UNIVERSITÀ
DI TRENTO
Department of
Information Engineering and Computer Science

ROS 2 Development

Edoardo Lamon

Software Development for Collaborative Robots

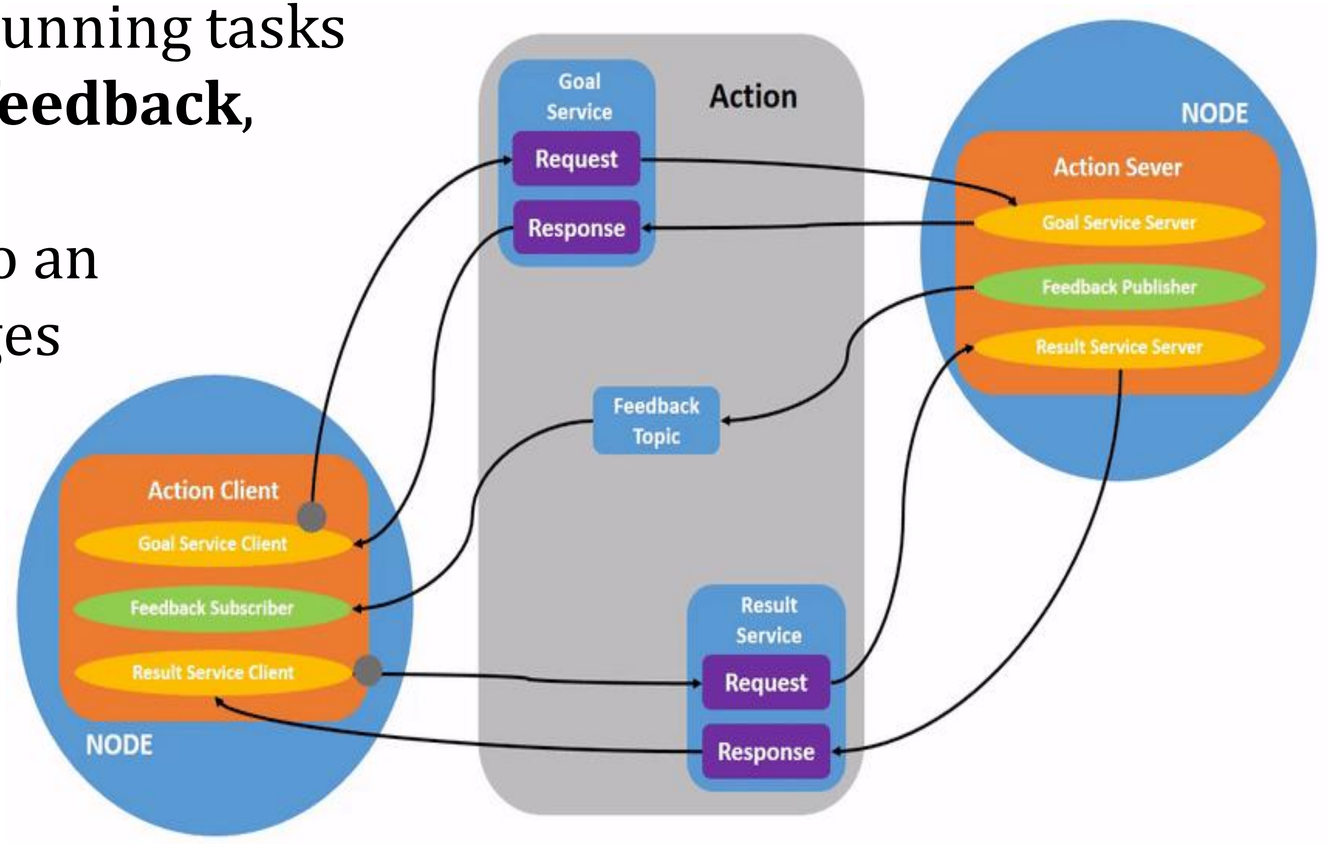
Academic Year 2025/26

ROS 2 Nodes

Actions: clients and servers

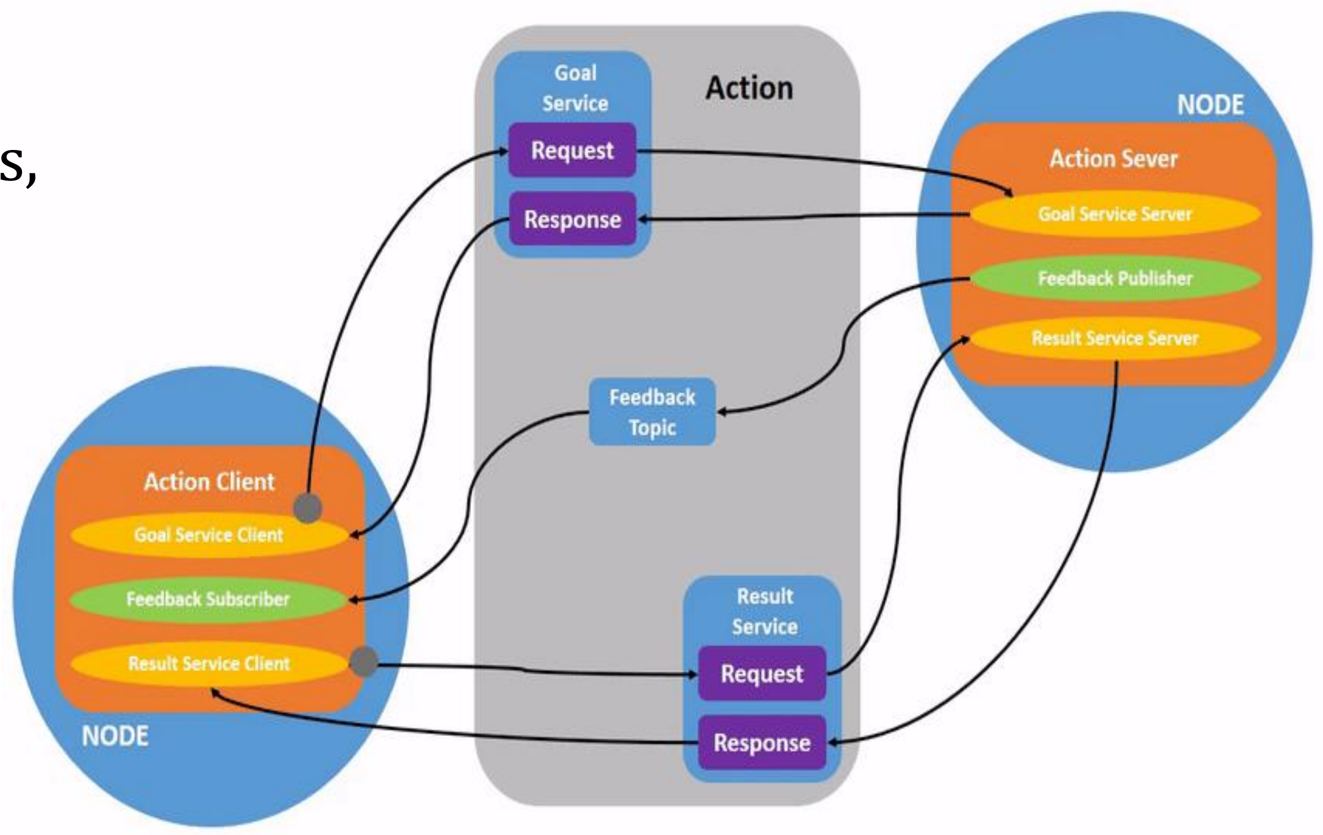
ROS 2 Actions

- ROS (2) actions are intended for long running tasks
- Actions consist of three parts: a **goal**, **feedback**, and a **result**
- An “**action client**” node sends a goal to an “**action server**” node that acknowledges the goal and returns a stream of feedback and a result
- Each goal runs **asynchronously** (server spawns a thread)
- Threads don’t block the **executor**



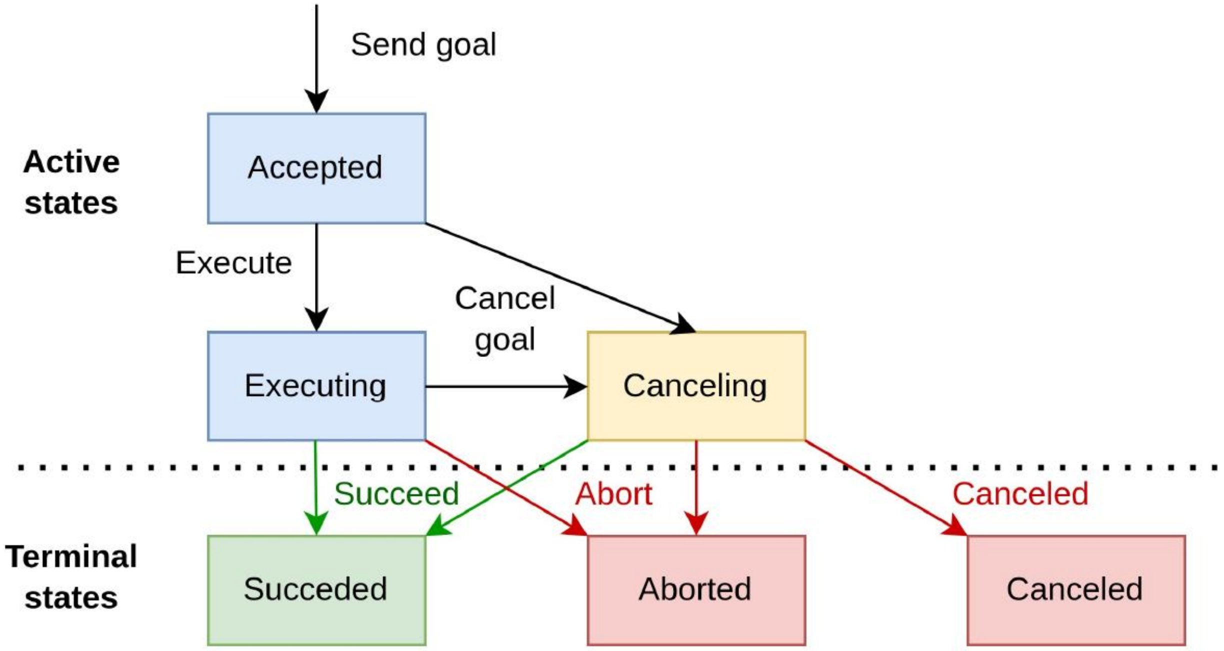
ROS 2 Actions

- Actions are built on *topics* and *services*
- Their functionality is similar to services, except **actions can be canceled**. They also provide steady **feedback**, as opposed to services which return a single response (essential for long-running robotic tasks).
- Actions require **more code** than topics or services but offer more control



ROS 2 Actions

- Any time an action server receives a goal from a client, it can decide if accepting or rejecting it.
- If accepting it, the server creates a new state machine for the goal:



State	Description / Transition
`accepted → executing`	Goal accepted, action running
`executing → succeeded`	Task completed successfully
`executing → aborted`	Task failed
`executing → canceled`	Client requested cancel

Custom ROS 2 Actions

- As for messages and services, also action shave a specific format (.action) and standard destination folder (*action)*.

- A *request* message is sent from an action client to an action server initiating a new goal.
- A *result* message is sent from an action server to an action client when a goal is done.
- *Feedback* messages are periodically sent from an action server to an action client with updates about a goal.

```
# Request
---
# Result
---
# Feedback
```

Custom ROS 2 Actions

- Let’s compute the Fibonacci sequence!
- Create an *action* directory in your custom msgs package and then the file Fibonacci.action:

```
mkdir action
touch Fibonacci.action
```

- The goal request is the *order* of the Fibonacci sequence we want to compute, the result is the final *sequence*, and the feedback is the *partial_sequence* computed so far

```
int32 order
---
int32[] sequence
---
int32[] partial_sequence
```

Compile the Custom Action

- In the *CmakeLists.txt:*

```
find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
 "action/Fibonacci.action“
)
```

- In the *package.xml:*

```
<buildtool_depend>rosidl_default_generators</buildtool_depend>
<depend>action_msgs</depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

Test the Custom Action

- After we compiled, test from the command line:

```
# Source our workspace
. install/setup.bash

# Check that our action definition exists
ros2 interface show <action_pkg_name>/action/Fibonacci
```

Writing the Action Server

An action server requires 6 things:

1. The templated action type name: *Fibonacci*.
2. A ROS 2 node to add the action to: *this*.
3. The action name: *'fibonacci'*.
4. A callback function for handling goals: *handle_goal.*
5. A callback function for handling cancellation: *handle_cancel*.
6. A callback function for handling goal acceptance: *handle_accept*.

Writing the Action Server

```
#include "rclcpp/rclcpp.hpp"
#include "rclcpp_action/rclcpp_action.hpp"

#include "calculator_msgs/action/fibonacci.hpp"

class FibonacciActionServer : public rclcpp::Node
{
public:
  using Fibonacci = calculator_msgs::action::Fibonacci;
  using GoalHandleFibonacci = rclcpp_action::ServerGoalHandle<Fibonacci>;

  FibonacciActionServer(const rclcpp::NodeOptions & options = rclcpp::NodeOptions())
    : Node("fibonacci_action_server_node", options){  }

private:
  rclcpp_action::Server<Fibonacci>::SharedPtr action_server_;


};  // class FibonacciActionServer
```

```
int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<FibonacciActionServer>());
  rclcpp::shutdown();
  return 0;
}
```

Writing the Action Server

```
class FibonacciActionServer : public rclcpp::Node
{
public:
  using Fibonacci = calculator_msgs::action::Fibonacci;
  using GoalHandleFibonacci = rclcpp_action::ServerGoalHandle<Fibonacci>;

  FibonacciActionServer(const rclcpp::NodeOptions & options = rclcpp::NodeOptions()): Node("fibonacci_action_server_node", options)
  {
    using namespace std::placeholders;

    this->action_server_ = rclcpp_action::create_server<Fibonacci>(
      this,
      "fibonacci",
      std::bind(&FibonacciActionServer::handle_goal, this, _1, _2),
      std::bind(&FibonacciActionServer::handle_cancel, this, _1),
      std::bind(&FibonacciActionServer::handle_accepted, this, _1));}
```

Writing the Action Server

```
  // Callback for handling new goals
 rclcpp_action::GoalResponse handle_goal(
  const rclcpp_action::GoalUUID & uuid,
  std::shared_ptr<const Fibonacci::Goal> goal)
 {
  RCLCPP_INFO(this->get_logger(), "Received goal request with order %d", goal->order);
  (void)uuid;
  return rclcpp_action::GoalResponse::ACCEPT_AND_EXECUTE;
 }

 // Callback for dealing with cancellation requests
 rclcpp_action::CancelResponse handle_cancel(
  const std::shared_ptr<GoalHandleFibonacci> goal_handle)
 {
  RCLCPP_INFO(this->get_logger(), "Received request to cancel goal");
  (void)goal_handle;
  return rclcpp_action::CancelResponse::ACCEPT;}
```

```
  // Callback for handling accepted goals and processing them.
  // Since the execution is a long-running operation, we spawn off a thread to do the actual work and return from handle_accepted quickly
  void handle_accepted(const std::shared_ptr<GoalHandleFibonacci> goal_handle)
  {
   using namespace std::placeholders;
   // this needs to return quickly to avoid blocking the executor, so spin up a new thread
   std::thread{std::bind(&FibonacciActionServer::execute, this, _1), goal_handle}.detach();
  }
```

Writing the Action Server

```
 // This work thread processes one sequence number of the Fibonacci sequence every second,
 // publishing a feedback update for each step. When it has finished processing,
 // it marks the goal_handle as succeeded, and quits.
 void execute(const std::shared_ptr<GoalHandleFibonacci> goal_handle)
 {
   RCLCPP_INFO(this->get_logger(), "Executing goal");
   rclcpp::Rate loop_rate(1);
   const auto goal = goal_handle->get_goal();
   auto feedback = std::make_shared<Fibonacci::Feedback>();
   auto & sequence = feedback->partial_sequence;
   sequence.push_back(0);
   sequence.push_back(1);
   auto result = std::make_shared<Fibonacci::Result>();
```

```
 for (int i = 1; (i < goal->order) && rclcpp::ok(); ++i) {
   // Check if there is a cancel request
   if (goal_handle->is_canceling()) {
     result->sequence = sequence;
     goal_handle->canceled(result);
     RCLCPP_INFO(this->get_logger(), "Goal canceled");
     return;
   }

   // Update sequence
   sequence.push_back(sequence[i] + sequence[i - 1]);
   // Publish feedback
   goal_handle->publish_feedback(feedback);
   RCLCPP_INFO(this->get_logger(), "Publish feedback");

   loop_rate.sleep();
 }

 // Check if goal is done
 if (rclcpp::ok()) {
   result->sequence = sequence;
   goal_handle->succeed(result);
   RCLCPP_INFO(this->get_logger(), "Goal succeeded");}
}
```

Compiling the Action Server

In the CMakeLists.txt, add:

```
find_package(rclcpp_action REQUIRED)

add_executable(fibonacci_action_server src/fibonacci_action_server.cpp)
ament_target_dependencies(fibonacci_action_server rclcpp rclcpp_action <action_pkg_name>)

install(TARGETS
  fibonacci_action_server
  DESTINATION lib/${PROJECT_NAME}
)
```

Running the Action Server

- After building:

```
# Source our workspace
. install/setup.bash

# Run the server node
ros2 run <server_pkg_name> fibonacci_action_server

# Verify from the terminal
ros2 action send_goal /fibonacci <action_pkg_name>/action/Fibonacci order:\ 7\

ros2 action send_goal --feedback /fibonacci <action_pkg_name>/action/Fibonacci
order:\ 7\
```

Writing the Action Client

An action client requires 3 things:

1. The templated action type name: *Fibonacci*.
2. A ROS 2 node to add the action client to: *this*.
3. The action name: *'fibonacci'*.
4. (Optional) Response, feedback, and result callbacks.

Writing the Action Client

```cpp
#include "rclcpp/rclcpp.hpp"
#include "rclcpp_action/rclcpp_action.hpp"

#include "calculator_msgs/action/fibonacci.hpp"

class FibonacciActionClient : public rclcpp::Node
{
public:
  using Fibonacci = calculator_msgs::action::Fibonacci;
  using GoalHandleFibonacci = rclcpp_action::ClientGoalHandle<Fibonacci>;

  FibonacciActionClient(const rclcpp::NodeOptions & options = rclcpp::NodeOptions())
  : Node("fibonacci_action_client_node", options){  }

private:
  rclcpp_action::Client<Fibonacci>::SharedPtr client_ptr_;
  rclcpp::TimerBase::SharedPtr timer_;
};  // class FibonacciActionClient
```

```cpp
int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<FibonacciActionClient>());
  rclcpp::shutdown();
  return 0;
}
```

Writing the Action Client

```
class FibonacciActionClient : public rclcpp::Node
{
public:
  using Fibonacci = calculator_msgs::action::Fibonacci;
  using GoalHandleFibonacci = rclcpp_action::ClientGoalHandle<Fibonacci>;

  FibonacciActionClient(const rclcpp::NodeOptions & options = rclcpp::NodeOptions())
  : Node("fibonacci_action_client_node", options)
  {
    this->client_ptr_ = rclcpp_action::create_client<Fibonacci>(this, "fibonacci");

    this->timer_ = this->create_wall_timer(
      std::chrono::milliseconds(500),
      std::bind(&FibonacciActionClient::send_goal, this));
  }
```

Writing the Action Client

```
void send_goal()  {
   using namespace std::placeholders;

   // Cancels the timer (so the function is only called once)
   this->timer_->cancel();

   // Waits for the action server to come up
   if (!this->client_ptr_->wait_for_action_server()) {
     RCLCPP_ERROR(this->get_logger(), "Action server not available after waiting");
     rclcpp::shutdown();
   }

   // Instantiates a new Fibonacci::Goal
   auto goal_msg = Fibonacci::Goal();
   goal_msg.order = 10;
   RCLCPP_INFO(this->get_logger(), "Sending goal");
```

```
  // Sets the response, feedback, and result callbacks
   auto send_goal_options =
rclcpp_action::Client<Fibonacci>::SendGoalOptions();
   send_goal_options.goal_response_callback =
     std::bind(&FibonacciActionClient::goal_response_callback, this, _1);
   send_goal_options.feedback_callback =
     std::bind(&FibonacciActionClient::feedback_callback, this, _1, _2);
   send_goal_options.result_callback =
     std::bind(&FibonacciActionClient::result_callback, this, _1);
   //Sends the goal to the server
   this->client_ptr_->async_send_goal(goal_msg, send_goal_options);
   // this->client_ptr_->async_send_goal(goal_msg);
  }
```

Writing the Action Client

```
private:
  rclcpp_action::Client<Fibonacci>::SharedPtr client_ptr_;
  rclcpp::TimerBase::SharedPtr timer_;

  void goal_response_callback(const GoalHandleFibonacci::SharedPtr & goal_handle){
    if (!goal_handle) {
      RCLCPP_ERROR(this->get_logger(), "Goal was rejected by server");
    } else {
      RCLCPP_INFO(this->get_logger(), "Goal accepted by server, waiting for result"); }

  void feedback_callback(
    GoalHandleFibonacci::SharedPtr,
    const std::shared_ptr<const Fibonacci::Feedback> feedback){
    std::stringstream ss;
    ss << "Next number in sequence received: ";
    for (auto number : feedback->partial_sequence) {
      ss << number << " ";}
    RCLCPP_INFO(this->get_logger(), ss.str().c_str());}
```

```
void result_callback(const GoalHandleFibonacci::WrappedResult & result)
{
  switch (result.code) {
    case rclcpp_action::ResultCode::SUCCEEDED:
      break;
    case rclcpp_action::ResultCode::ABORTED:
      RCLCPP_ERROR(this->get_logger(), "Goal was aborted");
      return;
    case rclcpp_action::ResultCode::CANCELED:
      RCLCPP_ERROR(this->get_logger(), "Goal was canceled");
      return;
    default:
      RCLCPP_ERROR(this->get_logger(), "Unknown result code");
      return;
  }
  std::stringstream ss;
  ss << "Result received: ";
  for (auto number : result.result->sequence) {
    ss << number << " ";
  }
  RCLCPP_INFO(this->get_logger(), ss.str().c_str());
  rclcpp::shutdown();
}
```

Compiling the Action Client

In the CMakeLists.txt, add:

```
add_executable(fibonacci_action_client src/fibonacci_action_client.cpp)
ament_target_dependencies(fibonacci_action_clientrclcpp rclcpp_action <action_pkg_name>)

install(TARGETS
  fibonacci_action_server
  fibonacci_action_client
  DESTINATION lib/${PROJECT_NAME}
)
```

Running the Action Client

After building:

```
# Source our workspace
. install/setup.bash

# Run the server node
ros2 run <server_pkg_name> fibonacci_action_server

# Run the client node
ros2 run <server_pkg_name> fibonacci_action_client
```