ROS 2: Launch System, Time Handling & Synchronization, Actions

Course: Robotic Programming Environments

Michał Drwięga
michal.drwiega@pwr.edu.pl
www.mdrwiega.com/edu/rpe

Department of Cybernetics and Robotics
Wrocław Univeristy of Science and Technology

For educational use only ©2026 Michał Drwięga

Wrocław University
of Science and Technology

Content

1. ROS 2 Launch System
2. Time Sources in ROS 2
3. Time Synchronization
4. Actions

For educational use only
©2026 Michał Drwięga

ROS 2 Launch System

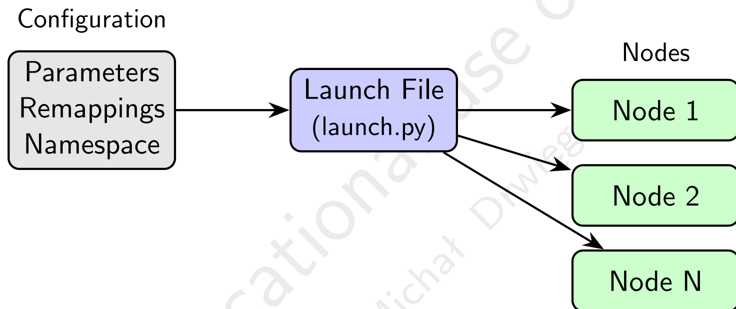
For educational use only
©2026 Michał Dziwiega

Launch System: Motivation

- Modern robotic systems consist of **multiple interacting nodes**.
- Manual startup becomes:
  - Error-prone
  - Hard to reproduce
  - Difficult to scale
- Need for a unified way to:
  - Start and configure multiple nodes simultaneously
  - Manage parameters and namespaces
  - Coordinate complex system behavior
- Enables:
  - **Reproducibility** of experiments
  - **Automation** of system startup
  - **Modularity** and reuse of configurations

For educational use only ©2026 Michał Drwięga

ROS 2 Launch System



What it does:

- Starts multiple nodes with **one command**
- Sets **parameters, namespaces, and topic remappings**
- Supports **conditional logic** and **event handlers**

For educational use only ©2026 Michał Dwileg

Launchers - code example to run just one node (Python)

```
1  from launch import LaunchDescription
2  from launch_ros.actions import Node
3
4  def generate_launch_description():
5      node1 = Node(
6          package="turtlesim",
7          executable="turtlesim_node",
8          name='turtlesim1'
9      )
10
11     return LaunchDescription([
12         node1
13     ])
```

Launching Multiple Nodes

```
1  from launch import LaunchDescription
2  from launch_ros.actions import Node
3
4  def generate_launch_description():
5      return LaunchDescription([
6          Node(
7              package='demo_nodes_cpp',
8              executable='talker',
9              name='talker_node'
10         ),
11         Node(
12             package='demo_nodes_cpp',
13             executable='listener',
14             name='listener_node'
15         )
16     ])
```

Remapping Topics

```
Node(
    package='demo_nodes_cpp',
    executable='talker',
    remappings=[
        ('/chatter', '/my_topic')
    ]
)
```

- Allows changing topic names without modifying code
- Useful for:
  - Integration of multiple subsystems
  - Avoiding topic conflicts

For educational use only ©2026 Mic

Using Parameters from File

```
1 Node(
2     package='my_package',
3     executable='my_node',
4     parameters=['config/params.yaml']
5 )
```

- YAML files used for structured configuration
- Example YAML:

```
1 my_node:
2   ros__parameters:
3     speed: 1.0
4     use_sim_time: true
```

Using Launch Arguments

```python
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument
from launch.substitutions import LaunchConfiguration
from launch_ros.actions import Node

def generate_launch_description():
    arg = DeclareLaunchArgument(
        'use_sim_time', default_value='false'
    )
    return LaunchDescription([
        arg,
        Node(
            package='demo_nodes_cpp',
            executable='talker',
            parameters=[{'use_sim_time': LaunchConfiguration(
                'use_sim_time')}]
        )
    ])
```

Including Other Launch Files

```
1 from launch.actions import IncludeLaunchDescription
2 from launch.launch_description_sources import
      PythonLaunchDescriptionSource
3
4 IncludeLaunchDescription(
5     PythonLaunchDescriptionSource(
6         'path/to/other_launch.py'
7     )
8 )
```

- Enables modular launch design
- Useful for large systems
- Can pass arguments to included launch files

Conditional Launching

```
from launch.conditions import IfCondition
from launch.substitutions import LaunchConfiguration

Node(
    package='demo_nodes_cpp',
    executable='talker',
    condition=IfCondition(
        LaunchConfiguration('enable_talker')
    )
)
```

- Nodes can be started conditionally
- Controlled via launch arguments

Running a Launch File

- Python launch file:

```
ros2 launch <package> <file.launch.py>
```

- XML launch file:

```
ros2 launch <package> <file.launch.xml>
```

- Pass arguments at runtime:

```
ros2 launch demo_nodes_cpp talker_launch.py
use_sim_time:=true
```

Best Practices

- Use Python launch files for flexibility
- Keep launch files modular (use includes)
- Store parameters in YAML files
- Use arguments for configurable systems
- Avoid hardcoding paths (use substitutions)

For educational use only
©2026 Michał Drwięga

Time Sources in ROS 2

Epoch Time (Linux / time.time())

- Epoch time: seconds elapsed since 1970-01-01 00:00:00 UTC.
- Represents "real-world" wall-clock time.
- Can jump forward/backward due to system clock changes
- Ideal for:
  - Timestamps
  - Logging
  - Scheduling with absolute time
- Example in Python:

```python
import time

epoch_seconds = time.time()
print("Seconds:", epoch_seconds)
```

Epoch Time Size (Memory Representation)

- Common representations:
  - **32-bit integer (4 bytes)**
    - Range: 1901 – 2038
    - Known issue: **Year 2038 problem**
  - **64-bit integer (8 bytes)**
    - Very large range (billions of years)
    - Standard in modern systems
  - **Floating point (8 bytes)**
    - Used in Python: `time.time()`
    - Includes fractional seconds
- In robotics / ROS 2 often stored as:
  - `int64` (nanoseconds), or
  - separate `sec` + `nsec`

Time Formats (Human-Readable vs Machine)

- Two main representations of time:
  - **Machine-readable** (numeric)
    - Epoch time (e.g., 1713018600)
    - Efficient for computation and comparison
  - **Human-readable** (formatted strings)
    - Example: `2026-04-13 14:30:00`
    - Easier for users and debugging
- Common standard: **ISO 8601**
  - Format: `YYYY-MM-DDTHH:MM:SS`
  - Example: `2026-04-13T14:30:00`
- Example in Python:

```python
import time

# Convert epoch to readable time
t = time.time()
readable = time.strftime("%Y-%m-%d %H:%M:%S",
                         time.localtime(t))

print(readable)
```

Why Time Abstraction in ROS 2?

The Problem

- Robotic algorithms depend heavily on **synchronization** and **periodic execution**.
- **Real Hardware:** Uses system wall clock (steady/epoch time).
- **Simulation (Gazebo/Ignition):** Time may run faster, slower, or be paused.

The Solution

ROS 2 provides a **Time Source API** to switch between "Real Time" and "Simulated Time" **without changing a single line of algorithm code.**

The Three Core ROS 2 Clocks

System Clock (`RCL_SYSTEM_TIME`)

- Wall-clock time (OS time)
- Used for logging and external synchronization

Steady Clock (`RCL_STEADY_TIME`)

- Monotonic (never jumps backward)
- Ideal for measuring time intervals and durations

ROS Clock (`RCL_ROS_TIME`)

- Default for message timestamps
- Can be overridden via `/clock`
- Enables simulation and rosbag replay

Best practice: Use ROS time for algorithm logic to ensure consistent behavior in real and simulated systems.

For educational use only ©2026 Michał Drwięga

How Time Sources Work Internally

The ROS-Time Override Mechanism:

1. Node checks parameter `use_sim_time`.
2. If `True`, the Node subscribes to **Topic:** `/clock` (Type: `Clock`).
3. The Node **ignores** the system's real-time clock (RTC) and uses the timestamp received on `/clock`.

```cpp
// C++ Example: Node waits for valid simulation time before spinning
while (rclcpp::ok() && this->get_clock()->now().seconds() == 0.0) {
    // Waiting for first /clock message...
}
```

Setting Up a Simulation Time Source (Publisher)

Responsibility of the Simulator (Gazebo / Ignition):

- The simulator **publishes** the current virtual time to `/clock`.

Important

Without a node publishing to `/clock`, all simulation nodes will see `time = 0` forever.

Configuring Nodes to Listen (Subscribers)

Nodes must explicitly opt-in to simulation time.

Method 1: Command Line

```
ros2 run my_pkg my_node \
  --ros-args -p use_sim_time
     :=true
```

Method 2: Launch File

```
Node(
    package='my_package',
    executable='my_node',
    parameters=[{
        'use_sim_time': True
    }]
)
```

Critical Pitfall: The Time Jump

The Problem

When `use_sim_time` is `true`, `Node::now()` returns **0** until the first `/clock` message arrives.

- Control loops fail silently at startup.

The Fix: Wait for Valid Time

```
// Block until simulation time is non-zero
if (this->get_parameter("use_sim_time").as_bool()) {
    while (this->get_clock()->now().nanoseconds() == 0) {
        rclcpp::sleep_for(std::chrono::milliseconds(10));
    }
}
```

ROS 2 Time vs. Real-Time Kernel

Common Misconception

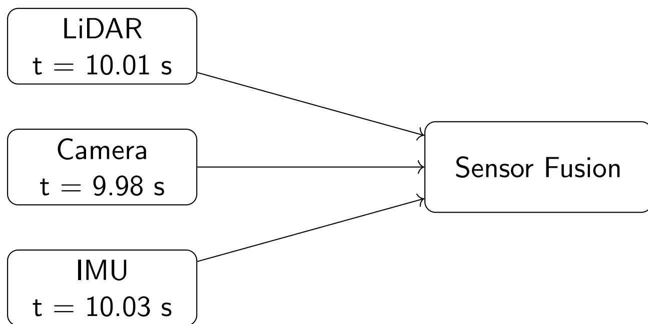
"Using ROS_TIME makes my node real-time safe."

Reality:

- **Time Source** is a software abstraction for **synchronization**.
- **Real-Time Performance** requires:
  1. Real-time kernel (PREEMPT_RT).
  2. Memory locking.
  3. Using `RCL_STEADY_TIME` (Monotonic) inside real-time callbacks (avoiding the `/clock` subscription lock inside the critical path).

Time Synchronization

Why Time Synchronization Matters in Robotics



Problem:

- Sensors operate at different rates
- Each can have its own clock
- Data arrives with delays

Consequence:

- Incorrect sensor fusion
- Distorted environment perception
- Poor localization

Solution: Use a common time reference (e.g., hardware clock, NTP) or estimate and apply time offsets.

What is NTP?

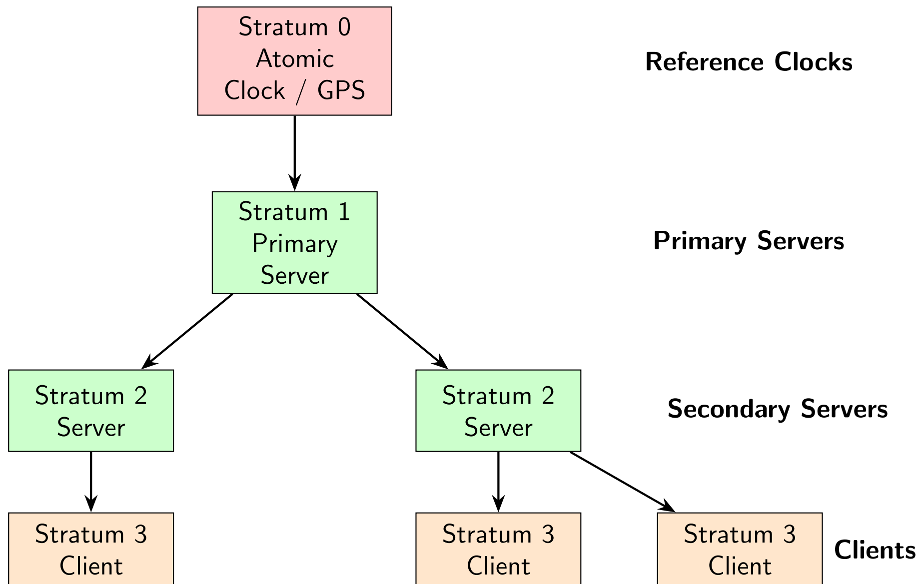
Definition

NTP (Network Time Protocol) is a time synchronisation protocol for network equipment which has a tree-like architecture.

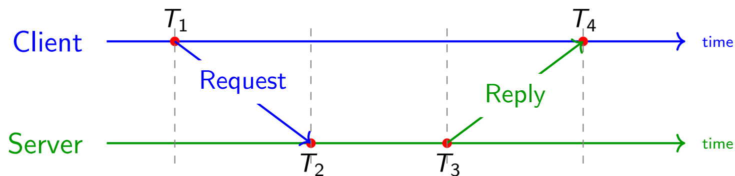
Key Facts

- Developed by David Mills at University of Delaware (1985)
- **Hierarchical system**. Each level of the hierarchy is called a stratum. Stratum levels (0 = reference clock, 1 = primary server, ...)
- Current version: NTPv4 (RFC 5905, 2010)
- Port: UDP/123
- Accuracy: milliseconds over Internet, microseconds in LAN

NTP: Stratum Hierarchy



How NTP Works - Timing Exchange



Four Timestamps

T_1: Client sends request
T_2: Server receives request
T_3: Server sends reply
T_4: Client receives reply

Calculations

Round-trip delay:

$$\delta = (T_4 - T_1) - (T_3 - T_2)$$

Clock offset:

$$\theta = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}$$

Clock Adjustment

$\theta > 0$: Client is **fast** $\rightarrow$ slow down

$\theta < 0$: Client is **slow** $\rightarrow$ speed up

Chrony: Time Synchronization for ROS 2

What is Chrony?

- Modern NTP client for Linux
- More robust and faster than traditional `ntpd`
- Keeps system clocks accurate even with intermittent network or variable latency

How it works:

- Synchronizes the system clock with remote NTP servers
- Continuously measures and compensates for clock drift
- Can act as a server to synchronize other machines in a network

For educational use only
©2026 Michał Drwięga

ROS 2 Time Workflow

What is an RTC (Real-Time Clock)?

- A hardware clock that keeps track of the current time even when the computer is powered off
- Typically battery-backed (or super capacitor)
- Provides a stable reference for system boot and timestamp consistency

Workflow:

1. On boot, system clock is set from RTC
2. Chrony/NTP synchronizes system clock with network time
3. ROS 2 nodes read either system time (`use_sim_time=false`) or simulation time (`use_sim_time=true`)

For educational use only
©2026 Michał Drwięga

Example: Multicamera System Synchronization

Objective: Ensure all cameras capture frames at the same time for accurate data fusion and analysis.

Why Synchronization Matters:

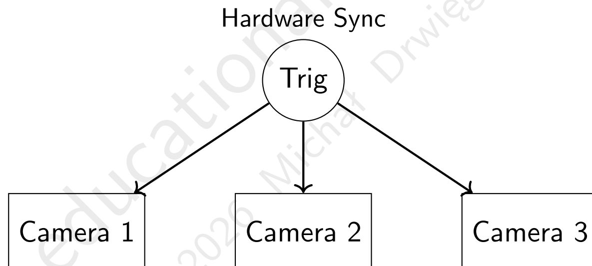
- Consistent temporal alignment between camera streams
- Accurate 3D reconstruction and depth estimation
- Reliable motion tracking and sensor fusion

For educational use only
©2026 Michał Drwięga

Multicamera System Synchronization

Synchronization Methods:

- **Hardware Triggering**
  - External trigger signal (e.g., TTL pulse)
  - High precision, low latency



- **Software Synchronization**
  - Timestamp-based alignment
  - Easier to implement, but less precise

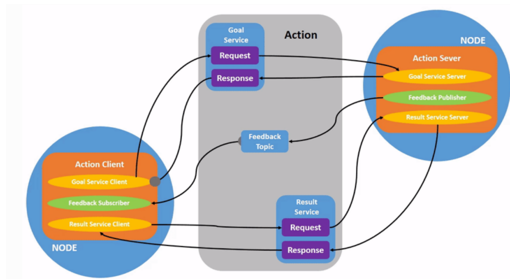
For educational use only ©2026 Michał Drwięga

Actions

For educational use only
©2026 Michał Drwięga

Communication - actions

- Use a client-server model
- Functionality similar to services
- Preemptable (can be cancelled during the execution)
- Provides feedback during the execution
- An action client sends a goal to an action server that acknowledges the goal and returns a feedback and a result



[1]

[1] https://docs.ros.org

Communication - actions

Actions: Lifecycle

- Goal State Transitions
- Feedback loop until result
- Goal succeeded, aborted, or canceled

Action Messages

Goal

```
int32 order
float32 duration
```

Feedback

```
float32 percent_complete
string status
```

Result

```
bool success
float32 final_position
```

- Three-part message structure
- Custom message types for actions

Action Client Example (Python)

```
import rclpy
from rclpy.action import ActionClient
from example_interfaces.action import Fibonacci

def feedback_callback(feedback_msg):
    print('Feedback:', feedback_msg.feedback.sequence)

def main():
    rclpy.init()
    node = rclpy.create_node('fibonacci_client')
    client = ActionClient(node, Fibonacci, 'fibonacci')

    goal_msg = Fibonacci.Goal(order=5)
    future = client.send_goal_async(goal_msg,
        feedback_callback=feedback_callback)

    rclpy.spin_until_future_complete(node, future)
    print('Result:', future.result().result.sequence)
    node.destroy_node()
```

Action Server Example (Python)

```
1  import rclpy
2  from rclpy.action import ActionServer
3  from example_interfaces.action import Fibonacci
4
5  def callback(goal_handle):
6      feedback_msg = Fibonacci.Feedback()
7      sequence = [0, 1]
8      for i in range(2, goal_handle.request.order):
9          sequence.append(sequence[i-1] + sequence[i-2])
10         feedback_msg.sequence = sequence
11         goal_handle.publish_feedback(feedback_msg)
12     goal_handle.succeed()
13     result = Fibonacci.Result()
14     result.sequence = sequence
15     return result
16
17 rclpy.init()
18 node = rclpy.create_node('fibonacci_server')
19 server = ActionServer(node, Fibonacci, 'fibonacci', callback)
20 rclpy.spin(node)
```

Communication mechanisms - comparison

Topics

- Purpose: continuous data streams, e.g. sensor data
- Many to many connection
- Data might be published and subscribed independently (at any time)

Services

- Purpose: remote procedure calls that can be executed quickly e.g. getting the current battery state
- One to one connection

Actions

- Purpose: remote procedure calls that runs for a longer time but provides feedback during the execution e.g. robot movement
- One to one connection, can be preempted

Summary

1. ROS 2 Launch System
2. Time Sources in ROS 2
3. Time Synchronization
4. Actions

The end

Thank you for your attention!