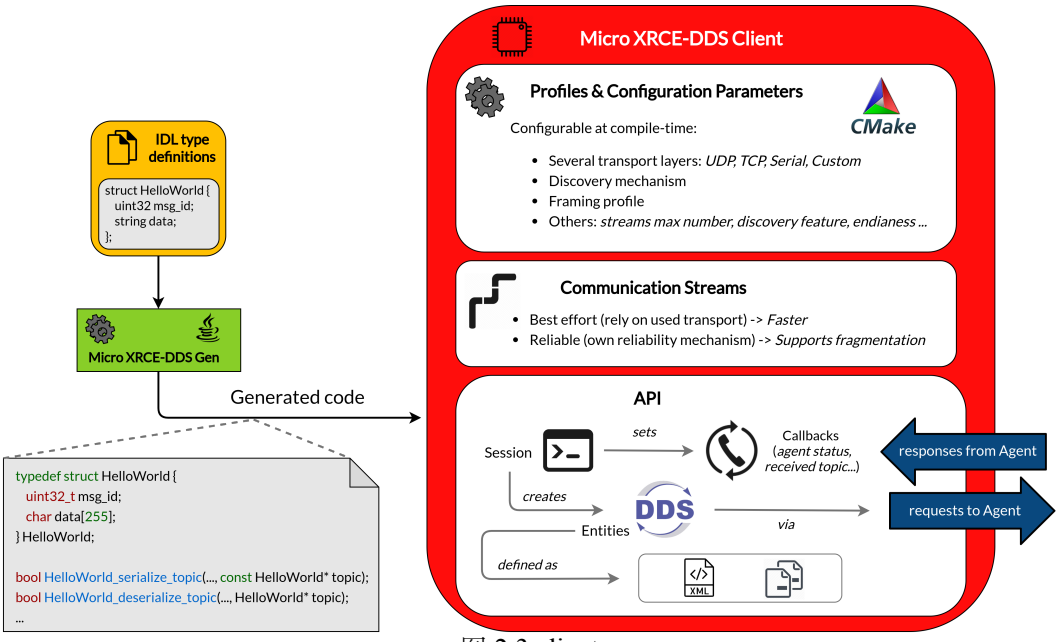

图 2.3 client

with a C API to create Micro XRCE-DDS Clients applications. The library can be configured at compile-time via a set of CMake flags allowing to enable or disable some profiles before compilation, and to manipulate several parameters controlling some of the library's functionalities, which in turn allow tuning the library size.

The communication between a Micro XRCE-DDS Client and a Micro XRCE-DDS Agent is achieved by means of several kinds of built-in transports: UDPv4, UDPv6, TCPv4, TCPv6 and Serial communication. In addition, there is the possibility for the user to generate its own Custom transport.

2.4.3 Micro XRCE-DDS Agent

The Micro XRCE-DDS Agent[3] is a broker which bridges the Clients with the DDS world.

2.5 USED POSIX API

[3]Github: https://github.com/eProsima/Micro-XRCE-DDS-Agent

第 3 章 Implementation of uROS

3.1 RCLC

This repository provides the *rclc* package, which complements the ROS Client Support Library (*rcl*) to make up a complete ROS 2 client library for the C programming language. That is, *rclc* does not add a new layer of types on top of *rcl* (like rclcpp and rclpy do) but only provides convenience functions that ease the programming with the *rcl* types. New types are introduced only for concepts that are missing in *rcl*, most important an Executor and a Lifecycle Node.

The API of the RCLC-Executor can be divided in several phases: **Configuration**, **Running** and **Clean-Up**.

3.1.1 Configuration Phase

During the configuration phase, the user shall define: the total number of callbacks, trigger condition (optional, default: ANY), data communication semantics (optional, default RCLCPP) and the processing sequence of the callbacks.

3.1.1.1 rclc_executor_t * rclc_get_zero_initialized_executor()

Returns a zero initialized executor object.

3.1.1.2 rclc_executor_init(rclc_executor_t * executor, rcl_context_t * context, const size_t number_of_handles, const rcl_allocator_t * allocator)

As the Executor is intended for embedded controllers, dynamic memory management is crucial. Therefore at initialization of the RCLC-Executor, the user defines the total number of handles *number_of_handles*. **A handle is a term for subscriptions, timers, services, clients and guard conditions.** The necessary dynamic memory will be allocated only in this phase and no more memory in the running phase. The corresponding wait-set is allocated in the first execution of the spin-method or in the optional call to *rclc_executor_prepare*. This makes this Executor static in the sense, that during runtime no heap allocations occur. You can add, however, at runtime as many handles, e.g. subscriptions, to the executor until the maximum number of handles is reached. The *context* is the RCL context, and *allocator* points to a memory allocator.

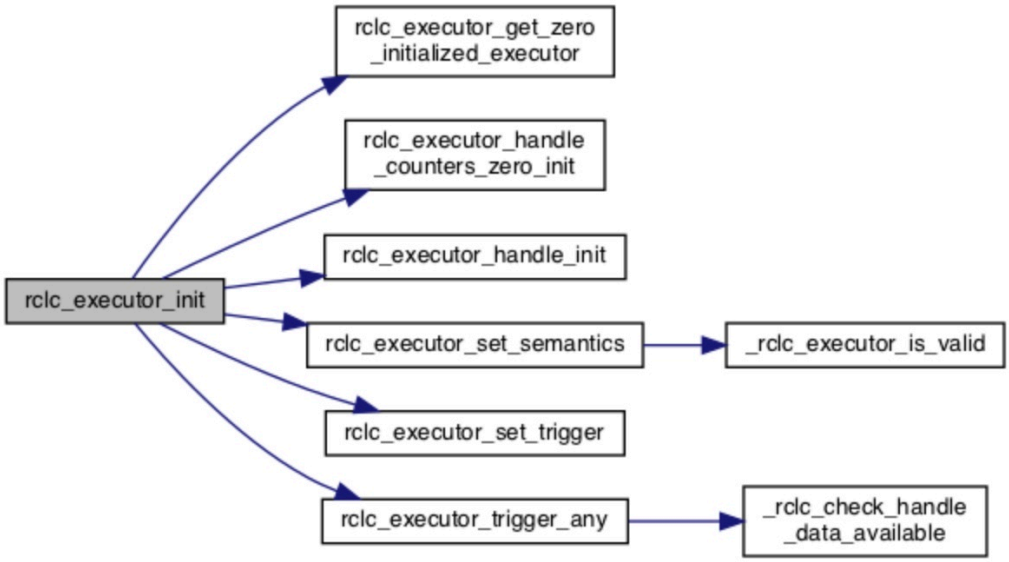

图 3.1 executor_init call

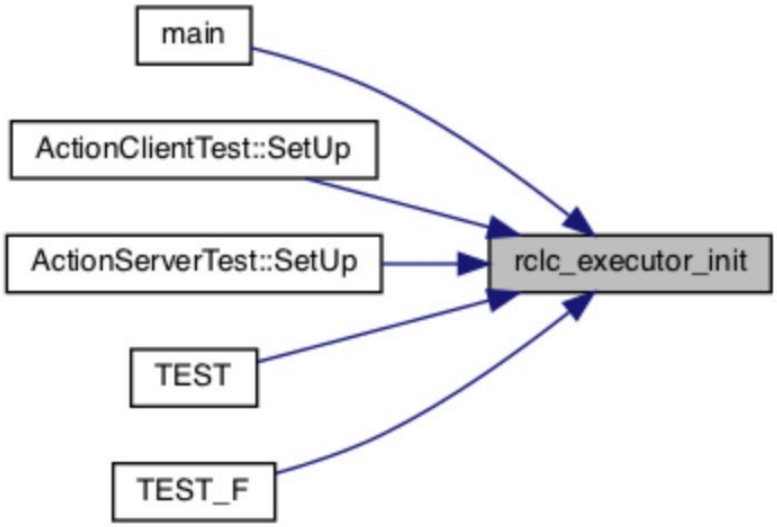

图 3.2 executor_init caller

3.1.1.3 rclc_executor_set_timeout(rclc_executor_t * executor, const uint64_t timeout_ns)

The timeout in nano-seconds *timeout_ns* for waiting for new data from the DDS-queue is specified in rclc_executor_set_timeout() (this is the timeout parameter for rcl_wait()).

3.1.1.4 rclc_executor_set_semantics(rclc_executor_t * executor, rclc_executor_semantics_t semantics)

The **data communication semantics** can either be *RCLCPP*(default) or *LET*.

To be compatible with ROS 2 rclcpp Executor, the existing rclcpp semantics is implemented with the option *RCLCPP*. That is, with the spin-function the DDS-queue is constantly monitored for new data (rcl_wait()). If new data becomes available, then

it is fetched from DDS (rcl_take()) immediately before the callback is executed. All callbacks are processed in the user-defined order, this is the only difference to the rclcpp Executor, in which the order can not be defined by the user.

The LET semantics is implemented such that at the beginning of processing, all available data is fetched (rcl_take()) and buffered and then the callbacks are processed in the pre-defined operating on the buffered copy.

3.1.1.5 rclc_executor_set_trigger(rclc_executor_t * executor, rclc_executor_trigger_t trigger_function, void * trigger_object)

The trigger condition rclc_executor_set_trigger defines when the processing of the callbacks shall start. For convenience some trigger conditions have been defined:

- *rclc_executor_trigger_any(default)*: start executing if any callback has new data
- *rclc_executor_trigger_all*: start executing if all callbacks have new data
- *rclc_executor_trigger_one(&data)*: start executing if *data* has been received
- *rclc_executor_trigger_always*: returns always true, that is every time the Executor spins, the processing of the callbacks is invocated. For example with *spin_period* and this trigger condition as well as specifying all callbacks of subscriptions being called as *ALWAYS*, a fixed period execution of all callbacks can be implemented, irrespective whether new data is available or not.
- *user_defined_function*: the user can also define its own function with more complex logic

With the *rclc_executor_trigger_one* trigger, the handle to trigger is specified with *trigger_object*. In the other cases of the trigger conditions this parameter shall be *NULL*.

3.1.1.6 rclc_executor_add_subscription(rclc_executor_t * executor, rcl_subscription_t * subscription, void * msg, rclc_subscription_callback_t callback, rclc_executor_handle_invocation_t invocation) AND rclc_executor_add_timer(rclc_executor_t * executor, rcl_timer_t * timer)

To add handles to the Executor, the functions rclc_executor_add_subscription() for subscriptions and rclc_executor_add_timer() for timers. The order in which these functions are called, defines later the sequential processing order during runtime.

For adding a subscription, the rcl subscription handle *subscription*, a pointer an allocated message *msg*, the message callback *callback* and an invocation option *invocation* need to be specified. **The invocation option specifies**, whether the callback shall be executed only if new data is available (*ON_NEW_DATA*) or if the callback shall always be executed (*ALWAYS*). The second option is useful for example when the callback is expected to be called at a fixed rate.

For a timer, only the rcl timer object *timer* is needed.

3.1.1.7 rclc_executor_prepare(rclc_executor_t * executor)

The function rclc_executor_prepare prepares the internal RCL wait set allocating the required dynamic memory. Its use is optional becouse it also will be checked in the spin functions. If used and no entities are added to the executor during running phase, no dynamic allocations are guaranteed during the running phase.

3.1.2 Running Phase

3.1.2.1 rclc_executor_spin_some (rclc_executor_t * executor, const uint64_t timeout_ns)

The function rclc_executor_spin_some checks for new data from the DDS queue once. **It first copies all data into local data structures and then executes all handles according the specified order.** This implements the LET semantics. Note that memory is dynamically allocated within rcl-layer, when DDS queue is accessed with rcl_wait_set_init().

The static-LET executor performs the following actions:

(1) initializes the `wait_set` with all handle of the array executor->handles

(2) waits for new data from DDS queue with rcl_wait() with timeout executor->timeout_ns

(3) takes all ready handles from the `wait_set` with rcl_take()

(4) processes all handles in the order, how they were added to the executor with the respective add-functions by calling respective callback (thus implementing first-read, process, semantic of LET)

3.1.2.2 rclc_executor_spin(rclc_executor_t * executor)

The function rclc_executor_spin calls rclc_executor_spin_some indefinitely as long as the ROS system is alive. This might create a high performance load on your processor.

3.1.2.3 rclc_executor_spin_period(rclc_executor_t * executor, const uint64_t period)

The function rclc_executor_spin_period calls rclc_executor_spin_some periodically (as defined with the argument period) as long as the ROS system is alive.

```c
rcl_ret_t
rclc_executor_spin_some(rclc_executor_t * executor, const uint64_t timeout_ns)
{
  rcl_ret_t rc = RCL_RET_OK;
  rclc_executor_prepare(executor);

  // 1. set rmw fields to NULL
  rc = rcl_wait_set_clear(&executor->wait_set);
  if (rc != RCL_RET_OK) {
    PRINT_RCLC_ERROR(rclc_executor_spin_some, rcl_wait_set_clear);
    return rc;
  }

  // 2. add handles to wait_set
  for (size_t i = 0; (i < executor->max_handles && executor->handles[i].initialized); i++) {
    switch (executor->handles[i].type) {
      case SUBSCRIPTION:
      case SUBSCRIPTION_WITH_CONTEXT:
        // add subscription to wait_set and save index
        rc = rcl_wait_set_add_subscription(
          &executor->wait_set, executor->handles[i].subscription,
          &executor->handles[i].index);
        break;

      case TIMER:
        // case TIMER_WITH_CONTEXT:
        // add timer to wait_set and save index
        rc = rcl_wait_set_add_timer(
          &executor->wait_set, executor->handles[i].timer,
          &executor->handles[i].index);
        break;
      ...
    }
  }

  // 3. wait up to 'timeout_ns' to receive notification about which handles reveived
  // new data from DDS queue.
  rc = rcl_wait(&executor->wait_set, timeout_ns);

  // 4. based on semantics process input data
  switch (executor->data_comm_semantics) {
    case LET:
      rc = _rclc_let_scheduling(executor);
      break;
    case RCLCPP_EXECUTOR:
      rc = _rclc_default_scheduling(executor);
      break;
    default:
      PRINT_RCLC_ERROR(rclc_executor_spin_some, unknown_semantics);
      return RCL_RET_ERROR;
  }

  return rc;
}
```

图 3.3 rclc_executor_spin_some code

3.1.2.4 rclc_executor_spin_one_period (rclc_executor_t * executor, const uint64_t period)

This is a function used by rclc_executor_spin_period to spin one time. The purpose is to test the accuracy of the spin_period function in the unit tests.

3.1.3 Clean-Up Phase

3.1.3.1 rclc_executor_fini ()

The function rlce_executor_fini frees the dynamically allocated memory of the executor.

3.1.4 RCL convenience functions

The rclc package also provides a number of convenience functions, which make it easier to create the RCL-objects *rcl_node_t*, *rcl_subscription_t*, *rcl_timer_t* and *rcl_publisher_t*. Convenience functions:

- rclc_support_init()
- rclc_support_init_with_options()
- rclc_support_fini()
- rclc_node_init_default()
- rclc_publisher_init_default()
- rclc_subscription_init_default()
- rclc_timer_init_default()

3.1.5 Examples

3.1.5.1 Example of real-time embedded application use-case

In embedded systems, real-time behavior is approached by using the time-triggered paradigm, which means that the processes are periodically activated. Processes can be assigned priorities to allow pre-emptions. Figure 3.4 shows an example, in which three processes with fixed periods are shown. The middle and lower process are preempted multiple times depicted with empty dashed boxes. To each process one or multiple tasks can be assigned, as shown in Figure 3.5. These tasks are executed sequentially, which is often called cooperative scheduling.

With sequential execution the co-operative scheduling of tasks within a process can be modeled. The trigger condition is used to periodically activate the process which will then execute all callbacks in a pre-defined order. Data will be communicated using the LET-semantics. Every Executor is executed in its own tread, to which an appropriate

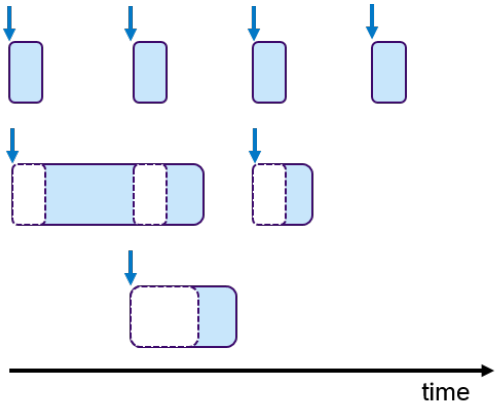

图 3.4 Fixed periodic preemptive scheduling

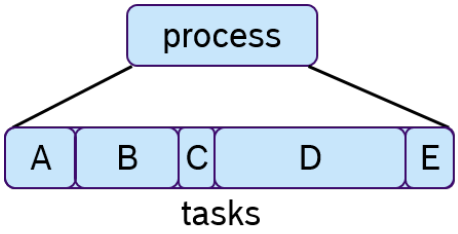

图 3.5 Processes with sequentially executed tasks

priority can be assigned.

In the following example, the Executor is setup with 4 handles. We assume a process has three subscriptions *sub1*, *sub2*, *sub3*. The sequential processing order is given by the order as they are added to the Executor. A timer *timer* defines the period. The *trigger_one* with the parameter timer is used, so that whenever the timer is ready, all callbacks are processed. Finally the data communication semantics LET is defined.

3.1.5.2 Example of sense-plan-act pipeline in mobile robotics

A common design paradigm in mobile robotics is a control loop, consisting of several phases: A sensing phase to acquire sensor data, a plan phase for localization and path planning and an actuation-phase to steer the mobile robot. Of course, more phases are possible, here these three phases shall serve as an example. Such a processing pipeline is shown in Figure 3.7. Typically multiple sensors are used to perceive the environment. For example an IMU and a laser scanner. The quality of localization algorithms highly depend on how old such sensor data is when it is processed. Ideally the latest data of all sensors should be processed. One way to achieve this is to execute first all sensor drivers in the sense-phase and then process all algorithms in the plan-phase.

In this example we want to realise a sense-plan-act pipeline in a single thread. The trigger condition is demonstrated by activating the sense-phase when both data for the Laser and IMU are available. Three executors are necessary *exe_sense*, *exe_plan* and *exe_act*. The two sensor acquisition callbacks *sense_Laser* and *sense_IMU* are registered in the Executor exe_sense. The trigger condition *ALL* is responsible to activate the sense-phase only when all data for these two callbacks are available. Finally all three Executors are spinning using a while-loop and the spin_some function.

```c
#include "rcl_executor/let_executor.h"

// define subscription callback
void my_sub_cb1(const void * msgin)
{
    // ...
}
// define subscription callback
void my_sub_cb2(const void * msgin)
{
    // ...
}
// define subscription callback
void my_sub_cb3(const void * msgin)
{
    // ...
}

// define timer callback
void my_timer_cb(rcl_timer_t * timer, int64_t last_call_time)
{
    // ...
}

// necessary ROS 2 objects
rcl_context_t context;
rcl_node_t node;
rcl_subscription_t sub1, sub2, sub3;
rcl_timer_t timer;
rcle_let_executor_t exe;

// define ROS context
context = rcl_get_zero_initialized_context();
// initialize ROS node
rcl_node_init(&node, &context,...);
// create subscriptions
rcl_subscription_init(&sub1, &node, ...);
rcl_subscription_init(&sub2, &node, ...);
rcl_subscription_init(&sub3, &node, ...);
// create a timer
rcl_timer_init(&timer, &my_timer_cb, ... );
// initialize executor with four handles
rclc_executor_init(&exe, &context, 4, ...);
// define static execution order of handles
rclc_executor_add_subscription(&exe, &sub1, &my_sub_cb1, ALWAYS);
rclc_executor_add_subscription(&exe, &sub2, &my_sub_cb2, ALWAYS);
rclc_executor_add_subscription(&exe, &sub3, &my_sub_cb3, ALWAYS);
rclc_executor_add_timer(&exe, &timer);
// trigger when handle 'timer' is ready
rclc_executor_set_trigger(&exe, rclc_executor_trigger_one, &timer);
// select LET-semantics
rclc_executor_data_comm_semantics(&exe, LET);
// spin forever
rclc_executor_spin(&exe);
```

图 3.6 real-time embedded application use-case

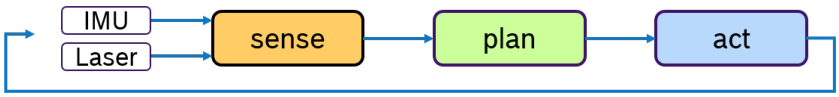

图 3.7 Multiple sensors driving a Sense-Plan-Act pipeline

```
...
rcl_subscription_t sense_Laser, sense_IMU, plan, act;
rcle_let_executor_t exe_sense, exe_plan, exe_act;
// initialize executors
rclc_executor_init(&exe_sense, &context, 2, ...);
rclc_executor_init(&exe_plan, &context, 1, ...);
rclc_executor_init(&exe_act, &context, 1, ...);
// executor for sense-phase
rclc_executor_add_subscription(&exe_sense, &sense_Laser, &my_sub_cb1, ON_NEW_DATA);
rclc_executor_add_subscription(&exe_sense, &sense_IMU, &my_sub_cb2, ON_NEW_DATA);
rclc_let_executor_set_trigger(&exe_sense, rclc_executor_trigger_all, NULL);
// executor for plan-phase
rclc_executor_add_subscription(&exe_plan, &plan, &my_sub_cb3, ON_NEW_DATA);
// executor for act-phase
rclc_executor_add_subscription(&exe_act, &act, &my_sub_cb4, ON_NEW_DATA);

// spin all executors
while (true) {
    rclc_executor_spin_some(&exe_sense);
    rclc_executor_spin_some(&exe_plan);
    rclc_executor_spin_some(&exe_act);
}
```

图 3.8 sense-plan-act pipeline in mobile robotics

3.1.5.3 Example of synchronization of multiple rates

Often multiple sensors are being used to sense the environment for mobile robotics. While an IMU sensor provides data samples at a very high rate (e.g. 500 Hz), laser scans are available at a much slower frequency (e.g. 10Hz) determined by the revolution time. Then the challenge is, how to deterministically fuse sensor data with different frequencies. This problem is depicted in Figure 3.9.

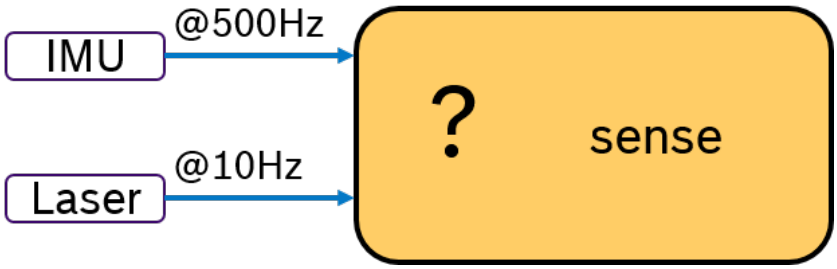

图 3.9 How to deterministically process multi-frequent sensor data

An Alternative would be to evaluate the IMU sample and the laser scan by synchronizing their frequency. For example by processing always 50 IMU samples with one laser scan. This approach is shown in Figure 3.10. A pre-processing callback aggregates the IMU samples and sends an aggregated message with 50 samples at 10Hz rate. Now both messages have the same frequency. With a trigger condition, which fires when both messages are available, the sensor fusion algorithm can expect always synchronized input data.

The sensor fusion synchronizing the multiple rates with a trigger is shown below.

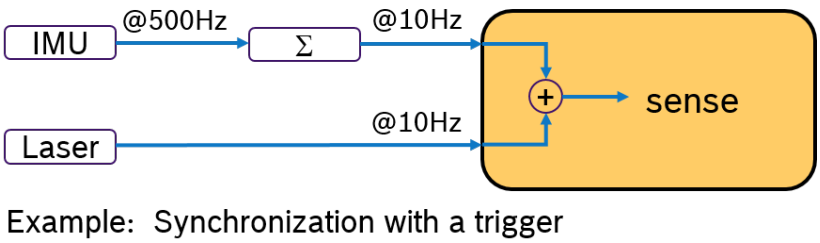

图 3.10 How to deterministically process multi-frequent sensor data

```
...
rcl_subscription_t aggr_IMU, sense_Laser, sense_IMU;
rcle_let_executor_t exe_aggr, exe_sense;
// initialize executors
rclc_executor_init(&exe_aggr, &context, 1, ...);
rclc_executor_init(&exe_sense, &context, 2, ...);
// executor for aggregate IMU data
rclc_executor_add_subscription(&exe_aggr, &aggr_IMU, &my_sub_cb1, ON_NEW_DATA);
// executor for sense-phase
rclc_executor_add_subscription(&exe_sense, &sense_Laser, &my_sub_cb2, ON_NEW_DATA);
rclc_executor_add_subscription(&exe_sense, &sense_IMU, &my_sub_cb3, ON_NEW_DATA);
rclc_executor_set_trigger(&exe_sense, rclc_executor_trigger_all, NULL);

// spin all executors
while (true) {
    rclc_executor_spin_some(&exe_aggr);
    rclc_executor_spin_some(&exe_sense);
}
```

图 3.11 synchronization of multiple rates

Another idea would be to actively request for IMU data only when a laser scan is received. This concept is shown in Figure 3.12. Upon arrival of a laser scan message, first, a message with aggregated IMU samples is requested. Then, the laser scan is processed and later the sensor fusion algorithm. An Executor, which would support sequential execution of callbacks, could realize this idea.

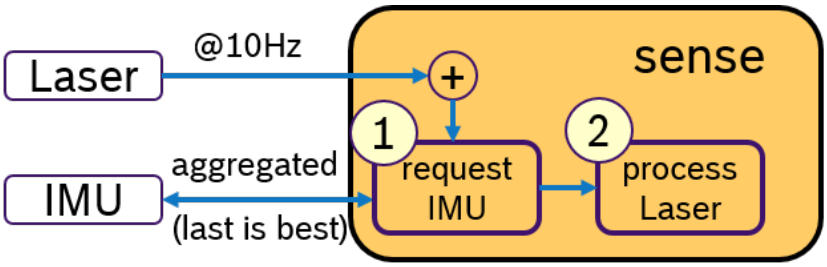

图 3.12 Synchronization of multiple input data with a trigger

The setup for the sensor fusion using sequential execution is shown below. Note, that the sequential order is sense_IMU, which will request the aggregated IMU message, and then sense_Laser while the trigger will fire, when a laser message is received.

```
...
rcl_subscription_t sense_Laser, sense_IMU;
rcle_let_executor_t exe_sense;
// initialize executor
rclc_executor_init(&exe_sense, &context, 2, ...);
// executor for sense-phase
rclc_executor_add_subscription(&exe_sense, &sense_IMU, &my_sub_cb1, ALWAYS);
rclc_executor_add_subscription(&exe_sense, &sense_Laser, &my_sub_cb2, ON_NEW_DATA);
rclc_executor_set_trigger(&exe_sense, rclc_executor_trigger_one, &sense_Laser);
// spin
rclc_executor_spin(&exe_sense);
```

图 3.13 synchronization of multiple rates

3.1.5.4 Example of high-priority processing path

Often a robot has to fulfill several activities at the same time. For example following a path and avoiding obstacles. While path following is a permanent activity, obstacle avoidance is triggered by the environment and should be immediately reacted upon. Therefore one would like to specify priorities to activities. This is depicted in Figure 3.14. Assuming a simplified control loop with the activities sense-plan-act, the obstacle avoidance, which might temporarily stop the robot, should be processed before the planning phase. In this example we assume that these activities are processed in one thread.

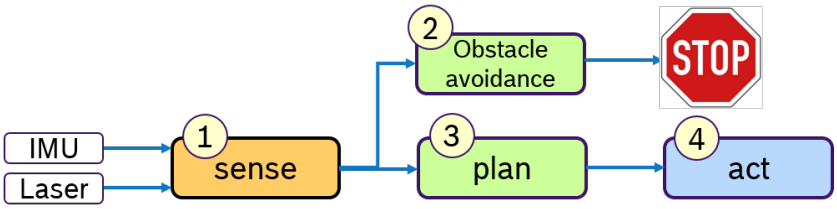

图 3.14 Managing high priority path with sequential order

This example shows the sequential processing order to execute the obstacle avoidance obst_avoid after the callbacks of the sense-phase and before the callback of the planning phase plan. The control loop is started when a laser message is received. Then an aggregated IMU message is requested, like in the example above. Then all the other callbacks are always executed. This assumes that these callbacks communicate via a global data structure. Race conditions cannot occur, because the callbacks run all in one thread.

```
...
rcl_subscription_t sense_Laser, sense_IMU, plan, act, obst_avoid;
rcle_let_executor_t exe;
// initialize executors
rclc_executor_init(&exe, &context, 5, ...);
// define processing order
rclc_executor_add_subscription(&exe, &sense_IMU, &my_sub_cb1, ALWAYS);
rclc_executor_add_subscription(&exe, &sense_Laser, &my_sub_cb2, ON_NEW_DATA);
rclc_executor_add_subscription(&exe, &obst_avoid, &my_sub_cb3, ALWAYS);
rclc_executor_add_subscription(&exe, &plan, &my_sub_cb4, ALWAYS);
rclc_executor_add_subscription(&exe, &act, &my_sub_cb5, ALWAYS);
rclc_executor_set_trigger(&exe, rclc_executor_trigger_one, &sense_Laser);
// spin
rclc_executor_spin(&exe);
```

图 3.15 high-priority processing path

```
static
rcl_ret_t
_rclc_let_scheduling(rclc_executor_t * executor)
{
  rcl_ret_t rc = RCL_RET_OK;

  // logical execution time
  // 1. read all input
  // 2. process
  // 3. write data (*) data is written not at the end of all callbacks, but it will not be
  //    processed by the callbacks 'in this round' because all input data is read in the
  //    beginning and the incoming messages were copied.

  // step 0: check for available input data from DDS queue
  // complexity: O(n) where n denotes the number of handles
  for (size_t i = 0; (i < executor->max_handles && executor->handles[i].initialized); i++) {
    rc = _rclc_check_for_new_data(&executor->handles[i], &executor->wait_set);
    if ((rc != RCL_RET_OK) && (rc != RCL_RET_SUBSCRIPTION_TAKE_FAILED)) {
      return rc;
    }
  }

  // if the trigger condition is fullfilled, fetch data and execute
  // complexity: O(n) where n denotes the number of handles
  if (executor->trigger_function(
      executor->handles, executor->max_handles,
      executor->trigger_object))
  {
    // step 1: read input data
    for (size_t i = 0; (i < executor->max_handles && executor->handles[i].initialized); i++) {
      rc = _rclc_take_new_data(&executor->handles[i], &executor->wait_set);
      if ((rc != RCL_RET_OK) && (rc != RCL_RET_SUBSCRIPTION_TAKE_FAILED)) {
        return rc;
      }
    }

    // step 2:  process (execute)
    for (size_t i = 0; (i < executor->max_handles && executor->handles[i].initialized); i++) {
      rc = _rclc_execute(&executor->handles[i]);
      if (rc != RCL_RET_OK) {
        return rc;
      }
    }
  }
  return rc;
}
```

图 3.16 _rclc_let_scheduling code

```c
static
rcl_ret_t
_rclc_default_scheduling(rclc_executor_t * executor)
{
  rcl_ret_t rc = RCL_RET_OK;

  for (size_t i = 0; (i < executor->max_handles && executor->handles[i].initialized); i++) {
    rc = _rclc_check_for_new_data(&executor->handles[i], &executor->wait_set);
    if ((rc != RCL_RET_OK) && (rc != RCL_RET_SUBSCRIPTION_TAKE_FAILED)) {
      return rc;
    }
  }
  // if the trigger condition is fullfilled, fetch data and execute
  if (executor->trigger_function(
      executor->handles, executor->max_handles,
      executor->trigger_object))
  {
    // take new input data from DDS-queue and execute the corresponding callback of the handle
    for (size_t i = 0; (i < executor->max_handles && executor->handles[i].initialized); i++) {
      rc = _rclc_take_new_data(&executor->handles[i], &executor->wait_set);
      if ((rc != RCL_RET_OK) && (rc != RCL_RET_SUBSCRIPTION_TAKE_FAILED) &&
        (rc != RCL_RET_SERVICE_TAKE_FAILED))
      {
        return rc;
      }
      rc = _rclc_execute(&executor->handles[i]);
      if (rc != RCL_RET_OK) {
        return rc;
      }
    }
  }
  return rc;
}
```

图 3.17 _rclc_let_scheduling code

3.1.6 Kernel Functions

3.1.6.1 _rclc_let_scheduling (rclc_executor_t * executor)

3.1.6.2 _rclc_default_scheduling (rclc_executor_t * executor)

3.1.6.3 _rclc_check_for_new_data (rclc_executor_handle_t * handle, rcl_wait_set_t * wait_set)

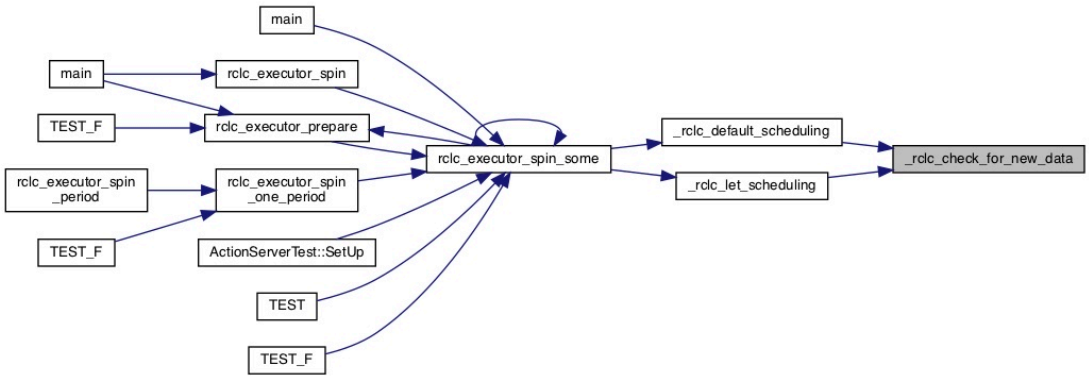

图 3.18 _rclc_let_scheduling caller

3.1.6.4 _rclc_take_new_data (rclc_executor_handle_t * handle, rcl_wait_set_t * wait_set)

3.1.6.5 _rclc_execute (rclc_executor_handle_t * handle)

```
static
rcl_ret_t
_rclc_check_for_new_data(rclc_executor_handle_t * handle, rcl_wait_set_t * wait_set)
{
  rcl_ret_t rc = RCL_RET_OK;

  switch (handle->type) {
    case SUBSCRIPTION:
    case SUBSCRIPTION_WITH_CONTEXT:
      handle->data_available = (NULL != wait_set->subscriptions[handle->index]);
      break;
    case TIMER:
      ...
    case SERVICE:
    case SERVICE_WITH_REQUEST_ID:
    case SERVICE_WITH_CONTEXT:
      ...
    case CLIENT:
    case CLIENT_WITH_REQUEST_ID:
      ...
    case GUARD_CONDITION:
      ...
    case ACTION_CLIENT:
      ...
    case ACTION_SERVER:
      ...

    default:
      RCUTILS_LOG_DEBUG_NAMED(
        ROS_PACKAGE_NAME, "Error in _rclc_check_for_new_data:wait_set unknwon handle type: %d",
        handle->type);
      return RCL_RET_ERROR;
  }    // switch-case
  return rc;
}

```

图 3.19 _rclc_let_scheduling code

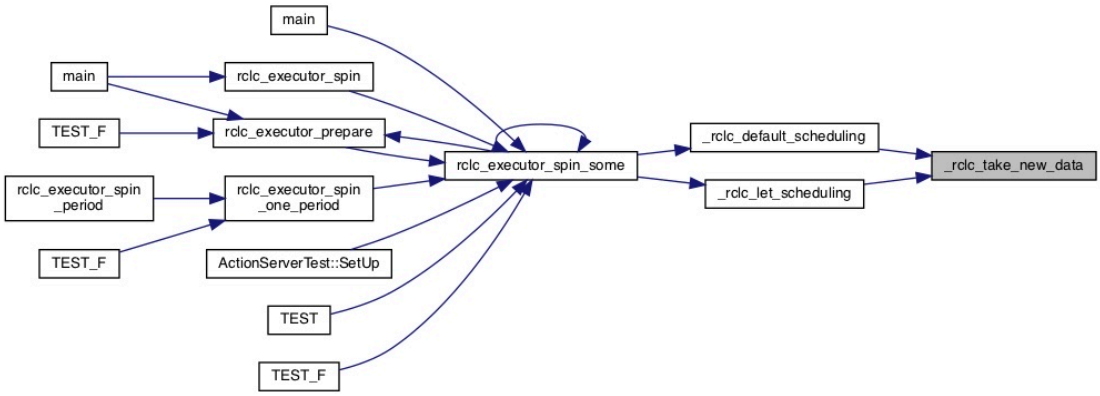

图 3.20 _rclc_let_scheduling caller

```
static
rcl_ret_t
_rclc_take_new_data(rclc_executor_handle_t * handle, rcl_wait_set_t * wait_set)
{
  rcl_ret_t rc = RCL_RET_OK;

  switch (handle->type) {
    case SUBSCRIPTION:
    case SUBSCRIPTION_WITH_CONTEXT:
      if (wait_set->subscriptions[handle->index]) {
        rmw_message_info_t messageInfo;
        rc = rcl_take(
          handle->subscription, handle->data, &messageInfo,
          NULL);
        if (rc != RCL_RET_OK) {
          // rcl_take might return this error even with successfull rcl_wait
          if (rc != RCL_RET_SUBSCRIPTION_TAKE_FAILED) {
            PRINT_RCLC_ERROR(rclc_take_new_data, rcl_take);
            RCUTILS_LOG_ERROR_NAMED(ROS_PACKAGE_NAME, "Error number: %d", rc);
          }
          // invalidate that data is available, because rcl_take failed
          if (rc == RCL_RET_SUBSCRIPTION_TAKE_FAILED) {
            handle->data_available = false;
          }
          return rc;
        }
      }
      break;
    ...
  }
}
```

图 3.21 _rclc_let_scheduling code

```c
static
rcl_ret_t
_rclc_execute(rclc_executor_handle_t * handle)
{
  rcl_ret_t rc = RCL_RET_OK;
  bool invoke_callback = false;
  // determine, if callback shall be called
  if (handle->invocation == ON_NEW_DATA &&
    _rclc_check_handle_data_available(handle)) {
    invoke_callback = true;
  }
  if (handle->invocation == ALWAYS) {
    invoke_callback = true;
  }
  // execute callback
  if (invoke_callback) {
    switch (handle->type) {
      case SUBSCRIPTION:
        if (handle->data_available) {
          handle->subscription_callback(handle->data);
        } else {
          handle->subscription_callback(NULL);
        }
        break;

      case SUBSCRIPTION_WITH_CONTEXT:
        ...

      case TIMER:
        // case TIMER_WITH_CONTEXT:
        rc = rcl_timer_call(handle->timer);

        // cancled timer are not handled, return success
        if (rc == RCL_RET_TIMER_CANCELED) {
          rc = RCL_RET_OK;
          break;
        }

        if (rc != RCL_RET_OK) {
          PRINT_RCLC_ERROR(rclc_execute, rcl_timer_call);
          return rc;
        }
        break;
      ...
    }
  }
}
```

图 3.22 _rclc_execute code

3.2 RCL

3.2.1 RCL Layer Structures

3.2.1.1 WORKING: wait.h

1. `rcl_wait_set_t`:

```c
/// Container for subscription's, guard condition's, etc to be waited on.
typedef struct rcl_wait_set_t
{
  /// Storage for subscription pointers.
  const rcl_subscription_t ** subscriptions;
  /// Number of subscriptions
  size_t size_of_subscriptions;
  /// Storage for guard condition pointers.
  const rcl_guard_condition_t ** guard_conditions;
  /// Number of guard_conditions
  size_t size_of_guard_conditions;
  /// Storage for timer pointers.
  const rcl_timer_t ** timers;
  /// Number of timers
  size_t size_of_timers;
  /// Storage for client pointers.
  const rcl_client_t ** clients;
  /// Number of clients
  size_t size_of_clients;
  /// Storage for service pointers.
  const rcl_service_t ** services;
  /// Number of services
  size_t size_of_services;
  /// Storage for event pointers.
  const rcl_event_t ** events;
  /// Number of events
  size_t size_of_events;
  /// Implementation specific storage.
  struct rcl_wait_set_impl_t * impl;
} rcl_wait_set_t;
```

图 3.23 Structure: rcl_wait_set_t

2. `rcl_wait_set_impl_t`:

3.2.2 RCL Layer APIs

3.2.2.1 WORKING: wait.h

Wait sets for waiting on messages/service requests and responses/timers to be ready.

1. rcl_wait_set_init (rcl_wait_set_t * wait_set, size_t number_of_subscriptions, size_t number_of_guard_conditions, size_t number_of_timers, size_t number_of_clients, size_t number_of_services, size_t number_of_events, rcl_context_t * context, rcl_allocator_t allocator): Initialize a rcl wait set with space for items to be waited on. This function allocates space for the subscriptions and other wait-able entities that can be stored in the wait set. It also sets the allocator to the given allocator and initializes the pruned member to be false. The `wait_set struct` should be allocated and initialized to `NULL`.

```
typedef struct rcl_wait_set_impl_t
{
  // number of subscriptions that have been added to the wait set
  size_t subscription_index;
  rmw_subscriptions_t rmw_subscriptions;
  // number of guard_conditions that have been added to the wait set
  size_t guard_condition_index;
  rmw_guard_conditions_t rmw_guard_conditions;
  // number of clients that have been added to the wait set
  size_t client_index;
  rmw_clients_t rmw_clients;
  // number of services that have been added to the wait set
  size_t service_index;
  rmw_services_t rmw_services;
  // number of events that have been added to the wait set
  size_t event_index;
  rmw_events_t rmw_events;

  rmw_wait_set_t * rmw_wait_set;
  // number of timers that have been added to the wait set
  size_t timer_index;
  // context with which the wait set is associated
  rcl_context_t * context;
  // allocator used in the wait set
  rcl_allocator_t allocator;
} rcl_wait_set_impl_t;
```

图 3.24 Structure: rcl_wait_set_impl_t

If the `wait_set` is allocated but the memory is uninitialized the behavior is undefined. Calling this function on a wait set that has already been initialized will result in an error. A wait set can be reinitialized if rcl_wait_set_fini () was called on it.

2. rcl_wait_set_add_subscription (rcl_wait_set_t * wait_set, const rcl_subscription_t * subscription, size_t * index): Store a pointer to the given subscription in the next empty spot in the set. This function does not guarantee that the subscription is not already in the wait set. Also add the rmw representation to the underlying rmw array and increment the rmw array count.

3. rcl_wait (rcl_wait_set_t * wait_set, int64_t timeout): Block until the wait set is ready or until the timeout has been exceeded. This function will collect the items in the `rcl_wait_set_t` and pass them to the underlying rmw_wait () function. The items in the wait set will be either left untouched or set to `NULL` after this function returns. Items that are not NULL are ready, where ready means different things based on the type of the item. For subscriptions this means there may be messages that can be taken, or perhaps that the state of the subscriptions has changed, in which case rcl_take () may succeed but return with taken == false. For guard conditions this means the guard condition was triggered.

3.2.2.2 WORKING: graph.h

3.2.2.3 WORKING: init.h

3.2.2.4 WORKING: guard_condition.h

3.3 RMW

3.3.1 RMW Layer Structures

3.3.1.1 WORKING: wait

1. `rmw_wait_set_t`: Container for guard conditions to be waited on.

```c
/// Container for guard conditions to be waited on
typedef struct RMW_PUBLIC_TYPE rmw_wait_set_t
{
  /// The name of the rmw implementation
  const char * implementation_identifier;

  /// The guard condition to be waited on
  rmw_guard_conditions_t * guard_conditions;

  /// Type erased pointer to this wait set's data
  void * data;
} rmw_wait_set_t;
```

图 3.25 Structure: rmw_wait_set_t

2. `rmw_guard_conditions_t`: Array of guard condition handles. An array of void * pointers representing type-erased middleware-specific guard conditions. The number of non-null entries may be smaller than the allocated size of the array. The number of guard conditions represented may be smaller than the allocated size of the array. The creator of this structure is responsible for allocating and de-allocating the array.

```c
/// Array of guard condition handles.
/**
 * An array of void * pointers representing type-erased middleware-specific guard conditions.
 * The number of non-null entries may be smaller than the allocated size of the array.
 * The number of guard conditions represented may be smaller than the allocated size of the array.
 * The creator of this struct is responsible for allocating and deallocating the array.
 */
typedef struct RMW_PUBLIC_TYPE rmw_guard_conditions_t
{
  /// The number of guard conditions represented by the array.
  size_t guard_condition_count;
  /// Pointer to an array of void * pointers of guard conditions.
  void ** guard_conditions;
} rmw_guard_conditions_t;
```

图 3.26 Structure: rmw_guard_conditions_t

3.3.2 RMW Layer APIs

3.3.2.1 INIT: rmw_init.c

TODO: HERE!!!!!

3.3.2.2 WORKING: wait.h

1. rmw_wait (rmw_subscriptions_t * subscriptions, rmw_guard_conditions_t * guard_conditions, rmw_services_t * services, rmw_clients_t * clients, rmw_events_t * events, rmw_wait_set_t * wait_set, const rmw_time_t * wait_timeout): Waits on sets of different entities and returns when one is ready. This function adds middleware-specific conditions to the wait set and waits until one or more become ready, or until the timeout is reached. [1] Called API uxr_run_session_until_data () in XRCE-DDS layer.

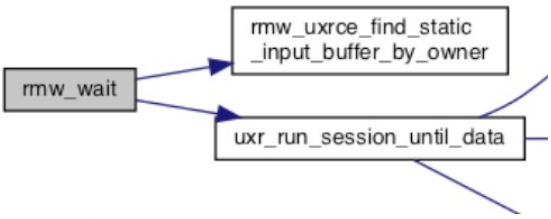

图 3.27 Function code: rmw_wait()

3.4 XRCE-DDS

3.4.1 XRCE-DDS Layer Structures

3.4.1.1 session.h

1. `uxrSession`:

3.4.2 XRCE-DDS Layer APIs

3.4.2.1 session.c

1. uxr_run_session_until_data (uxrSession* session, int timeout_ms): Keeps communication between the Client and the Agent. This function involves the following actions:

(1) flushing all the output streams sending the data through the transport (e.g., UCP, UART…). This actions will be performed in a loop until a data message is received or the `timeout` is exceeded.

[1] Elapsed time is measured against the system clock. Timeout granularity is thus bound to that of the aforementioned clock and, depending on the underlying implementation, to that of platform-specific APIs to sleep and/or wait. **The amount of time this function actually waits may be either above or below the specified timeout.**

```
typedef struct uxrSession
{
    uxrSessionInfo info;
    uxrStreamStorage streams;
    struct uxrCommunication* comm;

    const uint16_t* request_list;
    uint8_t* status_list;
    size_t request_status_list_size;

    uxrOnStatusFunc on_status;
    void* on_status_args;

    uxrOnTopicFunc on_topic;
    void* on_topic_args;

    uxrOnTimeFunc on_time;
    void* on_time_args;
    int64_t time_offset;
    bool synchronized;

    uxrOnRequestFunc on_request;
    void* on_request_args;

    uxrOnReplyFunc on_reply;
    void* on_reply_args;

    bool on_data_flag;
    uxrContinuousArgs continuous_args;

#ifdef UCLIENT_PROFILE_MULTITHREAD
    uxrMutex mutex;
#endif // ifdef UCLIENT_PROFILE_MULTITHREAD

#ifdef PERFORMANCE_TESTING
    uxrOnPerformanceFunc on_performance;
    void* on_performance_args;
#endif // ifdef PERFORMANCE_TESTING
} uxrSession;
```

图 3.28 Structure: uxrSession

(2) listening messages from the Agent calling the associated callback (topic, status, request and replies).

```
bool uxr_run_session_until_data(
        uxrSession* session,
        int timeout_ms)
{
    UXR_LOCK_SESSION(session);

    int64_t start_timestamp = uxr_millis();
    int remaining_time = timeout_ms;

    uxr_flash_output_streams(session);

    session->on_data_flag = false;
    do
    {
        listen_message_reliably(session, remaining_time);
        if (session->on_data_flag)
        {
            break;
        }
        remaining_time = timeout_ms - (int)(uxr_millis() - start_timestamp);
    }
    while (remaining_time > 0);

    UXR_UNLOCK_SESSION(session);

    return session->on_data_flag;
}
```

图 3.29 Function code: uxr_run_session_until_data()

As can be seen in function uxr_run_session_until_data (), it first calls UXR_LOCK_SESSION (session), which finally invokes xSemaphoreTakeRecursive (mutex->impl, portMAX_DELAY) on FreeRTOS, or pthread_mutex_lock (&mutex->impl) on POSIX platform.

2. uxr_stream_id (uxrSession* session, int timeout_ms): TODO: HERE!!!!!!!!!!!!!!!!!!

3. uxr_prepare_best_effort_buffer_to_send (uxrSession* session, int timeout_ms):

4. uxr_stamp_session_header (uxrSession* session, int timeout_ms):

3.5 USED POSIX API

参考文献

[1] Group O M. Dds interoperability wire protocol[EB/OL]. 2014[March 9, 2021]. https://www.omg.org/spec/DDSI-RTPS/2.2/.

[2] OpenDDS. Opendds[EB/OL]. 2019[March 9, 2021]. https://www.opendds.org.

[3] Innovations R T. Connextdds[EB/OL]. 2019[March 9, 2021]. https://www.rti.com/.

[4] Kampmann A, Wustenberg A, Alrifaee B, et al. A Portable Implementation of the Real-Time Publish-Subscribe Protocol for Microcontrollers in Distributed Robotic Applications[C]// 0Confs. 2019: 448. DOI: 10.1109/ITSC.2019.8916835.

[5] Group O M. Dds for extremely resource constrained environments[EB/OL]. 2019[March 9, 2021]. https://www.omg.org/spec/DDS-XRCE/1.0/Beta2.