Topics

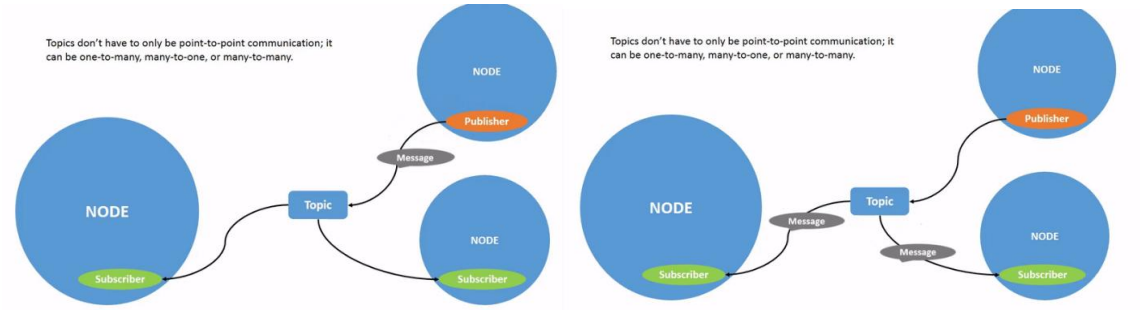
- 토픽은 노드 간 메시지를 전달하는 버스의 역할을 한다.

- 토픽은 송신- 수신 형식으로 메시지를 전달한다.

- 토픽은 지속적으로 데이터를 업데이트 받는다.

- 토픽은 송신자가 누군지 알 수 없다(원한다면 알아낼 수는 있다)

- 토픽을 사용하는 송신 노드는 수신 노드들에게 동시에 메시지를 전달할 수 있다.



Service

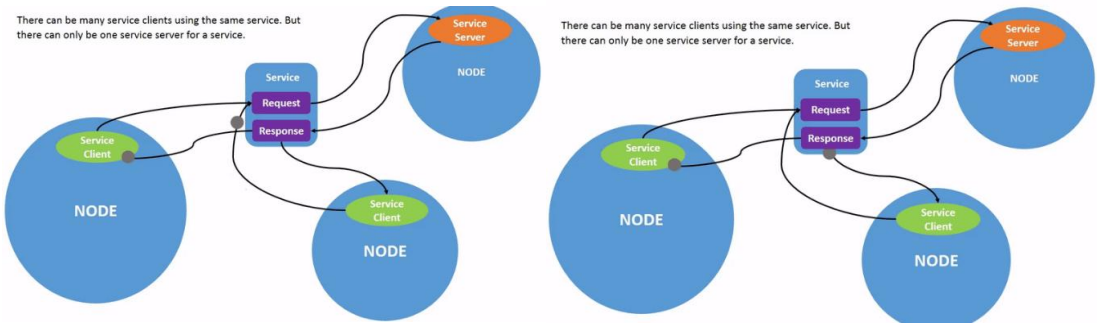
- 서비스는 호출-응답 형식으로 메시지를 전달한다.

- 서비스는 호출이 없는 이상 응답을 하지 않는다.

- 단시간에 응답이 오는 프로그램에서 사용하기 좋다.

- 서비스의 Server는 요청을 수락하고 요청에 대한 계산을 수행하는 요소다.

- 서비스의 Client는 Server에 계산 수행을 요청하고 계산 결과를 반환할 때까지 기다리는 요소다.



Parameters

- 파라미터는 노드의 설정 값이다.

- 코드 변경 없이 시작시, 런타임 중에도 노드를 구성하는데 사용된다.

- 파라미터는 노드 이름, 노드 네임스페이스, 파라미터 이름, 파라미터 네임스페이스로 처리를 한다.

- 노드를 정수, 실수, 문자열, 불, 벡터 형태로 설정할 수 있다.

Actions

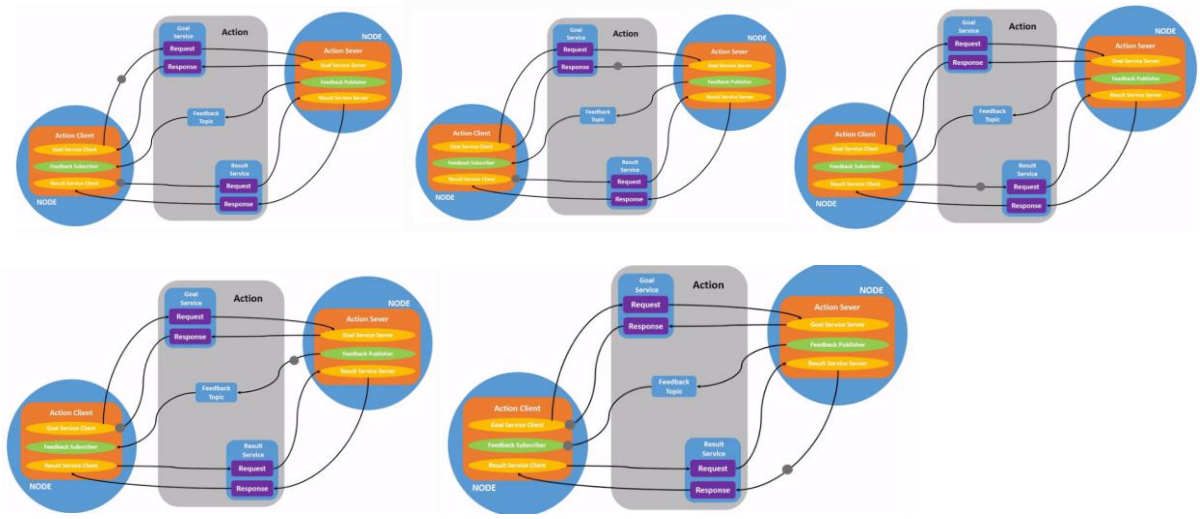
- 장기 작업을 위한 통신 방법이다.

- 목표, 피드백, 결과 세 부분으로 구성된다.

- 토픽과 서비스를 기반으로 만들어졌다.

- 서비스와 달리 액션은 요청을 취소할 수 있다.

- 지속적인 피드백을 제공한다.



과정: Client 노드에서 목표에 대한 요청을 보낸다. -> Server 노드에서 목표에 대한 응답을 보낸다. -> Client 노드에서 결과에 대한 요청을 보낸다. -> Server 노드는 결과 요청에 따라 연산을 수행한다. -> 수행하는 동안 Topic을 이용해 피드백을 보내 작업이 진행 중임을 Client 노드에게 알린다. -> 작업이 끝나면 결과를 Client 노드에 보낸다.

Publisher 설정법

1. Ros2를 빌드할 파일에 src 디렉토리를 만든다.
2. Src 디렉토리 안에 패키지를 만든다.

   (ros2 pkg create --build-type ament_cmake --license Apache-2.0 cpp_pubsub)
3. 패키지 내 src 파일에 CPP 파일을 생성한다.
4. Publisher 코드를 작성한다. (추가 설명은 아래에)
5. CmakeLists.txt와 package.xml을 수정한다.

Subscriber 설정법

1. Publisher에서 만든 패키지 내 src 폴더 안에 CPP 파일을 만든다.
2. Subscriber 코드를 작성한다. (추가 설명은 아래에)
3. CmakeLists.txt와 package.xml을 수정한다.

실행법

1. Ros2가 빌드된 디렉토리에 다시 빌드해 변동 사항을 업데이트한다.
2. Source 설정을 하여 경로설정을 해준다.
3. 명령어를 이용해 서로 다른 터미널에 Publisher와 Subscriber을 실행한다.
4. Topic이 잘 작동하는지 확인한다.

Service 설정법

1. Publisher와 동일하게 수행한다.
2. Publisher와 다르게 파라미터를 설정하기 위해 패키지 내에 .srv파일 생성한다.

   (src와 같은 위치)
3. CmakeLists.txt와 package.xml을 수정한다.

Client 설정법

1. Subscriber와 동일하게 수행한다.
2. CmakeLists.txt와 package.xml을 수정한다.

실행법

- Publisher Subscriber와 동일하다.

코드설명(Publisher)

```
#include <chrono>
#include <functional>
#include <memory>
#include <string>

#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"

using namespace std::chrono_literals;
```

헤더파일, ROS2 시스템에 일반적인 부분을 사용할 수 있는rclcpp/rclcpp.hpp 헤더파일 , 데이터 게시용 메시지 유형 헤더파일 설정한다.(사용을 위해 Cmakelists.txt와 package.xml 수정)

```
class MinimalPublisher : public rclcpp::Node
{
  public:
    MinimalPublisher()
    : Node("minimal_publisher"), count_(0)
    {
      publisher_ = this->create_publisher<std_msgs::msg::String>("topic", 10);
      timer_ = this->create_wall_timer(
      500ms, std::bind(&MinimalPublisher::timer_callback, this));
    }
```

- Rclcpp::Node를 상속받아 만든 MinimalPublisher 클래스 생성한다.

- 생성자는 노드 이름을 "minimal_publisher"로 설정하고 count를 0으로 초기화한다.

- Publisher_는 문자열 메시지 유형, 토픽 이름(topic)과 백업시 큐의 크기 제한을 초기화한다.

- timer_는 초당 2번씩 초기화 되어 timer_callback 함수를 실행함

```
  private:
    void timer_callback()
    {
      auto message = std_msgs::msg::String();
      message.data = "Hello, world! " + std::to_string(count_++);
      RCLCPP_INFO(this->get_logger(), "Publishing: '%s'", message.data.c_str());
      publisher_->publish(message);
    }
    rclcpp::TimerBase::SharedPtr timer_;
    rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
    size_t count_;
};
```

- Timer_callback 함수는 메시지 데이터설정과 저장을 한 후 RCLCPP_INFO 매크로를 실행해 콘솔에 메시지를 출력한다.

- Publisher_를 통해 메시지 내용을 담아 송신한다.

- 그 아래로 timer_와 Publisher_, count_를 생성한다.

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

Int main에서는 다음 명령어들을 실행한다.

- Ros2를 초기화하는 rclcpp::init

- timer_callback 함수를 포함한 노드 데이터를 처리하는 rclcpp::spin

- 노드를 제거하는 rclcpp::shutdown

코드설명(Subscriber)

```
#include <memory>

#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"
using std::placeholders::_1;
```

Publisher 와 비슷하게 헤더파일 선언을 한다.

```
class MinimalSubscriber : public rclcpp::Node
{
  public:
    MinimalSubscriber()
    : Node("minimal_subscriber")
    {
      subscription_ = this->create_subscription<std_msgs::msg::String>(
      "topic", 10, std::bind(&MinimalSubscriber::topic_callback, this, _1));
    }
```

- Rclcpp::Node를 상속받아 만든 MinimalSubscriber 클래스 생성한다.

- 생성자는 노드 이름을 "minimal_subscriber"로 설정한다.

- Subscription_은 문자열 메시지 유형, 토픽 이름(topic)과 백업시 큐의 크기 제한을 초기화하고 문자가 수신될 때 마다 topic_callback이라는 함수를 실행한다.

```
private:
  void topic_callback(const std_msgs::msg::String & msg) const
  {
    RCLCPP_INFO(this->get_logger(), "I heard: '%s'", msg.data.c_str());
  }
  rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription_;
```

- topic_callback은 Publisher의 timer_callback과 유사하지만 publisher_와 같이 송신하는 코드가 없이 터미널에 받은 메시지 내용만 띄운다.

-그 아래로 subscription_을 선언한다.

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

Int main에서는 다음 명령어들을 실행한다.

- Ros2를 초기화하는 rclcpp::init

- topic_callback함수를 포함한 노드 데이터를 처리하는 rclcpp::spin

- 노드를 제거하는 rclcpp::shutdown

코드설명(Server)

```
int64 a
int64 b
---
int64 sum
```

.srv 폴더에 저장되어있는 파일의 내용이다.

- int64 a, b는 요청하는 파라미터이다.

-int64 sum은 응답 받을 파라미터이다.

```
#include "rclcpp/rclcpp.hpp"
#include "example_interfaces/srv/add_two_ints.hpp"

#include <memory>
```

ROS2 시스템에 일반적인 부분을 사용할 수 있는 rclcpp/rclcpp.hpp 헤더파일과 파라미터를 설정하는 .srv 의 파일을 불러오고 다른 헤더파일도 불러온다.

```
void add(const std::shared_ptr<example_interfaces::srv::AddTwoInts::Request> request,
         std::shared_ptr<example_interfaces::srv::AddTwoInts::Response>      response)
{
    response->sum = request->a + request->b;
    RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Incoming request\na: %ld" " b: %ld",
        request->a, request->b);
    RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "sending back response: [%ld]", (long int)response->sum);
}
```

Add 함수는 요청 받은 2 개의 정수를 더하고 합을 응답으로 보내면서 동시에 요청받은 a, b 의 값, 응답할 값을 로그에 띄우는 역할을 한다.

```
int main(int argc, char **argv)
{
  rclcpp::init(argc, argv);

  std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("add_two_ints_server");

  rclcpp::Service<example_interfaces::srv::AddTwoInts>::SharedPtr service =
    node->create_service<example_interfaces::srv::AddTwoInts>("add_two_ints", &add);

  RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Ready to add two ints.");

  rclcpp::spin(node);
  rclcpp::shutdown();
}
```

Int main에서는 다음 명령어들을 실행한다.

- Ros2를 초기화하는 rclcpp::init

- "add_two_ints_server"라는 노드를 생성

- add_two_ints라는 서비스를 생성하고 &add를 통해 add함수를 실행

-RCLCPP_INFO를 통해 로그를 띄움

-spin을 통해 다시 서비스를 준비함

-종료시 서비스를 소멸함

코드설명(Client)

```
#include "rclcpp/rclcpp.hpp"
#include "example_interfaces/srv/add_two_ints.hpp"

#include <chrono>
#include <cstdlib>
#include <memory>

using namespace std::chrono_literals;
```

Server와 유사하게 헤더파일 설정

```
int main(int argc, char **argv)
{
  rclcpp::init(argc, argv);

  if (argc != 3) {
      RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "usage: add_two_ints_client X Y");
      return 1;
  }
```

-ROS2 초기화

- 입력받은 데이터의 요소가 3개가 아니면 다음 문자를 출력

```
std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("add_two_ints_client");
rclcpp::Client<example_interfaces::srv::AddTwoInts>::SharedPtr client =
  node->create_client<example_interfaces::srv::AddTwoInts>("add_two_ints");
```

- Server와 유사하게 "add_two_ints_client"라는 노드를 생성한다.

- add_two_ints라는 서비스를 생성한다.

```
auto request = std::make_shared<example_interfaces::srv::AddTwoInts::Request>();
request->a = atoll(argv[1]);
request->b = atoll(argv[2]);
```

Server에 요청할 데이터는 .srv파일에 있는 내용을 기반으로 생성된다.

```
while (!client->wait_for_service(1s)) {
  if (!rclcpp::ok()) {
    RCLCPP_ERROR(rclcpp::get_logger("rclcpp"), "Interrupted while waiting for the service. Exiting.");
    return 0;
  }
  RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "service not available, waiting again...");
}
```

- while 루프문은 1초동안 클라이언트가 서비스 노드를 탐색할 시간을 준다.

- 노드를 찾았다면 로그에 해당 문구 띄운 후 반복문을 나간다.

- 아직 못 찾았다면 문구를 띄운 후 계속해서 기다린다.

```
  auto result = client->async_send_request(request);
  // Wait for the result.
  if (rclcpp::spin_until_future_complete(node, result) ==
    rclcpp::FutureReturnCode::SUCCESS)
  {
    RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Sum: %ld", result.get()->sum);
  } else {
    RCLCPP_ERROR(rclcpp::get_logger("rclcpp"), "Failed to call service add_two_ints");
  }

  rclcpp::shutdown();
  return 0;
}
```

- Server에서 응답을 받는다.

- 만일 응답이 제대로 왔다면 응답받은 값을 로그에 띄운다.

- 아니면 실패했다는 로그를 띄운다.

- 종료한다면 해당 노드를 제거한다.

공통(CmakeLists.txt, package.xml)

CmakeList.txt는 각 파일들이 사용하는 헤더파일(의존성)을 설정한다.

```
cmake_minimum_required(VERSION 3.5)
project(cpp_srvcli)

find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(example_interfaces REQUIRED)

add_executable(server src/add_two_ints_server.cpp)
ament_target_dependencies(server rclcpp example_interfaces)

add_executable(client src/add_two_ints_client.cpp)
ament_target_dependencies(client rclcpp example_interfaces)

install(TARGETS
  server
  client
  DESTINATION lib/${PROJECT_NAME})

ament_package()
```

서비스 camke예시

- Find_pakage를 통해 사용할 의존성을 찾는다.

- add_executable을 통해 어떤 파일을 실행할지 설정한다.

- ament_target_dependencies를 통해 해당 파일에 어떤 의존성을 사용할지 정한다.

- install(TARGETS)를 이용해 실행할 파일을 최종적으로 설정한다.

Package.xml도 cmakelists.txt와 유사한 역할을 한다.

```
<description>Examples of minimal publisher/subscriber using rclcpp</description>
<maintainer email="you@email.com">Your Name</maintainer>
<license>Apache License 2.0</license>
```

먼저 description, maintainer, license를 자신에게 맞게 설정한다.

```
<depend>rclcpp</depend>
<depend>std_msgs</depend>
```

Include문에 사용하는 종속성들을 다음 형식으로 입력한다.

이와 같이 설정을 하면 제대로 각 파일이 잘 작동할 것이다.