

Robot Operating System 2 를 활용한
자율 주행 로봇 프로그래밍

엣지아이랩

목차

목차.....	3
1. Robot Operating System 개요.....	5
1.1. Robot Operating System 이란 무엇인가?.....	5
1.2. Robot Operating System 의 역사.....	6
1.3. ROS1 과 ROS2 의 차이점.....	7
1.4. ROS2 와 DDS(Data Distribution Service).....	16
2. ROS2 개발환경 설정.....	25
2.1. 리눅스의 사용환경과 구조.....	26
2.2. Ubuntu Linux 설치.....	29
2.3. ROS Dashing 설치.....	37
2.4. ROS2 통합 개발환경(IDE) 구축.....	39
3. 패키지의 설치와 노드 실행.....	43
3.1. Turtlesim 소개.....	43
3.2. Turtlesim 패키지와 노드.....	44
3.3. Turtlesim 패키지의 노드 실행.....	45
3.4. 노드 토픽, 서비스, 액션의 조회.....	46
3.5. RQT_GRAPH 로 보는 노드와 토픽의 그래프 뷰.....	48
4. ROS2 노드와 데이터 통신.....	49
4.1. 노드(Node)와 메시지 통신(Message Communication)....	49
4.2. 메시지 타입.....	50
4.3. 노드의 실행.....	53
4.4. 노드 목록.....	54
4.5. 노드 정보 확인하기.....	55
5. ROS2 Message Type.....	57
5.1. 토픽(Topic).....	57
5.2. 서비스(Service).....	66
5.3. 액션(Action).....	74
5.4. 파라미터.....	81
6. ROS2 기본 프로그래밍.....	87
6.1. ROS 프로그래밍 규칙.....	87
6.2. ROS2 패키지 생성 및 빌드.....	93
6.3. Simple Publisher / Subscriber 작성하기(C++).....	97
6.4. Simple Publisher / Subscriber 작성하기(Python).....	108
6.5. ROS2 인터페이스 구현하기.....	118
6.6. ROS2 패키지 설계.....	125
6.7. ROS2 Python 프로그래밍.....	140
6.8. ROS2 C++ 프로그래밍.....	160
7. ROS2 기반 주행로봇 구동하기.....	187
7.1. Turtlebot3 개발환경 구축.....	187
7.2. Turtlebot3 제어 Node 실행 및 기본 조작하기.....	197
7.3. Turtlebot3 를 이용한 SLAM & Navigation.....	201
7.4. 가상환경에서의 Turtlebot3 제어.....	214

8.	ROS2 기반 주행로봇 프로그래밍.....	220
8.1.	Teleoperation 구현하기.....	220
8.2.	배터리 정보 Subscribe 하기.....	238
8.3.	장애물 인식하기.....	243

1. Robot Operating System 개요

1.1. Robot Operating System 이란 무엇인가?

Robot Operating System(ROS)는 로봇 운영체제로 로봇을 구동하기 위한 프레임워크 중 하나이다. 다양한 로봇 플랫폼에 걸쳐 복잡하고 강력한 로봇의 동작을 만드는 작업을 단순화 하는 것을 목표로 하는 다양한 Tool과 라이브러리 등의 모음이다.

강력하고 범용적인 로봇 소프트웨어를 만드는 것은 아주 어려운 일이다. 우리 인간에게 사소한 것으로 보이는 문제들이 로봇의 관점에서 보면 동일한 문제라도 작업환경에 따라 많은 제약사항이 따르기 때문이다. 이러한 제약사항을 다루는 것은 매우 까다로운 작업이기 때문에 개인 및 소규모 연구실에서는 이러한 제약사항을 해결하기 매우 어려운 것이 현실이다.

그 결과 ROS는 협력적인 로봇 소프트웨어 개발을 장려하는 것을 목표로 구축이 시작되었다. 예를 들어 어느 한곳의 실험실은 실내 환경 지도제작에 관한 전문가가 있을 수 있고, 지도제작에 세계적인 수준의 시스템을 제공할 수 있다. 다른 실험실은 지도를 사용한 Navigation 전문가가 있을 수 있고, 또 다른 곳에서는 여러 물체 사이에서 우리가 원하는 하나의 물체를 인식하는 컴퓨터 비전 접근법을 발견했을 수 있다. ROS는 이 세개의 그룹이 서로 협업하고 서로의 작업을 기반으로 새로운 Tool을 개발할 수 있도록 하기 위해 설계된 것이다.

그림 1 ROS의 구성요소



1.2. Robot Operating System 의 역사

2000년대 중반 스탠포드대학에서는 내장형 AI를 포함하고 있는 Stanford AI 로봇 (STAIR)과 Personal Robots(PR)프로그램과 같은 로봇을 제작하고 활용하기 위한 프로 토타입을 완성하게 되었다. 2007년, 로봇 인큐베이터인 Willow Garage는 이러한 개념을 훨씬 더 확장하고 잘 테스트된 구현을 만들수 있는 중요한 자원을 제공하며 ROS의 개념이 다져지게 되었다. 이러한 노력은 ROS의 핵심 아이디어와 기본적인 소프트 웨어 패키지에 시간과 전문 지식을 기여한 수많은 연구자들에 의해 증대되었다. 전체 적으로 이 소프트웨어는 관대한 BSD 오픈소스 라이선스를 이용하여 공개적으로 개발되었으며, 점차 로봇 연구 커뮤니티에서 널리 사용되는 플랫폼이 되었다.

ROS는 Willow Garage에서 제공한 PR2 로봇을 받은 많은 기관들을 포함하여 여러 기관과 여러



그림 2 Willow Garage에서 탄생한 최초의 ROS Platform PR2

로봇을 위해 개발되었다. 비록 모든 개발자들이 코드를 같은 서버에 배 치하는 것이 훨씬 더 간단했겠지만, 수년동안 “집단그룹”모델은 ROS 생태계의 가장 큰 장점중 하나로 떠올랐다. 어떤 그룹이든 자신의 서버에서 자신의 ROS 코드 저장 소를 시작할 수 있으며, 그들은 본인들이 만든 소스코드에 대한 완전한 소유권과 통제권을 가지게 될 수 있었다. 만약 그들이 그들의 저장소를 공개적으로 이용할 수 있게 하는 것을 선택한다면, 그들은 그들의 업적에 대해 그들이 받을만한 인정과 신용을 받을 수 있고, 모든 오픈소스 소프트웨어 프로젝트처럼 특정한 기술적 피드백과 개선으로부터 이익을 얻을 수 있는 것이다.

1.3. ROS1 과 ROS2 의 차이점

2017년 12월 8일에 OpenRobotics 의해 공개된 ROS 2는 기존의 ROS와 호환되지 않는 새로운 ROS 시스템이다. 여기에서는 ROS 2가 개발된 경위에 대해 설명한 후 기존의 ROS(이하 ROS 1으로 표기)와 ROS 2의 기능 차이를 설명하고 새롭게 추가 된 기능에 대해 알아보자.

1.3.1. ROS1

ROS 1은 2007년에 개발이 시작되어 지금은 대학, 연구 기관, 산업계, 로봇 자작의 취미 활동까지 폭넓게 이용되고 있다. 원래 ROS 1은 Willow Garage[2], [3]사가 개인 서비스 로봇인 PR2[4] 개발에 필요한 미들웨어 형태의 로봇 개발 프레임워크를 다양한 개발 툴과 함께 오픈 소스로 공개한 것으로 시작하였다. 따라서 개발 환경으로는 PR2의 초기 컨셉을 그대로 이어받아 다음과 같은 제한 사항이 있었다.

- 단일 로봇
- 워크스테이션급 컴퓨터
- Linux 환경
- 실시간 제어 지원하지 않음
- 안정된 네트워크 환경이 요구됨
- 주로 대학이나 연구소와 같은 아카데믹 연구 용도

위의 개발 환경은 PR2 개발 당시 설정된 것이며, 오늘날 요구되는 로봇 개발 환경과는 큰 차이가 있다. 예를 들어, 최근의 ROS 1은 종래 가장 많이 이용되고 있던 학술분야뿐만 아니라, 제조 로봇, 농업 로봇, 드론, 소셜 로봇과 같은 상용 로봇 등으로 이용되고 있다. 극단적인 예로 NASA가 국제 우주 정거장에서 사용한 Robonaut에는 ROS 1가 채용되고 있지만, 거기에서는 실시간 제어가 요구되었고 이를 위해서 ROS 1을 수정하여 사용하였다. 이러한 새로운 로봇 개발 환경 및 요구되는 기능을 정리하면 다음과 같다.

- 복수대의 로봇
- 임베디드 시스템에서의 ROS 사용
- 실시간 제어
- 불안정한 네트워크 환경에서도 동작 할 수 있는 유연함
- 멀티 플랫폼 (Linux, macOS, Windows)
- 최신 기술 지원 (Zeroconf, Protocol Buffers, ZeroMQ, WebSockets, DDS 등)
- 상업용 제품 지원

ROS 1에서 이러한 새롭게 요구되는 기능을 제공하려면 대규모 API의 변경이 필요하다. 그러나 기존의 ROS 1과의 호환성을 유지하면서 수 많은 새로운 기능을 추가하는 것은 쉽지 않았다. 또한, 기존의 ROS 1을 문제 없이 이용하고 있는 사용자에게는 큰 API의 변경은 바람직하지 않다.

그래서 ROS의 차세대 기능을 도입한 버전을 ROS 2라고, ROS 1에서 분리하여 개발하게 된 것이다. 기존의 ROS1 사용자는 필요하다면 그대로 ROS 1을 이용할 수 있다. 한편, 새로운 기능이 필요한 사용자는 ROS 2를 선택하면 된다. 또한, ROS 1과 ROS 2 사이에서 서로 메시지 통신이 가능한 브리지 프로그램 (ros1_bridge) 제공되므로 두 버전 모두를 함께 사용하는 것도 가능하다.

Features	ROS 1	ROS 2
Platforms	Linux, macOS	Linux, macOS, Windows
Real-time	external frameworks like OROCOS	real-time nodes when using a proper RTOS with carefully written user code
Security	SROS	SROS 2, DDS-Security, Robotic Systems Threat Model
Communication	XMLRPC + TCPROS	DDS (RTPS)
Middleware interface	-	rmw
Node manager (discovery)	ROS Master	No, use DDS's dynamic discovery
Languages	C++03, Python 2.7	C++14 (C++17), Python 3.5+
Client library	roscpp, rospy, rosjava, rosnodejs, and more	rclcpp, rclpy, rcljava, rcljs, and more
Build system	rosbuild -- catkin (CMake)	ament (CMake), Python setuptools (Full support)
Build tools	catkin_make, catkin_tools	colcon
Build options	-	Multiple workspace, No non-isolated build, No devel space
Version control system	rosws -- wstool, rosinstall (*.rosinstall)	vcstool (*.repos)
Life cycle	-	node life cycle
Multiple nodes	one node in a process	multiple nodes in a process
Threading model	single-threaded execution or multi-threaded execution	custom executors
Messages (topic, service, action)	*.msg, *.srv, *.action	*.msg, *.srv, *.action, *.idl
Command Line Interface	rosrun, roslaunch, ros topic ...	ros2 run, ros2 launch, ros2 topic ...
roslaunch	XML	Python, XML, YAML
Graph API	remapping at startup time only	remapping at runtime
Embedded Systems	rosserial, mROS	microROS, XEL Network, ros2arduino, Renesas DDS-XRCE(Micro-XRCE-DDS), AWS ARCLM

그림 3 ROS2 와 ROS1 의 차이점

1.3.2. ROS2 의 특징

Platforms

ROS 2 부터는 3 대 운영체제인 Linux, Windows, macOS 를 모두 지원한다. 이는 바이너리 파일로 설치가 가능하다는 의미로 Windows 사용자가 많은 한국의 경우에는 반가운 소식으로 받아들이는 분들이 많을 것 같다. ROS 2 Foxy Fitzroy 기준으로 보았을 때 Linux 는 Ubuntu Focal (20.04), Windows 는 Windows 10 버전, macOS 는 Mojave (10.14)버전을 지원하고 있다. Linux 의 경우에는 Linux 배포판중 일반 사용자가 가장 많은 Canonical 의 Ubuntu 진영에서 ROS 2 TSC 로 가입되어 있어서 관련된 내용을 많이 볼 수 있다. 그리고 Linux, macOS 는 ROS 1 부터 지원하고 있었는데 이번 ROS 2 에서는 Microsoft 가 ROS 2 TSC 로 들어오고 Windows 용 패키지 및 테스트, Visual Studio Code Extension for ROS 까지 준비하는 등 굉장한 노력을 기울여 Windows 사용자들도 ROS 2 를 쉽게 사용할 수 있게 되었다. Windows 사용자라면 Windows 관련 문서를 참고하기를 추천한다.

Real-time

ROS 2는 Real-time 을 지원한다. 단, 선별된 하드웨어 사용, 리얼타임 운영체제 사용, DDS의 RTPS(Real-time Publish-Subscribe Protocol)와 같은 통신 프로토콜을 사용, 매우 잘 짜여진 리얼타임 코드 사용을 전제로 실시간성을 지원하고 있다.

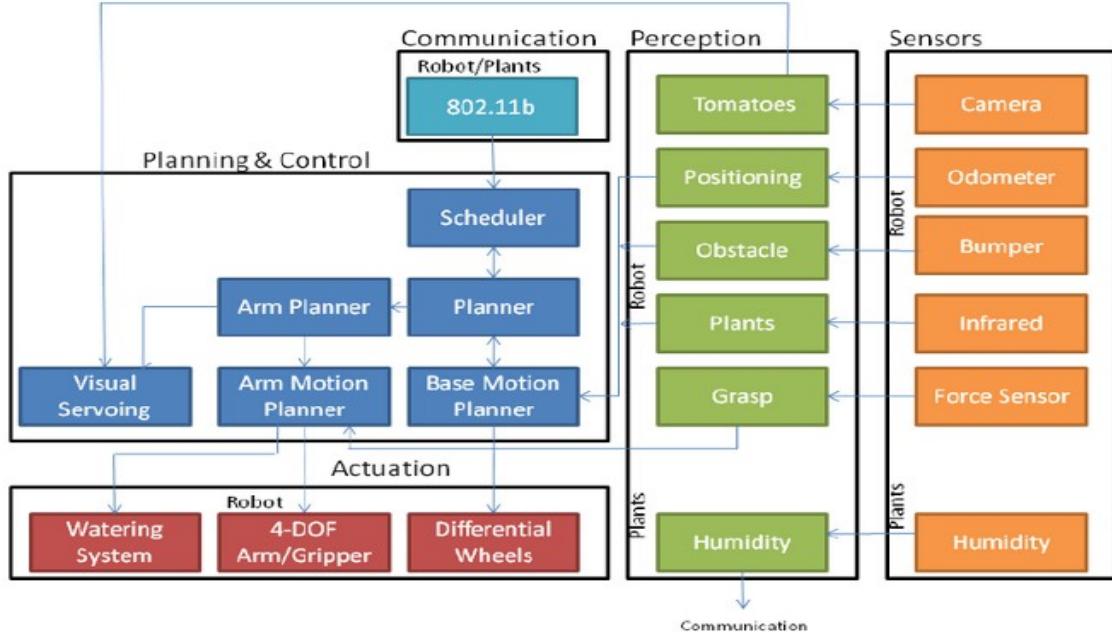


그림 4 ROS 의 Realtime Architechture

Security

ROS 1에서는 항상 보안이 문제였다. 노드를 관리하는 ROS master의 하나의 IP와 포트만 노출되면 모든 시스템을 죽일 수 있었으며, 보안 입장에서는 TCPROS는 뻥 뚫린 큰 구멍에 가까웠다. 하지만 ROS 1은 이러한 부족한 부분을 매우 기름개발에 사용되는 다양한 하드웨어와 소프트웨어를 평가하고 작동하는 유연성에 더 무게를 두었고 이러한 유연성은 보안에 대한 기회비용보다 더 중요하게 여겼다. 즉, 보안 이슈는 개발 우선 순위에서 뒤져 있었다. 당연히 ROS 초창기에는 이 선택은 옳았다.

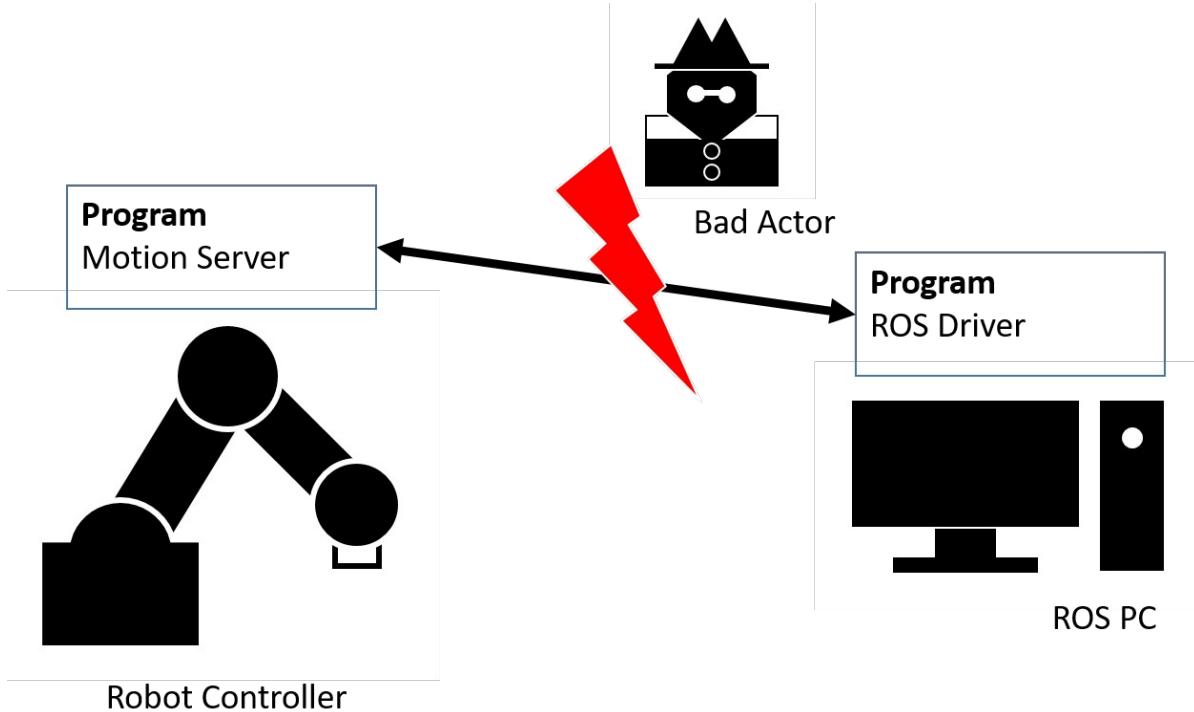


그림 5 ROS 의 보안 Issue

하지만 시간이 흘러 이러한 취약점은 상용 로봇에 ROS를 도입할 수 없게 만드는 첫번째 이유이자 가장 큰 걸림돌이 되었다. 이에 ROS 2에서는 디자인 설계부터 이 부분을 명확히 짚고 넘어갔다. 우선, TCP 기반의 통신은 OMG(Object Management Group)에서 산업용으로 사용 중인 DDS(Data Distribution Service)를 도입하였고, 자연스럽게 DDS-Security이라는 DDS 보안 사양을 ROS에 적용하여 보안에 대한 이슈를 통신단부터 해결하였다. 또한 ROS 커뮤니티에서는 SROS 2(Secure Robot Operating System 2)라는 툴을 개발하였고 보안 관련 RCL 서포트 및 보안관련 프로그래밍에 익숙지 않은 로보틱스 개발자를 위해 보안을 위한 툴킷을 만들어 배포하고 있다. 이 부분에 대한 더 자세한 내용은 ROS 2 디자인 문서의 `Security` 관련 문서를 참고하길 바라며 지난 ROSCon2019에서의 워크샵에서 진행한 `Is your robot secure? ROS 1 & ROS 2 Security Workshop`의 문서도 참고하면 도움이 될 것 같다.

Communication

ROS 1에서는 자체 개발한 TCPROS와 같은 통신 라이브러리를 사용하고 있던 반면, ROS 2은 리얼타임 퍼블리시와 서브스크라이브 프로토콜인 RTPS(Real Time Publish Subscribe)를 지원하는 통신 미들웨어 DDS를 사용하고 있다. DDS는 OMG(Object Management Group)에 의해 표준화가 진행되고 있으며, 상업적인 용도에도 적합하다는 평가가 지배적이다. DDS에서는 IDL(Interface Description Language)를 사용하여 메시지 정의 및 직렬화를 더 쉽게, 더 포괄적으로 다룰 수 있다. 또한 통신 프로토콜로는 RTPS을 채용하여 실시간 데이터 전송을 보장하고 임베디드 시스템에도 사용할 수 있다. DDS는 노드 간의 자동 감지 기능을 지원하고 있어서 기존 ROS 1에서 각 노드들의 정보를 관리하였던 ROS 마스터가 없어도 여러 DDS 프로그램간에 통신 할 수 있다. 또한 노드 간의 통신을 조정하는 QoS(Quality of Service) 매개 변수를 설정할 수 있어서 TCP처럼 데이터 손실을 방지함으로써 신뢰도를 높이거나, UDP처럼 통신 속도를 최우선하여 사용할 수도 있다. 이러한 다양한

기능을 갖춘 DDS를 이용하여 ROS1의 퍼블리시, 서브스크라이브 형 메시지 전달은 물론, 실시간 데이터 전송, 불안정한 네트워크에 대한 대응, 보안 강화 등이 강화되었다. DDS의 채용은 ROS1에서 ROS2로 바뀌면서 가장 큰 변화점이다.

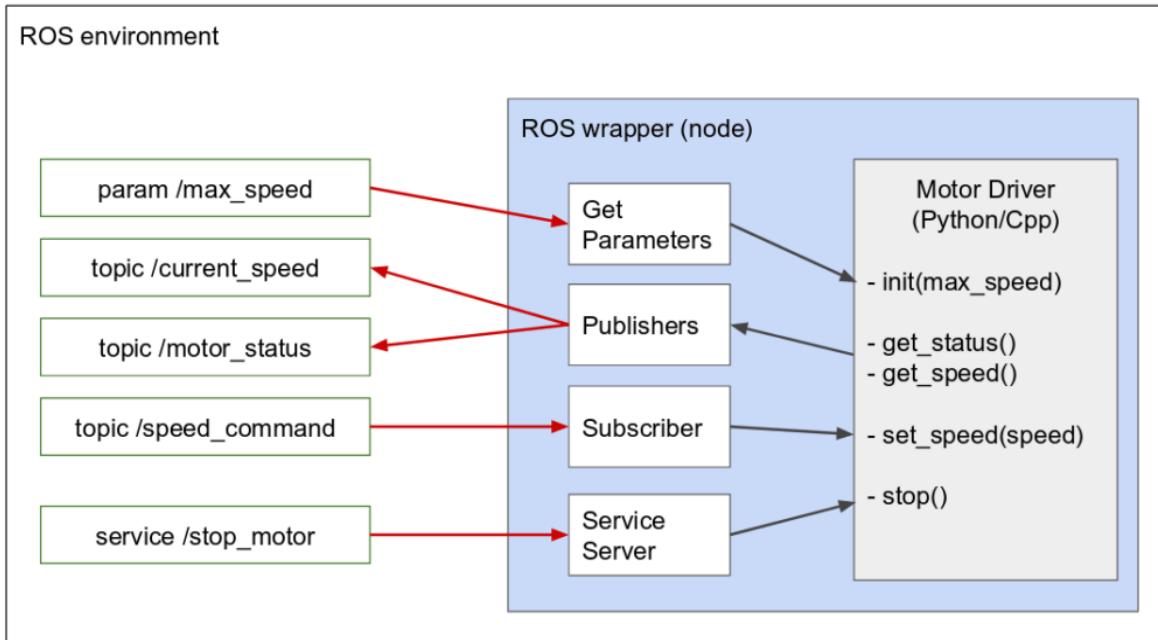


그림 6 An engine adapter node architecture

Middleware interface

앞서 설명한 DDS는 다양한 기업에서 통신 미들웨어 형태로 제공하고 있다. 그 벤더로는 아래의 리스트와 같이 10 곳이 있는데 이 중 ROS 2를 지원하는 업체는 ADLink, Eclipse Foundation, Eprosima, Gurum Network, RTI로 총 5 곳이다. DDS 제품명으로는 ADLINK의 OpenSplice, Eclipse Foundation의 Cyclone DDS, Eprosima의 Fast DDS, Gurum Network의 Gurum DDS, RTI의 Connext DDS가 있다. 참고로 이 중 Gurum Network는 유일하게 대한민국 기업으로 DDS를 순수 국산 기술로 개발하여 상용화에 성공한 기업이다.



그림 7 Gurum DDS Architecture

ROS 2에서는 이러한 벤더들의 미들웨어를 유저가 원하는 사용 목적에 맞게 선택하여 사용할 수 있도록 ROS Middleware(RMW, [38])형태로 지원하고 있다. 이는 각 벤더들의 미들웨어마다 API 가 약간씩

달라도 ROS 2 유저들은 이를 생각하지 않고 통일된 코드로 쉽게 바꿔서 사용할 수 있도록 것으로, RMW 는 여러 DDS 구현을 지원하기 위하여 API의 추상화 인터페이스로 지원하고 있다.

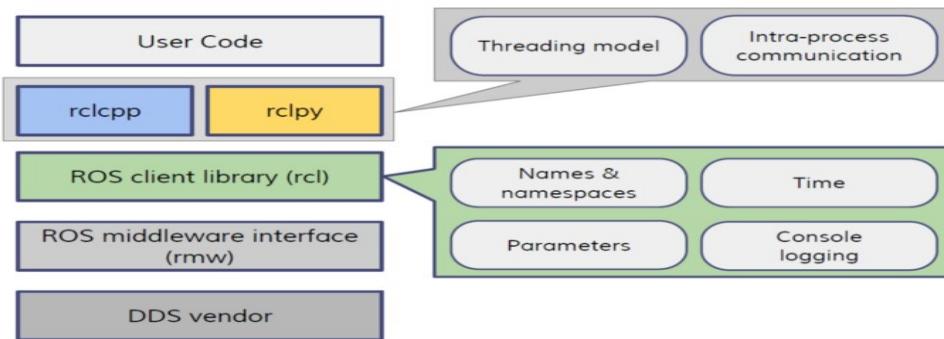


그림 8 RCL 추상화 Library Architecture

Node Manager

ROS 1에서의 필수 실행 프로그램으로는 roscore 가 있다. 이를 실행시키면 ROS Master, ROS Parameter Server, rosout logging node 가 실행 되었다. 특히 ROS Master는 ROS 시스템의 노드들의 이름 지정 및 등록 서비스를 제공하였고, 각 노드에서 퍼블리시 또는 서브스크라이브하는 메시지를 찾아서 연결할 수 있도록 정보를 제공해 주었다. 즉, 각각 독립되어 실행되는 노드들의 정보를 관리하여 서로 연결해야 하는 노드들에게 상대방 노드의 정보를 건네주어 연결할 수 있게 해 주는 매우 중요한 중매 역할을 수행했었다. 이 때문에 ROS 1에서는 노드 사이의 연결을 위해 네임 서비스를 마스터에서 실행했어야 했고, 이 ROS Master가 연결이 끊기거나 죽는 경우 모든 시스템이 마비되는 단점이 있었다.

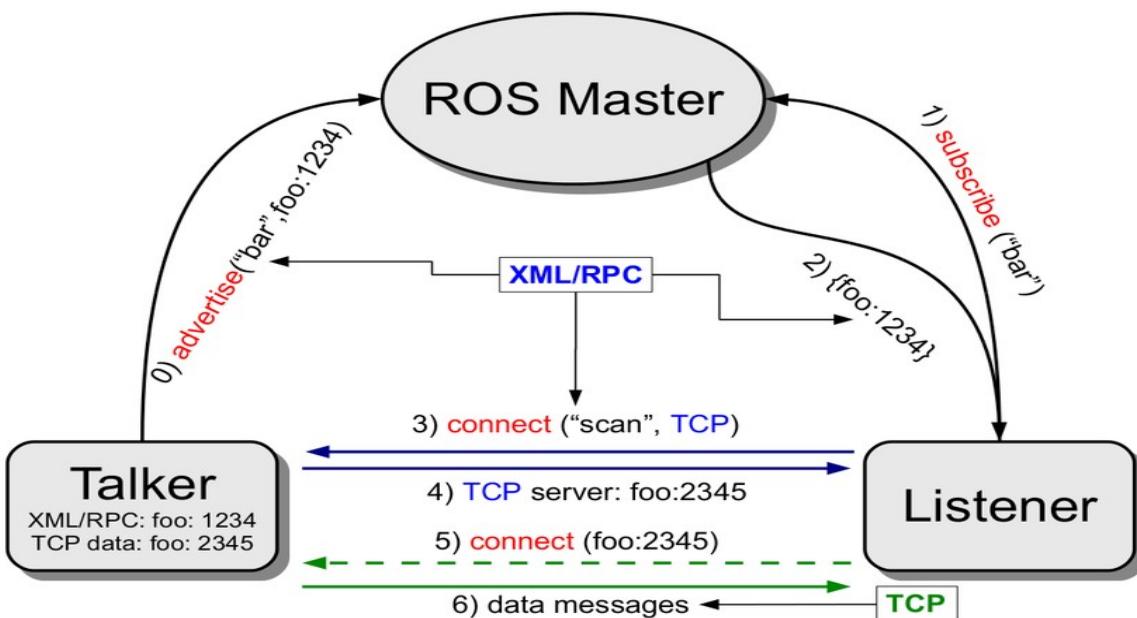


그림 9 ROS1.0ster Client 구조

ROS 2에서는 roscore 가 없어지고 3 가지 프로그램이 각각 독립 수행으로 바뀌었다. 특히, ROS Master의 경우 완전히 삭제되었는데 이는 DDS를 사용함에 따라 노드를 DDS의 Participant

개념으로 취급하게 되었으며, Dynamic Discovery 기능을 이용하여 DDS 미들웨어를 통해 직접 검색하여 노드를 연결 할 수 있게 되었다.

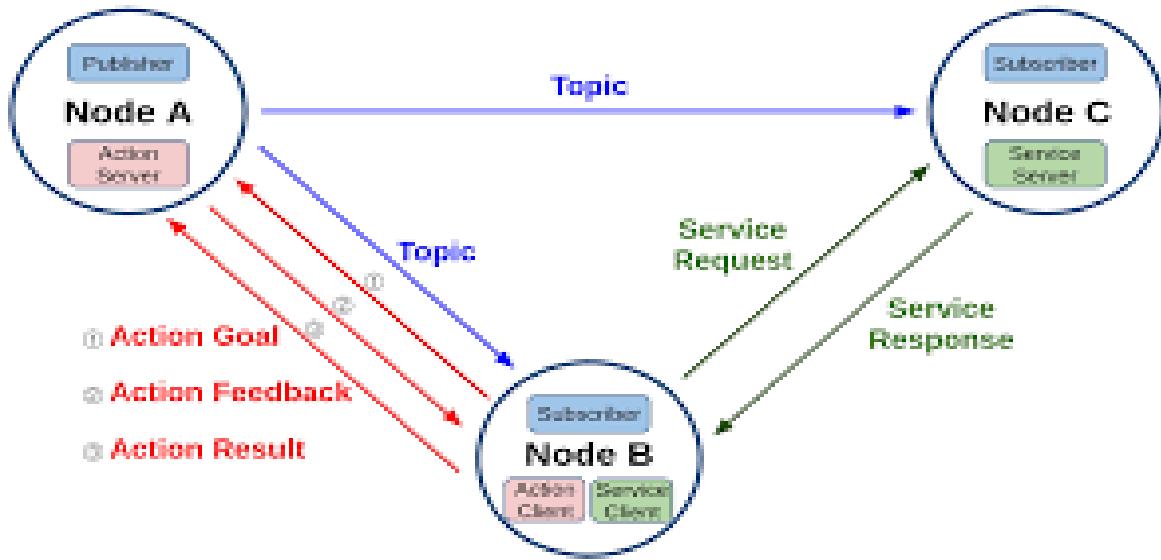


그림 10 ROS2 의 Node 간 통신 구조

Build Tools

ROS 1의 경우 여러 가지 다른 도구, 즉 catkin_make, catkin_make_isolated 및 catkin_tools가 지원되었다. ROS 2에서는 알파, 베타, 그리고 Ardent 릴리스까지 빌드 도구로 ament_tools가 이용되었고 지금에 와서는 colcon을 추천하고 있다. colcon은 ROS 2 패키지를 작성, 테스트, 빌드 등 ROS 2 기반의 프로그램할 때 빼놓을 수 없는 툴로 작업흐름을 향상시키는 CLI 타입의 명령어 도구이다. 사용 방법은 `colcon test`와 `colcon build`와 같이 터미널창에서 수행하게 되며 다양한 옵션을 사용할 수 있다.

ROS 2에서는 빌드 관련 내용들이 모두 변경되면서 빌드 옵션에도 새로운 변화가 생겼다. 그중 사용하면서 가장 좋았던 3 가지를 꼽자면 아래와 같다.

우선 `Multiple workspace` 이다. 이는 ROS 1에서는 `catkin_ws`와 같이 특정 워크스페이스를 확보하고 하나의 워크스페이스에서 모든 작업을 다 했는데 ROS 2에서는 복수의 독립된 워크스페이스를 사용할 수 있어서 작업 목적 및 패키지 종류별로 관리할 수 있게 되었다.

둘째는 `No non-isolated build` 이다. ROS 1에서는 하나의 CMake 파일로 여러 개의 패키지를 동시에 빌드 할 수 있었다. 이렇게 하면 빌드 속도가 빨라지지만 모든 패키지의 종속성에 신경을 많이 써야 하고 빌드 순서가 매우 중요하게 된다. 또한 모든 패키지가 동일 네임스페이스 사용하게 되므로 이름에서 충돌이 발생할 수 있었다. ROS 2에서는 이전 빌드 시스템인 catkin에서 일부 기능으로 사용되었던 `catkin_make_isolated` 형태와 같은 격리 빌드만을 지원함으로써 모든 패키지를 별도로 빌드하게 되었다. 이 기능 변화를 통해 설치용 폴더를 분리하거나 병합할 수 있게 되었다.

셋째는 `No devel space`이다. catkin은 패키지를 빌드 한 후 devel이라는 폴더에 코드를 저장한다. 이 폴더는 패키지를 설치할 필요 없이 패키지를 사용할 수 있는 환경을 제공한다. 이를 통해

파일 복사를 피하면서 사용자는 파이썬 코드를 편집하고 즉시 코드 실행할 수 있었다. 단 이러한 기능은 매우 편리한 기능이지만 패키지를 관리하는 측면에서 복잡성을 크게 증가시켰다. 이에 ROS 2에서는 패키지를 빌드 한 후 설치해야 패키지를 사용할 수 있도록 바뀌었다. 단 쉬운 사용성도 고려하여 colcon 사용 시에 `colcon build --symlink-install` 와 같은 옵션을 사용하여 심벌릭 링크 설치의 선택적 기능을 사용하여 동일한 이점을 제공하고 있다.

1.4. ROS2 와 DDS(Data Distribution Service)

본격적인 ROS 2 프로그래밍 강좌에 앞서서 로봇 운영체제 ROS에서 중요시 여기는 몇 가지 용어 정의 및 메시지, 메시지 통신에 대해 먼저 알아보도록 하자. 특히, 메시지 통신은 ROS 프로그래밍에 있어서 ROS 1과 2의 공통된 중요한 핵심 개념이기에 ROS 프로그래밍에 들어가기 전에 꼭 이해하고 넘어가야 할 부분이다.

ROS에서는 프로그램의 재사용성을 극대화하기 위하여 최소 단위의 실행 가능한 프로세서라고 정의하는 노드(node) 단위의 프로그램을 작성하게 된다. 이는 하나의 실행 가능한 프로그램으로 생각하면 된다. 그리고 하나 이상의 노드 또는 노드 실행을 위한 정보 등을 묶어 놓은 것을 패키지(package)라고 하며, 패키지의 묶음을 메타패키지 metapackage)라 하여 따로 분리한다.

여기서 제일 중요한 것은 실제 실행 프로그램인 노드인데 앞서 이야기한 것과 마찬가지로 ROS에서는 최소한의 실행 단위로 프로그램을 나누어 프로그래밍하기 때문에 노드는 각각 별개의 프로그램이라고 이해하면 된다. 이에 수많은 노드들이 연동되는 ROS 시스템을 위해서는 노드와 노드 사이에 입력과 출력 데이터를 서로 주고받게 설계해야만 한다. 여기서 주고받는 데이터를 ROS에서는 메시지(message)라고 하고 주고받는 방식을 메시지 통신이라고 한다. 여기서 데이터에 해당되는 메시지(message)는 integer, floating point, boolean, string 와 같은 변수 형태이며 메시지를 품고 있는 간단한 데이터 구조 및 메시지들의 배열과 같은 구조도 사용할 수 있다. 그리고 메시지를 주고받는 통신 방법에 따라 토픽(topic), 서비스(service), 액션(action), 파라미터(parameter)로 구분된다.

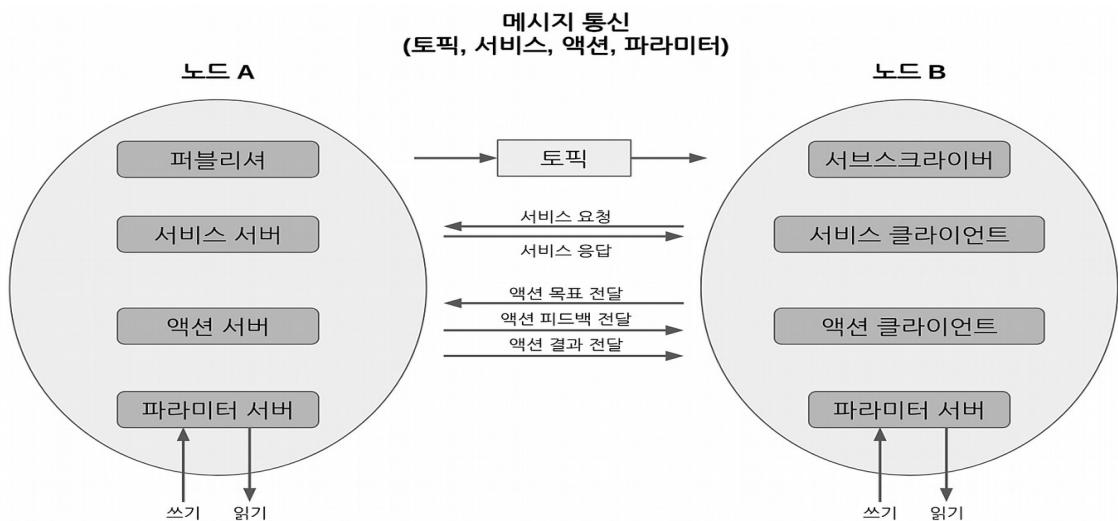


그림 11 ROS2 의 메시지 통신

ROS에서 사용되는 메시지 통신 방법으로는 토픽(topic), 서비스(service), 액션(action), 파라미터(parameter)가 있다. 각 메시지 통신 방법의 목적과 사용 방법은 다르기는 하지만 토픽의 발간(publish)과 구독(subscribe)의 개념을 응용하고 있다. 이 데이터를 보내고 받는 발간, 구독 개념은 ROS 1은 물론 ROS 2에서도 매우 중요한 개념으로 변함이 없는데 이 기술에 사용된 통신 라이브러리는 ROS 1,

2에서 조금씩 다르다. ROS 1에서는 자체 개발한 TCPROS[14]와 같은 통신 라이브러리를 사용하고 있던 반면, ROS 2에서는 OMG(Object Management Group)에 의해 표준화된 DDS(Data Distribution Service)의 리얼타임 퍼블리시와 서브스크라이브 프로토콜인 DDSI-RTPS(Real Time Publish Subscribe)를 사용하고 있다. ROS 2 개발 초기에는 기존 TCPROS를 개선하거나 ZeroMQ, Protocol Buffers 및 Zeroconf 등을 이용하여 미들웨어처럼 사용하는 방법도 제안되었으나 무엇보다 산업용 시장을 위해 표준 방식 사용을 중요하게 여겼고, ROS 1 때와 같이 자체적으로 만들기 보다는 산업용 표준을 만들고 생태계를 꾸려가고 있었던 DDS를 통신 미들웨어로써 사용하기로 하였다. DDS 도입에 따라 그림 2 과 같이 ROS의 레이아웃은 크게 바뀌게 되었다. 처음에는 DDS 채용에 따른 장점과 단점에 대한 팽팽한 줄다리기 토론으로 걱정의 목소리도 높았지만 지금에 와서는 ROS 2에서의 DDS 도입은 상업적인 용도로 ROS를 사용할 수 있게 발판을 만들었다는 것에 가장 큰 역할을 했다는 평가가 지배적이다.

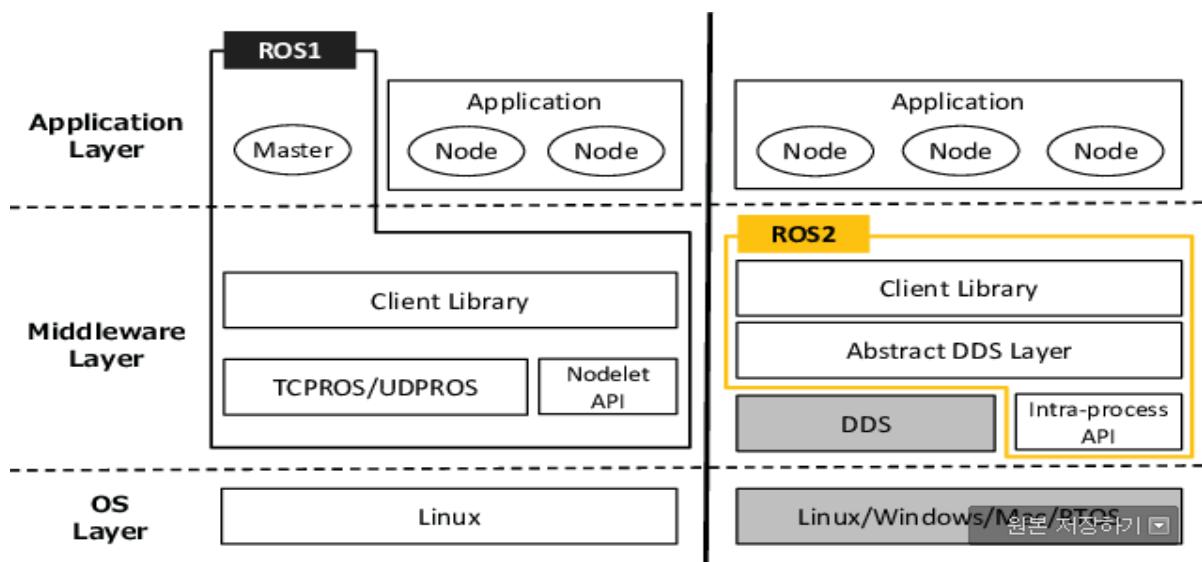


그림 12 DDS 사용에 따른 ROS의 구조 변화

DDS 도입으로 기존 메시지 형태 이외에도 OMG의 CORBA 시절부터 사용되던 IDL(Interface Description Language,)를 사용하여 메시지 정의 및 직렬화를 더 쉽게, 더 포괄적으로 다룰 수 있게 되었다. 또한 DDS의 중요 컨셉인 DCPS(data-centric publish-subscribe), DLRL(data local reconstruction layer)의 내용을 담아 재정한 통신 프로토콜인 DDSI-RTPS을 채용하여 실시간 데이터 전송을 보장하고 임베디드 시스템에도 사용할 수 있게 되었다. DDS의 사용으로 노드 간의 동적 검색 기능을 지원하고 있어서 기존 ROS 1에서 각 노드들의 정보를 관리하였던 ROS Master가 없어도 여러 DDS 프로그램 간에 통신할 수 있다. 또한 노드 간의 데이터 통신을 세부적으로 조정하는 QoS(Quality of Service)를 매개 변수 형태로 설정할 수 있어서 TCP처럼 데이터 손실을 방지함으로써 신뢰도를 높이거나, UDP처럼 통신 속도를 최우선시하여 사용할 수도 있다. 그리고 산업용으로 사용되는 미들웨어인 만큼 DDS-Security 도입으로 보안 측면에도 큰 혜택을 얻을 수 있었다. 이러한 다양한 기능을 갖춘 DDS를 이용하여 ROS 1의 퍼블리시, 서브스크라이브형 메시지 전달은 물론, 실시간 데이터 전송, 불안정한 네트워크에 대한 대응, 보안 등이 강화되었다. DDS의 채용은 ROS 1에서 ROS 2로 바뀌면서 가장 큰 변화점이자 그림 3과 같이 개발자 및 사용자로 하여금

통신 미들웨어에 대한 개발 및 이용 부담을 줄여 진짜로 집중해야 할 부분에 더 많은 시간을 쓸 수 있게 되었다.

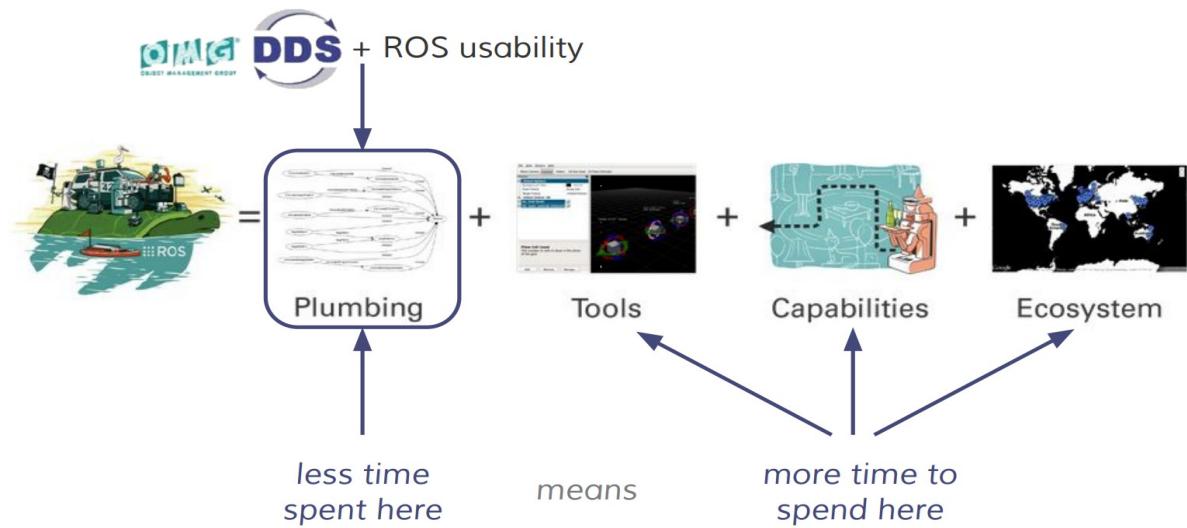


그림 13 DDS 사용에 따른 개발자 및 사용자 편의

DDS는 데이터 분산 시스템이라는 개념을 나타내는 단어이고 실제로는 데이터를 중심으로 연결성을 갖는 미들웨어의 프로토콜(DDSI-RTPS])과 같은 DDS 사양을 만족하는 미들웨어 API가 그 실체이다. 이 미들웨어는 그림 14 와 같이 ISO 7 계층 레이어에서 호스트 계층(Host layers)에 해당되는 4~7 계층에 해당되고 ROS 2 에서는 위에서 언급한 그림 13 과 같이 운영 체제와 사용자 애플리케이션 사이에 있는 소프트웨어 계층으로 이를 통해 시스템의 다양한 구성 요소를 보다 쉽게 통신하고 데이터를 공유할 수 있게 된다.

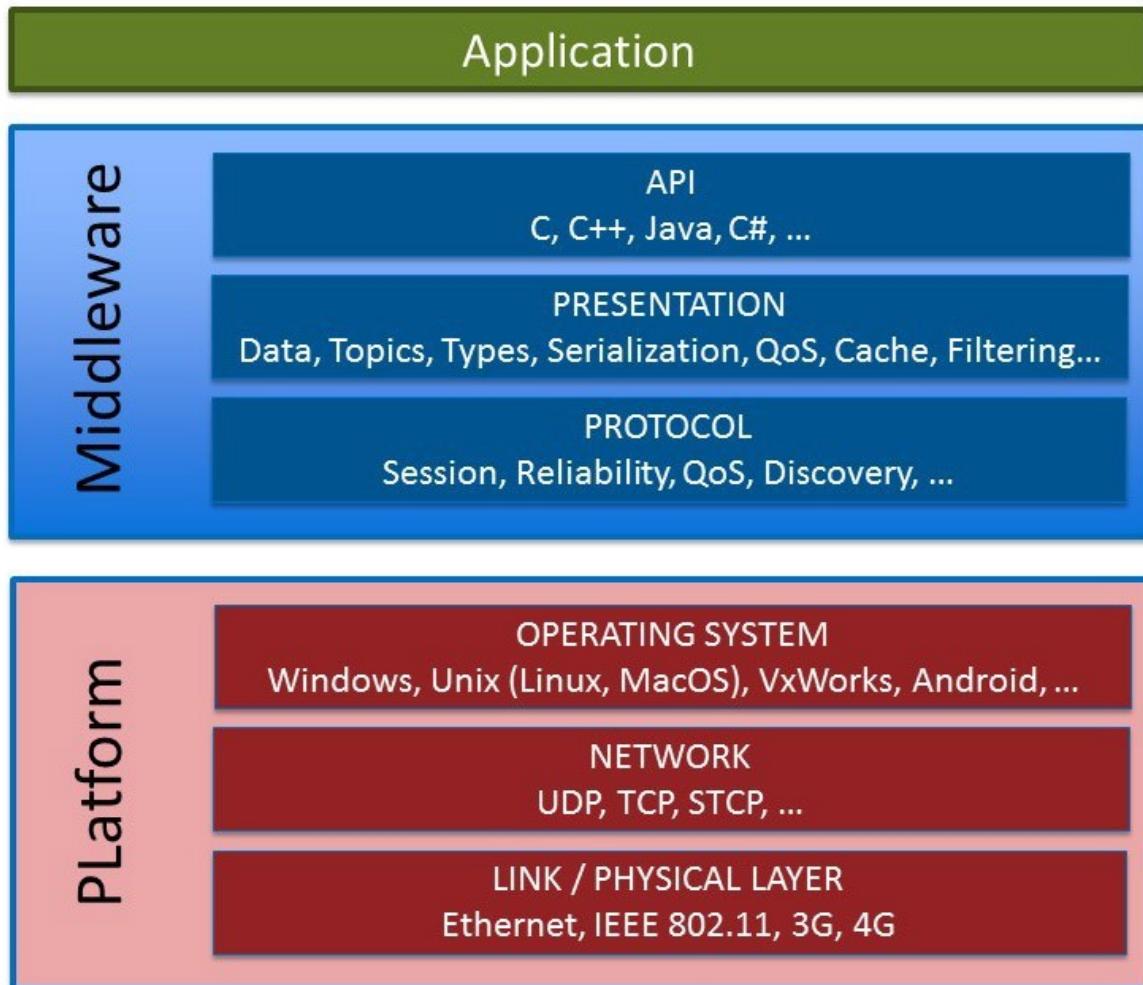


그림 14 미들웨어로서의 DDS

DDS의 특징은 다양하겠지만 DDS를 ROS 2의 미들웨어로 사용해보면서 느낀 장점은 아래와 같이 10 가지이다.

산업표준

DDS는 분산 객체에 대한 기술 표준을 제정하기 위해 1989년에 설립된 비영리 단체인 OMG(Object Management Group, 객체 관리 그룹)가 관리하고 있는 만큼 산업 표준으로 자리 잡고 있다. 지금까지 OMG가 진행하여 ISO 승인된 표준으로는 UML, SysML, CORBA 등이 있다. 2001년에 시작된 DDS 표준화 작업도 잘 진행되어 지금에 와서는 OpenFMB, Adaptive AUTOSAR, MD PnP, GVA, NGVA, ROS 2와 같은 시스템들에서 DDS를 사용하며 산업 표준의 기반이 되고 있다. ROS 1에서의 TCPROS는 독자적인 미들웨어라는 성격이 짙었는데 ROS 2에 와서는 DDS 사용으로 더 넓은 범위로 사용 가능하게 되었으며 산업 표준을 지키고 있는 만큼 로봇 운영체제 ROS가 IoT, 자동차, 국방, 항공,

우주 분야로 넓혀갈 수 있는 발판이 마련되었다.

운영체제 독립

DDS는 Linux, Windows, macOS, Android, VxWorks 등 다양한 운영체제를 지원하고 있기에 사용자가 사용하던 운영체제를 변경할 필요가 없다. 멀티 운영체제 지원을 컨셉으로 하고 있는 ROS 2 에도 매우 적합하다고 볼 수 있다.

언어 독립

DDS는 미들웨어이기에 그 상위 레벨이라고 볼 수 있는 사용자 코드 레벨에서는 DDS 사용을 위해 기존에 사용하던 프로그래밍 언어를 바꿀 필요가 없다. ROS 2에서도 이 특징을 충분히 살려 그림 15와 같이 DDS를 RMW(ROS middleware)으로 디자인되었으며 그 위에 사용자 코드를 위해 rclcpp, rclc, rclpy, rcljava, rclobjc, rclada, rclgo, rclnodejs 같이 다양한 언어를 지원하는 ROS 클라이언트 라이브러리 (ROS Client Library)를 제작하여 멀티 프로그래밍 언어를 지원하고 있다.

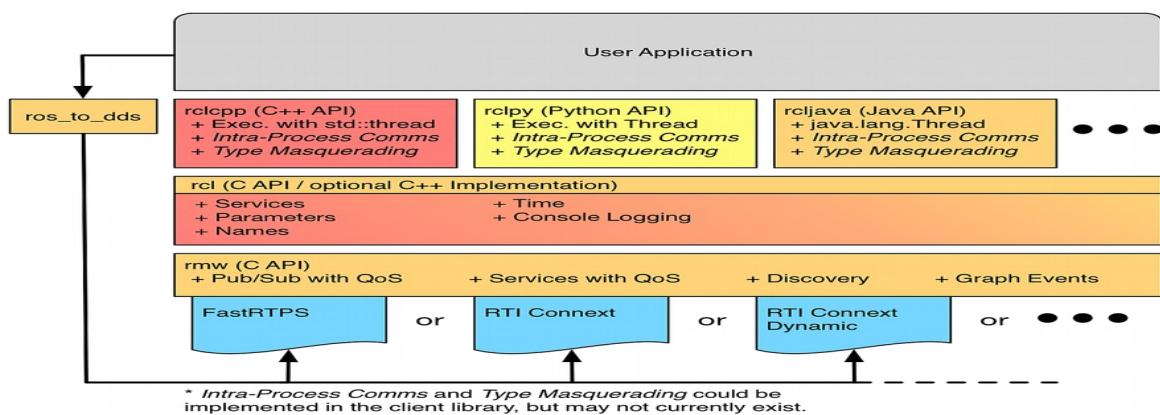


그림 15 ROS의 RMW, RCL, User Application

UDP 기반의 전송방식

DDS 벤더 별로 DDS Interoperability Wire Protocol (DDSI-RTPS, [17])의 구현 방식에 따라 상이할 수 있으나 일반적으로 UDP 기반의 신뢰성 있는 멀티캐스트(reliable multicast)를 구현하여 시스템이 최신 네트워킹 인프라의 이점을 효율적으로 활용할 수 있도록 돋고 있다. UDP 기반이라는 것이 ROS 1에서의 TCPROS가 TCP 기반이었던 것에 비해 매우 큰 변화인데 UDP의 멀티캐스트 (multicast)는 브로드캐스트(broad-cast)처럼 여러 목적지로 동시에 데이터를 보낼 수 있지만, 불특정 목적지가 아닌 특정된 도메인 그룹에 대해서만 데이터를 전송하게 된다. 참고로 ROS 2에서는 `ROS_DO_MAIN_ID`라는 환경 변수로 도메인을 설정하게 된다. 이 멀티캐스트의 방식 도입으로 ROS 2에서는 전역 공간이라 불리는 DDS Global Space이라는 공간에 있는 토픽들에 대해 구독 및 발행을

할 수 있게 된다. Best effort 개념인 UDP는 reliable을 보장하는 TCP에 비해 장단점이 있는데 이 또한 후에 설명하는 QoS(Quality of Service)를 통해 보완 및 해결되었다.

데이터 중심적 기능

다양한 미들웨어가 있겠지만 그중 DDS를 사용하면서 제일 많이 듣는 말 중에 하나는 `Data Centric`이라는 것이다. 우리말로는 데이터 중심적이라는 것인데 실제로 DDS를 사용하다보면 이 말이 이해가 된다. DDS 사양에도 DCPS(data-centric publish-subscribe)이라는 개념이 나오는데 이는 적절한 수신자에게 적절한 정보를 효율적으로 전달하는 것을 목표로 하는 발간 및 구독 방식이라는 것이다. DDS의 미들웨어를 사용자 입장에서 본다면 어떤 데이터인지, 이 데이터가 어떤 형식인지, 이 데이터를 어떻게 보낼 것인지, 이 데이터를 어떻게 안전하게 보낼 것인지에 대한 기능이 DDS 미들웨어에 녹여있기 때문이다.

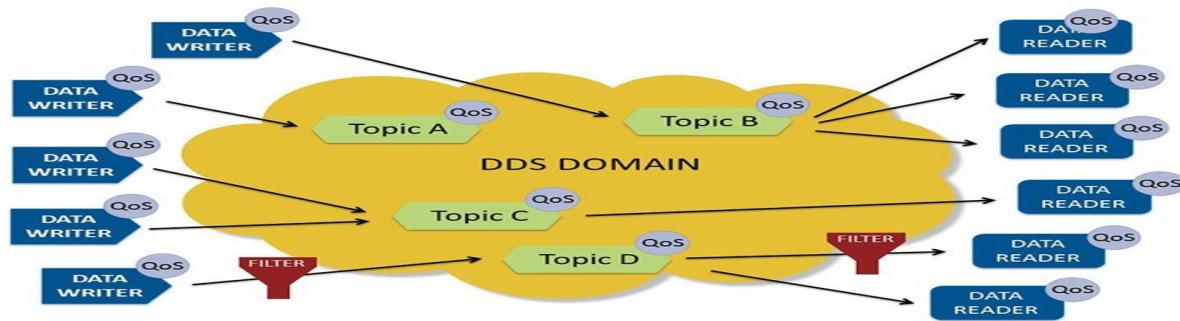


그림 16 데이터 중심성의 도식화

동적 검색

DDS는 동적 검색(Dynamic Discovery)을 제공한다. 즉, 응용 프로그램은 DDS의 동적 검색을 통하여 어떤 토픽이 지정 도메인 영역에 있으며 어떤 노드가 이를 발신하고 수신하는지 알 수 있게 된다. 이는 ROS 프로그래밍할 때 데이터를 주고받을 노드들의 IP 주소 및 포트를 미리 입력하거나 따로 구성하지 않아도 되며 사용하는 시스템 아키텍처의 차이점을 고려할 필요가 없기 때문에 모든 운영 체제 또는 하드웨어 플랫폼에서 매우 쉽게 작업할 수 있다. ROS 1에서는 ROS Master에서 ROS 시스템의 노드들의 이름 지정 및 등록 서비스를 제공하였고, 각 노드에서 퍼블리시 또는 서브스크라이브하는 메시지를 찾아서 연결할 수 있도록 정보를 제공해 주었다. 즉, 각각 독립되어 실행되는 노드들의 정보를 관리하여 서로 연결해야 하는 노드들에게 상대방 노드의 정보를 건네주어 연결할 수 있게 해 주는 매우 중요한 중매 역할을 수행했었다. 이 때문에 ROS 1에서는 노드 사이의 연결을 위해 네임 서비스를 마스터에서 실행했어야 했고, 이 ROS Master가 연결이 끊기거나 죽는 경우 모든 시스템이 마비되는 단점이 있었다. ROS 2에서는 ROS Master가 없어지고 DDS의 동적 검색 기능을 사용함에 따라 노드를 DDS의 Participant 개념으로 취급하게 되었으며, 동적 검색 기능을 이용하여 DDS 미들웨어를

통해 직접 검색하여 노드를 연결할 수 있게 되었다.

확장 가능한 아키텍처

OMG 의 DDS 아키텍처는 IoT 디바이스와 같은 소형 디바이스부터 인프라, 국방, 항공, 우주 산업과 같은 초대형 시스템으로까지 확장할 수 있도록 설계되었다. 그렇다고 사용하기 복잡한 것도 아니다. DDS의 Participant 형태의 노드는 확장 가능한 형태로 제공되어 사용할 수 있으며 단일 표준 통신 계층에서 많은 복잡성을 흡수하여 분산 시스템 개발을 더욱 단순화 시켜 편의성을 높였다. 특히 ROS 와 같이 최소 실행 가능한 노드 단위로 나누어 수백, 수천 개의 노드를 관리해야 하는 시스템에서는 이 부분이 강점으로 보이며 한대의 로봇이 아닌 복수의 로봇, 주변 인프라와 다양한 IT 기술, 데이터베이스, 클라우드로 연결 및 확장해야 하는 ROS 시스템에 매우 적합한 기능이다.

상호 운용성

ROS 2 에서 통신 미들웨어로 사용하고 있는 DDS 는 상호 운용성을 지원하고 있다. 즉, DDS 의 표준 사양을 지키고 있는 벤더 제품을 사용한다면 A 라는 회사의 제품을 사용하였다가도 B 라는 회사 제품으로 변경이 가능하고, A 제품과 B 제품을 혼용하여 서로 다른 제품의 DDS 제품을 사용하더라도 A 제품과 B 제품간의 상호 통신도 지원한다는 것이다. 현재 DDS 벤더로는의 리스트와 같이 10 곳이 있는데 이 중 ROS 2 를 지원하는 업체는 ADLink, Eclipse Foundation, Eprosima, Gurum Network, RTI 로 총 5 곳이며 DDS 제품명으로는 ADLINK 의 OpenSplice, Eclipse Foundation 의 Cyclone DDS, Eprosima 의 Fast DDS, Gurum Network 의 Gurum DDS, RTI 의 Connex DDS 가 있다. 이 중 Fast DDS 와 Cyclone DDS 는 오픈 소스를 지향하고 있기에 자유롭게 사용 가능하며 더 고성능을 위한다면 상용 제품인 OpenSplice, Connex DDS, Gurum DDS 를 사용하면 된다. 참고로 이 중 구름네트웍스(Gurum Network)는 유일하게 대한민국 기업으로 DDS 를 순수 국산 기술로 개발하여 상용화에 성공한 기업이다.

OMG Member Vendors



그림 17 DDS 의 공식 벤서들

서비스 품질

ROS 2 에서는 DDS 도입으로 데이터의 송수신 관련 설정을 목적에 맞추어 유저가 직접 설정할 수 있게 되었다. 노드 간의 DDS 통신 옵션을 설정하는 QoS(Quality of Service)가 그것인데 퍼블리셔 및 서브스크라이브 등을 선언하고 사용할 때 매개 변수처럼 QoS 를 사용할 수 있다.

DDS 사양상 설정 가능한 QoS 항목은 22 가지인데 ROS 2 에서는 현재 TCP 처럼 데이터 손실을 방지함으로써 신뢰도를 우선시하거나 (reliable), UDP 처럼 통신 속도를 최우선시하여 사용(best effort)할 수 있게 하는 신뢰성(reliability) 기능이 대표적으로 사용되고 있다. 그 이외에 또한 통신 상태에 따라 정해진 사이즈만큼의 데이터를 보관하는 History 기능, 데이터를 수신하는 서브스크라이버가 생성되기 전의 데이터를 사용할지 폐기할지에 대한 설정인 Durability 기능, 정해진 주기 안에 데이터가 발신 및 수신되지 않을 경우 이벤트 함수를 실행시키는 Deadline 기능, 정해진 주기 안에서 수신되는 데이터만 유효 판정하고 그렇지 않은 데이터는 삭제하는 Lifespan 기능, 정해진 주기 안에서 노드 혹은 토픽의 생사 확인하는 Liveliness 도 설정할 수 있다. 이러한 다양한 QoS 설정을 통해 DDS 는 적시성, 트래픽 우선순위, 안정성 및 리소스 사용과 같은 데이터를 주고받는 모든 측면을 사용자가 제어할 수 있게 되었고 특정 상황, 예를 들어 매우 빠른 속도 데이터를 주고받거나 매우 역동적이고 까다롭고 예측할 수 없는 통신환경에서 데이터 송/수신에 다양한 옵션 설정으로 이를 달성하거나 장애를 극복할 수 있게 되었다.

보안

ROS 1 의 가장 큰 구멍이었던 보안 부분은 ROS 2 개발에서 DDS 으로 해결되었다. DDS 의 사양에는 DDS-Security 이라는 DDS 보안 사양을 ROS 에 적용하여 보안에 대한 이슈를 통신단부터 해결하였다.

또한 ROS 커뮤니티에서는 SROS 2(Secure Robot Operating System 2)라는 툴을 개발하였고 보안 관련 RCL 서포트 및 보안 관련 프로그래밍에 익숙지 않은 로보틱스 개발자를 위해 보안을 위한 툴킷을 만들어 배포하고 있다.

2. ROS2 개발환경 설정

본 ROS2 교재는 Ubuntu Desktop 18.04 LTS 버전을 사용하고 있고, ROS 는 ROS2 Dashing Diademata 를 사용하도록 한다. ROS 는 각 OS 버전에 맞는 Release 버전을 사용하고 있으며 해당 Release 버전에 맞는 OS 를 설치하여야 한다.

Architecture	Ubuntu Bionic (18.04)	MacOS Sierra (10.12)	Windows 10 (VS2019)	Debian Stretch (9)	OpenEmbedded / webOS OSE
amd64	Tier 1 [d][a][s]	Tier 1 [a][s]	Tier 1 [a][s]	Tier 3 [s]	
arm64	Tier 1 [d][a][s]			Tier 3 [s]	Tier 3 [s]
arm32	Tier 2 [a][s]			Tier 3 [s]	Tier 3 [s]

그림 18 ROS2 Dashing Diademata 호환 OS

Package	Required Support			Recommended Support	
	Ubuntu Bionic	MacOS**	Windows 10**	Debian Stretch	OpenEmbedded**
CMake	3.10.2	3.14.4	3.14.4	3.7.2	3.16.1 / 3.12.2***
EmPY	3.3.2				
Gazebo	9.0.0	9.9.0	N/A	9.8.0*	N/A
Ogre	1.10*				N/A
OpenCV	3.2.0	4.1.0	3.4.6*	3.2*	4.1.0 / 3.2.0***
OpenSSL	1.1.0g	1.0.2r	1.0.2r	1.1.0j	1.1.1d / 1.1.1b***
Poco	1.8.0	1.9.0	1.8.0*	1.8.0*	1.9.4
Python	3.6.5	3.7.3	3.7.3	3.5.3	3.8.2 / 3.7.5***
Qt	5.9.5	5.12.3	5.10.0	5.7.1	5.14.1 / 5.12.5***
	Linux only				
PCL	1.8.1	N/A	N/A	1.8.0	1.8.1
RMW DDS Middleware Providers					
Connext DDS	5.3.1			N/A	
Cyclone DDS	0.7.x (Coquette)				
Fast-RTPS	1.8.0				
OpenSplice	6.9.190403OSS				N/A

그림 19 ROS2 Dashing Diademata 의존성

2.1. 리눅스의 사용환경과 구조

2.1.1. 리눅스의 시작



그림 20 리눅스를 개발한 리눅스 토발즈(왼쪽)와 미닉스를 개발한 교수 앤드류 타넨바움(오른쪽)

1991년

9월 17일, 핀란드 헬싱키대학에 다니던 리눅스 토발즈(Linus Benedict Torvalds)는 취미 삼아 개발하던 커널을 인터넷에 공개하였는데, 이 커널이 버전 0.1이라고 불리는 리눅스 운영체제의 시작이었다.

초창기의 리눅스는 앤드루 타넨바움(Andrew Tanenbaum)이 교육용으로 만든 미닉 스(MINIX)라는 유닉스 커널을 기반으로 제작되었지만, 인터넷에 공개된 지 얼마되지 않아 수많은 해커들의 열광 속에

미닉스를 능가하게 되었다. 이러한 분위기 속에 리눅스는 빠르게 발전하여 1994년에 안정적인 리눅스 커널 1.0이 발표되었고, 2년이 채 지나지 않은 1996년에 리눅스 커널 2.0이 발표되었다. 리눅스 커널 2.0이 발표되면서 IBM, 컴팩(Compaq), 오라클(Oracle) 같은 회사들이 리눅스를 공식으로 지원하기 시작하였고, 수많은 해커들과 회사들의 도움으로 급격한 성장을 이루었다.

2.1.2. 리눅스의 특징

리눅스는 기본적인 유닉스(UNIX) 운영체제의 특징을 가지고 있다. 동시에 여러 사람(multi-user)이 한꺼번에 사용할 수 있는 시스템이며, 여러 프로그램을 실행할 수 있는 멀티프로세스(multiprocess) 환경을 지원하고, 여러 개의 CPU를 지원하는 멀티프로세서(multi-processor) 시스템을 지원한다.

리눅스는 오픈 소스로 공개로 개발되는 유닉스 계열의 운영체제이며 GNU 도구를 사용해서 개발되고 있다. 초기에는 인텔의 i80386에서 개발된 후 여러 시스템으로 포팅되었으며 1990년대 초부터 개발을 시작해서 1990년대 말까지 서버 시장에서 유닉스나 윈도우의 대안으로 자리를 잡았으며, 1998년에 Dragonball 68k 시리즈에 처음으로 포팅하면서 시작한 HClinux(MicroController Linux)와 함께 2000년대부터 임베디드 시장을 석권하기 시작하였다.

초기의 리눅스는 MMU(Memory Management Unit)가 있는 시스템에서만 설치가 되었지만 jiClinux는 MMU를 지원하지 않은 32비트 프로세서를 지원하였고 2003년 12월에 출시된 리눅스 커널 2.6에서부터 기본으로 임베디드를 지원하면서 임베디드 시장뿐만 아니라 안드로이드 같은 모바일에서도 보다 폭넓게 사용되었다.

리눅스는 유닉스의 표준 규정인 POSIX(Portable Operating System Interface)를 지원하기

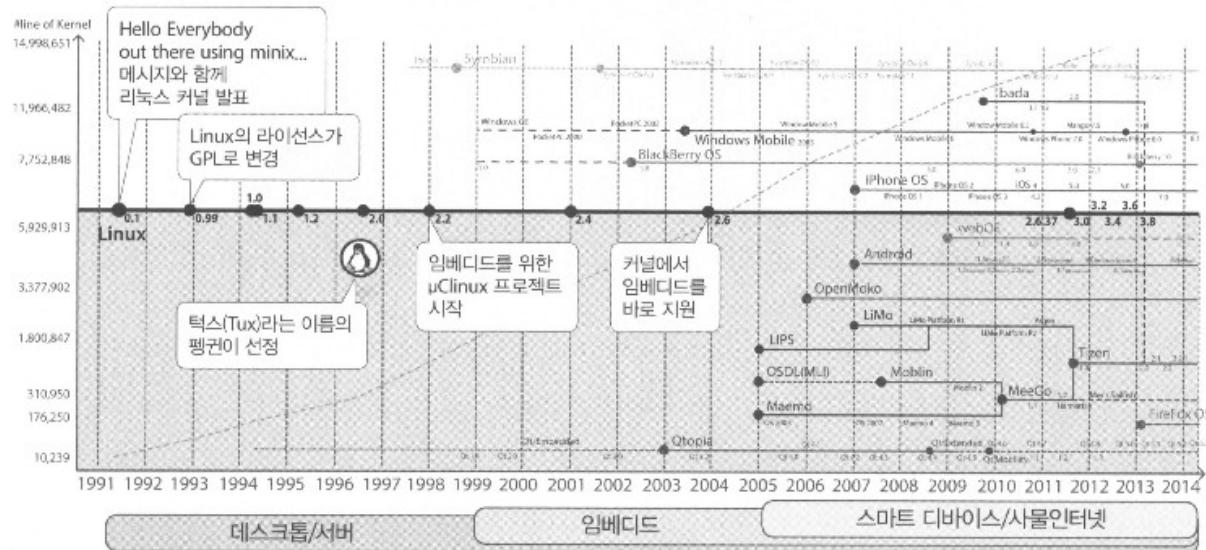


그림 21 리눅스 커널의 발전과 스마트 플랫폼

때문에 다른 유닉스에서 개발된 애플리케이션을 쉽게사용할 수이었으며, X 윈도 우같은 유닉스의 표준 GUI 시스템이 지원된다. 그리고 유닉스의 네트워크(BSD 소켓), IPC(Inter-Process Communication)나 프로세스 스케줄링(시분할 처리), 버퍼 캐시 (buffer cache)와 페이징(paging), 스레드(POSIX thread) 등도 지원한다.

유닉스는 여러 사용자들이 동시에 사용할 수 있는데, 각각의 사용자는 자신의 계정(user account)과 비밀번호, 그리고 홈 디렉터리(home directory)라는 저장공간을 가지고 있다. 유닉스는 사용자 계정과 비밀번호를 이용해서 사용자를 구분하며, 계정 별로 접근 권한의 제약을 둘으로써 기본적인 시스템 보안을 제공한다.

2.1.3. 리눅스의 구조

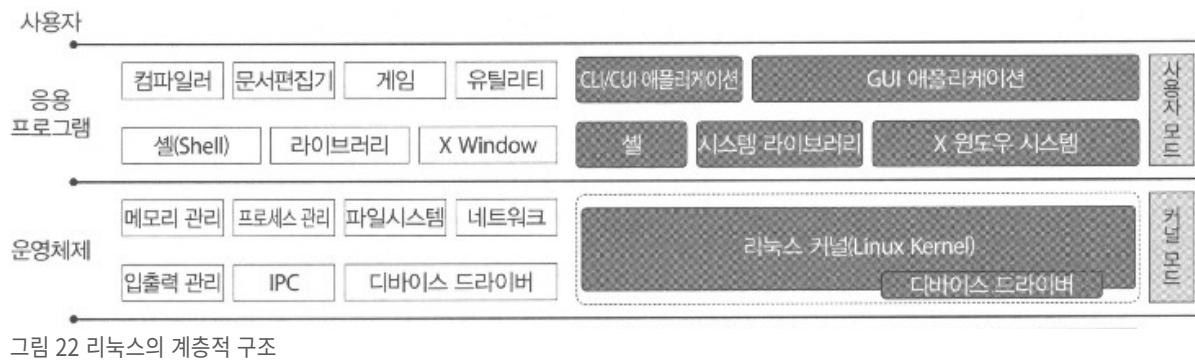


그림 22 리눅스의 계층적 구조

리눅스는 리눅스 토발즈에 의해서 개발된 커널 자체를 의미하기도 하지만 다양한 GNU 소프트웨어가 합쳐진 운영체제를 의미하기도 한다. 운영체제는 기본적으로 컴퓨터 자원의 효율적인 사용성과 사용자가 보다 쉽게 사용할 수 있는 편의성을 제공해야 하는데, 우분투와 같은 리눅스 배포판에서 리눅스 커널과 GNU 소프트웨어를 통해서 이러한 기능을 제공한다.

리눅스는 크게 커널, 디바이스 드라이버, 시스템 라이브러리, 셸(Shell), 유ти리티, X 윈도우로 나눌 수 있다. 먼저 커널(kernel)은 중심부, 핵심이라는 뜻인데, 리눅스 커널은 실제 운영체제(Operating System)를 구성하는 기본적인 토대로서 시스템의 가장 기본적인 메모리나 프로세스 등의 하드웨어를 관리하고, 애플리케이션이 커널을 이용할 수 있도록 시스템 호출(system call) 같은 API(Application Programming Interface)를 제공한다. 또한 커널은 하드웨어의 제어를 위한 디바이스 드라이버(device driver)를 포함한다.

이러한 커널 위에 GNU에서 개발한 다양한 리눅스 유ти리티와 glibc 같은 다양한 라이브러리들이 위치한다. 사용자는 이러한 유ти리티들을 통해서 리눅스를 사용할 수 있으며, 유ти리티는 커널에서 제공하는 시스템 호출인터페이스를 통해서 커널과 통신할 수 있다.

2.2. Ubuntu Linux 설치

2.2.1. Linux 와 Ubuntu

Ubuntu 는 캐노니컬이 개발한 컴퓨터 운영체제로 일반적으로 6 개월마다 새판이 하 나씩 나오게 되는데 이는 GNOME 의 새 판이 나오는 시기와 비슷하다. 데비안 GNU/ 리눅스와 견주어 볼 때 사용자 편의성에 많은 초점을 맞추고 있다는 것이 특징이다.



그림 23 Ubuntu 가동 모습

Ubuntu 라는 이름은 남아프리카공화국의 건국 이념인 ‘우분투정신’에서 가지고 왔다. 일반적으로 ‘네가 있으니 내가 있다’라는 의미로 ‘다른 사람을 위한 인간애(Humanity towards others)’라는 뜻으로도 사용되고 있다.

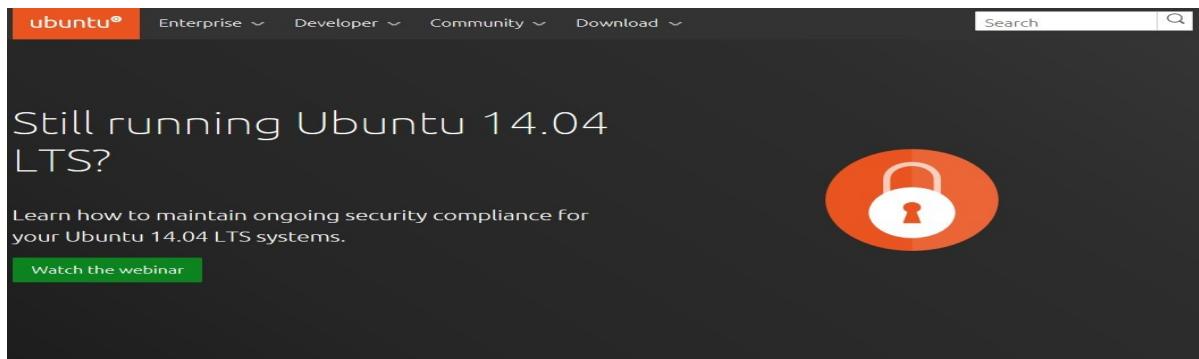
우분투는 배포판을 수정하거나 수정한 것을 재배포 할 수 있는 자유소프트웨어로, 지금까지 수많은 변형 배포판이나 공식 지원하지 않은 창 관리자를 데스크톱으로 하는 배포판들이 나왔다. 우분투의 기본 철학, 즉 전세계의 사람 누구나 어렵지 않게 리눅스를 사용하자는 표어에서 알 수 있듯이, 기본적으로 세계의 다양한 언어를 지원하고 그다지 높은 사양의 컴퓨터가 필요하지 않다.

우분투는 사용자가 손쉽게 운영 체제를 설치하고 사용할 수 있도록 설계되었다. 이를테면 7.10 의 경우, CD 를 넣고 시동한 다음, 일곱 단계만 거치면 바로 사용할 수 있도록 설치된다. Dapper(6.06) 버전부터 데스크톱 CD 에는 유비쿼티가 들어 있어 컴퓨터의 재시동 없이 데스크톱 CD 를 구동하는 동안 우분투를 시스템에 설치할 수 있는 기능이 있다. 또한, 아직까지 그래픽 카드에 따라 데스크톱 관리자가 시동이 되지 않는 경우가 많기 때문에, 텍스트 기반에서 설치를 할 수 있는 얼터너티브 CD 를 따로 배포한다. 버전 11.04 부터는 유니티가 기본 데스크탑으로 탑재된다.

또한 시스템 관리 작업에서 sudo 도구를 사용한다는 점이 있다. sudo는 사용자가 자신의 암호를 이용해 시스템 관리 권한을 얻어 작업을 진행할 수 있도록 인증한다. 따라서 관리 작업을 하기 위해 root 사용자의 암호를 따로 만들지 않아도 되고, 여러 사용자가 관리를 위해 root 암호를 공유함으로써 생길 수 있는 잠재적인 보안문제를 예방할 수 있다. 우분투에서는 1 회 sudo 인증이 완료되면 기본적으로 5 분 동안 다시 인증하지 않고도 시스템 관리를 계속할 수 있다. 또한, '제한된 장치 관리자'를 사용할 수 있는데, 이것을 통해 사용자는 드라이버를 번거롭게 직접 설치하는 과정을 거칠 필요 없이, 버튼을 몇 번 누르는 것으로 장치를 설정할 수 있다. 7.10 이후부터는 자동으로 제한된 장치 드라이버를 검색해 적용해 준다. 그래픽 카드 드라이버의 경우 설치하면 기능이 비정상적으로 작동할 가능성이 있기 때문에 사용자에게 드라이버를 설치할 것인지를 물어본다.

2.2.2. Ubuntu 설치 USB 만들기

Ubuntu 설치를 위해서는 우선 설치 iso 파일이 필요하다. 우선 우분투 공식홈페이지 (www.ubuntu.com)에 접속하도록 한다.



Latest news from our blog

[Ubuntu is #1 for embedded & IoT](#)

26 April 2019

[An introduction to AppArmor](#)

25 April 2019

[Visit Canonical at Dell Technologies World](#)

24 April 2019

Spotlight

Open infrastructure, developer desktop and IoT are the focus for Ubuntu 19.04

18 April 2019

그림 24 Ubuntu 공식홈페이지 메인화면

상단의 네비게이션 메뉴에서 Download 를 클릭하고 나오는 메뉴에서 Ubuntu Desktop 을 클릭하도록 하면 2018년 4월에 설치된 18.04 버전과 2020년 4월에 출시된 19.04 가 가장 먼저



그림 25 Alternative Downloads 버튼 위치

눈에 들어오게된다. 하지만 우리는 18.04 를 설치해야하기 때문에 상단에 있는 Alternative downloads 를 클릭하도록 한다.

Alternative downloads 에서 중간에 있는 Network installer 의 'Download the network installer for 08.04 LTS'를 클릭하여 다운로더 사이트로 접속하면 아래와 같은 페이지가 표시되게 된다. 해당 페이지에서 amd64 용 커널을 클릭하여 다운로드를 하도록 한다.

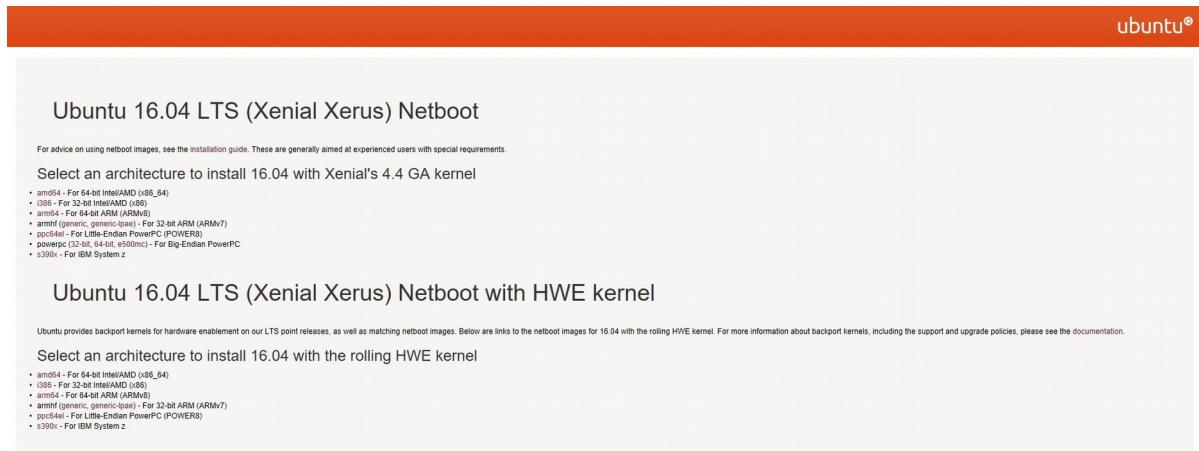


그림 26 Ubuntu LTS installer 페이지 뷰

다운로드가 완료되었으면 4Gb 이상의 usb 메모리를 하나 준비하고 설치 usb 를 생성 하는 프로그램을 다운받도록 한다. 본 교재에서는 rufus 를 사용하고 있으며 포털 사이트에서 검색 후 다운로드를 받아 설치 usb 를 생성하면 된다.

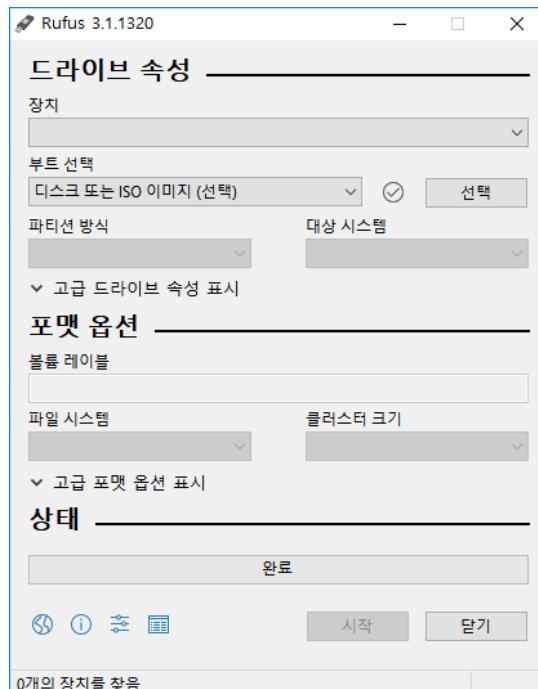


그림 27 rufs main 실행 페이지

2.2.3. Ubuntu 설치



그림 28 언어 설정

완성한 설치 usb를 pc에 꽂고 cmos 설정에서 usb를 설정 후 재부팅을 하게 되면 ubuntu 설치 화면으로 넘어가게 된다. 가장 먼저 언어 설정을 할 수 있도록 되어 있다. 기본 언어는 영어로 설정되어 있으며 좌측의 언어를 가장 아래로 내리게 되면 한 국어로 설정할 수 있다.

한국어 선택 후 오른쪽에 표시되는 ‘ubuntu 체험하기’와 ‘ubuntu 설치’ 버튼이 나타나게 된다. 이 중 ‘ubuntu 설치’를 클릭하고 다음 화면으로 넘어가면 된다. 다음 설정 화면은 ubuntu 기본 업데이트사항을 다운로드할지 설정하는체크박스와 서드파티 프로그램을 설정할지 결정하는 체크박스가 나오게 된다. 이 박스는 본인이 편한대로 체크하고 다음버튼을 클릭하도록 한다.

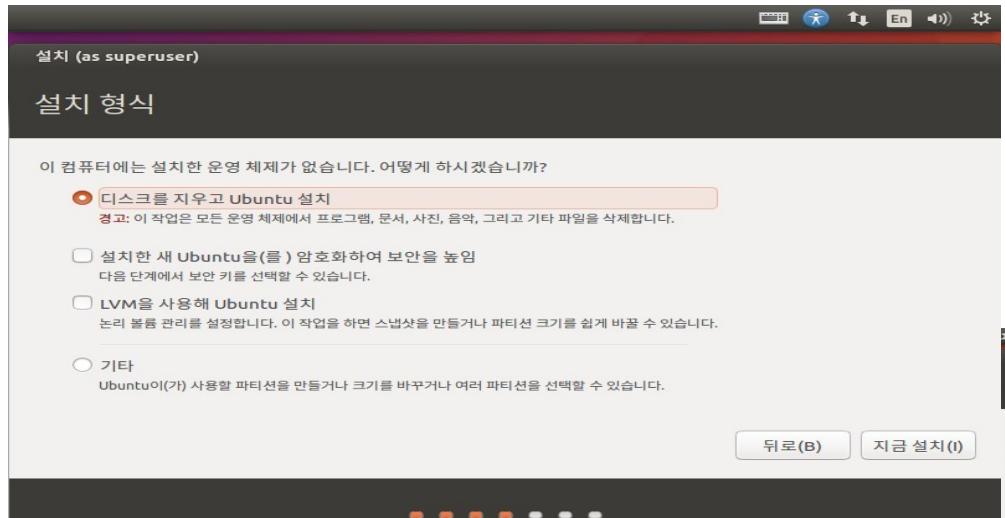


그림 30 ubuntu 설치 형식



그림 29 업데이트 다운로드 결정화면

다음으로 설치형식이 나오게 되는데, 기존의 windows를 사용하면서 ubuntu를 사용 할 수 있도록 하는 듀얼부팅이 이루어질 수 있도록 하기 위해서는 ‘기타’탭을 선택하여 파티션 조절 및 부트로더 설치 경로를 설정해주도록 한다.

시간대 및 키보드 설정을 하는 창이 나오게 된다. 이 창에서는 Seoul 을 선택 후 계속 버튼을 누르게 되면 키보드 배치 창이 나오게 된다 좌측의 키보드 배치 선택 부분에서 한국어로 선택 후 한국어-한국어 (101/104 키 호환)을 선택 후 계속 버튼을 누른다.

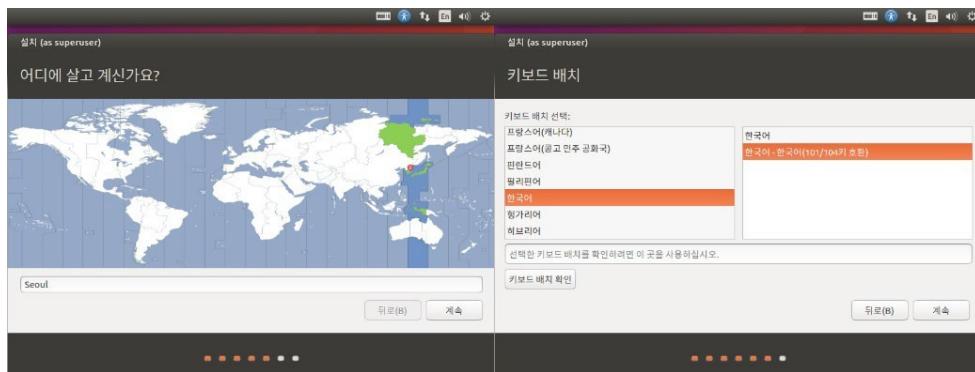


그림 31 ubuntu 시간대 설정 및 키보드 배치

2.2.4. Ubuntu 관련 유용한 Tool

지금부터 ROS를 설치하고 나면 바로 ROS 개발을 개발환경이 완료되지만 몇가지 유용한 Tool을 추가로 설치함으로써 보다 효과적인 개발이 이루어질 수 있도록 하다. 크게 2 가지를 소개할 것이며, 여러가지 Node를 하나의 터미널창에서 구동하기 위한 다중터미널창인 Terminator와 다중 Monitor view를 구현할 수 있는 Unity tweak Tool을 설치하여 보다 효율적인 개발이 이루어질 수 있도록 하자.

Terminator

Terminator는 위에서 말한 것처럼 하나의 터미널에서 창분할을 할 수 있도록 되어 있는 터미널 프로그램이다. 1 장에서 소개한 것처럼 ROS는 터미널창에서 각 Node를 실행해야 하기 때문에 Ubuntu에서 기본으로 제공해주는 터미널 창을 사용하게 된다면 화면이 매우 복잡해져 각 노드들이 잘 작동하고 있는지 확인하기가 어렵다. 이 Terminator 프로그램을 사용하면 하나의 창에서 여러 노드를 동시에 실행할 수 있어 Node들에 대한 작동 정보를 바로 확인할 수 있다는 장점이 있다.

```
$ sudo apt-get install terminator
```

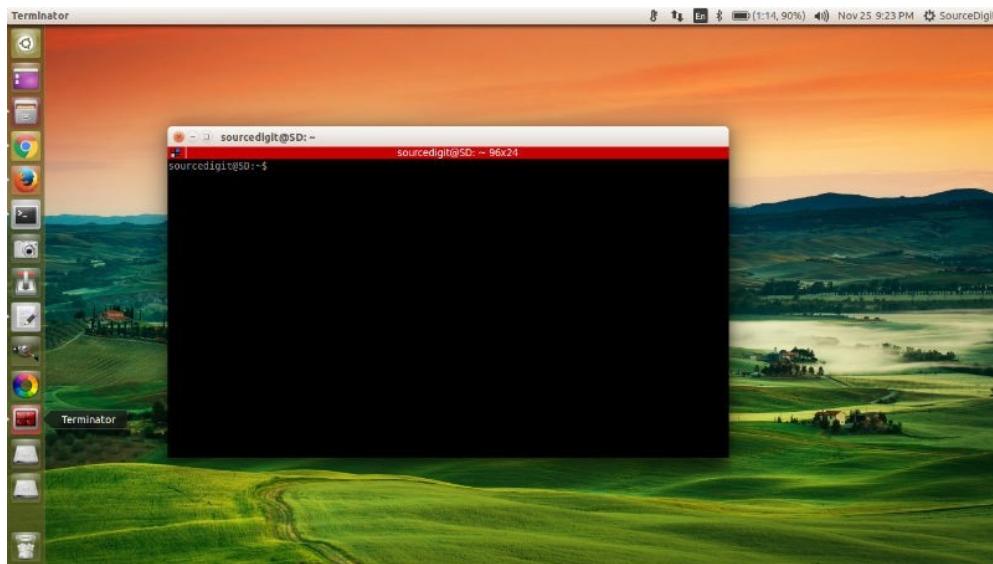


그림 32 Terminator

Gnome-tweak tool

Gnome tweak tool 은 그놈 기반의 리눅스의 제어판과 같은 역할을 하고 있으며 기존의 우분투 설정창에서 설정할 수 없는 다양한 설정 기능을 제공하고 있는 Tool 이다. 이 Gnome tweak tool 은 위에서 설명한 Terminator 와 동일하게 설치가 가능하다.

```
$ sudo apt-get update  
$ sudo apt-get install gnome-tweak-tool
```

설치가 완료되고 나면 터미널창에서 unity tweak tool 을 실행하도록 하자.

```
$ unity-tweak-tool
```

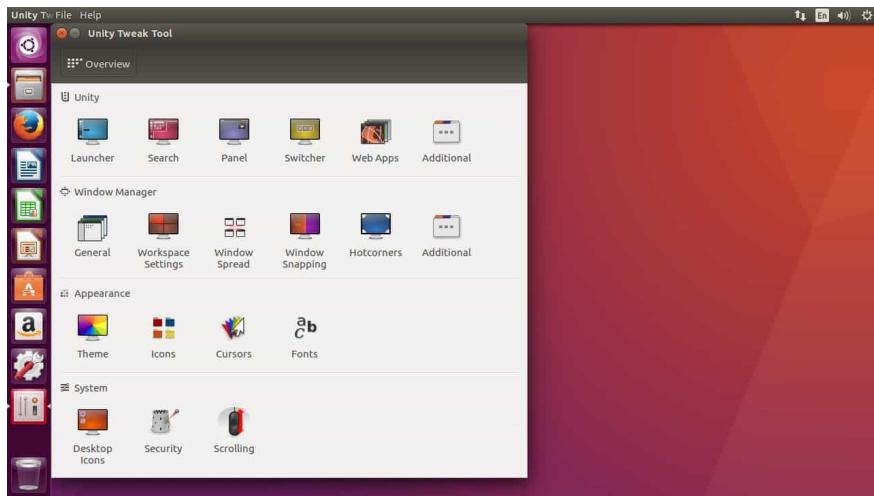


그림 33 Gnome tweak tool 실행 화면

Gnome tweak tool에서 2 번째 탭인 window Manager에서 workspace Settings를 클릭하면
다중바탕화면의 수를 선택할 수 있다. 본인이 원하는 만큼 설정 후 저장 버튼을 누르게 되면 러처에
workspace 선택 아이콘이 나오게 된다. 이 아이콘을 클릭하게 되면 본인이 생성한 수 만큼의
workspace 가 나오게 되고 해당 workspace에서 작업을 수행하면 된다. 단축키로 이동을 하고
싶다면 Ctrl + Alt + 방향키로 각 workspace로 이동이 가능하다.

2.3. ROS Dashing 설치

먼저 ROS2 저장소를 등록하기 위해 GPG 키를 승인한다.

```
$ sudo apt update && sudo apt install curl gnupg2 lsb-release  
$ sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -o /usr/share/  
keyrings/ros-archive-keyring.gpg
```

다음 시스템에 ROS2 용 저장소 repository 를 추가한다.

```
$ echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/ros-archive-  
keyring.gpg] http://packages.ros.org/ros2/ubuntu $(lsb_release -cs) main" | sudo tee  
/etc/apt/sources.list.d/ros2.list > /dev/null
```

저장소 repository 가 추가되었다면 repository 업데이트 후 dashing 버전을 설치한다.

```
$ sudo apt update && sudo apt install -y ros-dashing-desktop
```

다음으로 개발 환경 설정을 진행한다. ROS Dashing 의 setup 파일을 bashrc 에 추가한 후 빌드 tool 을 설치한다.

```
$ source /opt/ros/$name_ros_version/setup.sh  
sudo apt install -y python3-argcomplete python3-colcon-common-extensions python3-vcstool  
git
```

설치가 완료되었다면 개발 workspace 를 설정한다. 우리가 생성할 workspace 이름은 colcon_ws 로 설정할 것이다. 폴더를 생성한 후 최초 build 를 수행한다.

```
$ mkdir -p $HOME/$name_colcon_workspace/src  
$ cd $HOME/$name_colcon_workspace  
$ colcon build --symlink-install
```

마지막으로 ros2 와 관련된 간단한 단축키 세팅 및 workspace 의 setup.bash 파일을 bashrc 파일에 추가해준다.

```
$ echo \"alias nb='nano ~/.bashrc'\" >> ~/.bashrc  
$ echo \"alias sb='source ~/.bashrc'\" >> ~/.bashrc  
$ echo \"alias gs='git status'\" >> ~/.bashrc  
$ echo \"alias gp='git pull'\" >> ~/.bashrc  
$ echo \"alias cw='cd ~/{$name_colcon_workspace}'\" >> ~/.bashrc  
$ echo \"alias cs='cd ~/{$name_colcon_workspace}/src'\" >> ~/.bashrc  
  
$ echo \"alias cb='cd ~/{$name_colcon_workspace} && colcon build --symlink-install && source  
~/{$name_colcon_workspace}/src/.bashrc'\" >> ~/.bashrc  
  
$ echo \"source /opt/ros/$name_ros_version/setup.bash\" >> ~/.bashrc  
$ echo \"source ~/{$name_colcon_workspace}/install/local_setup.bash\" >> ~/.bashrc
```

정상적으로 ROS2 가 설치되어있는지 확인하기 위해 turtlesim 이라는 node 를 실행해본다.

```
$ ros2 run turtlesim turtlesim_node
```

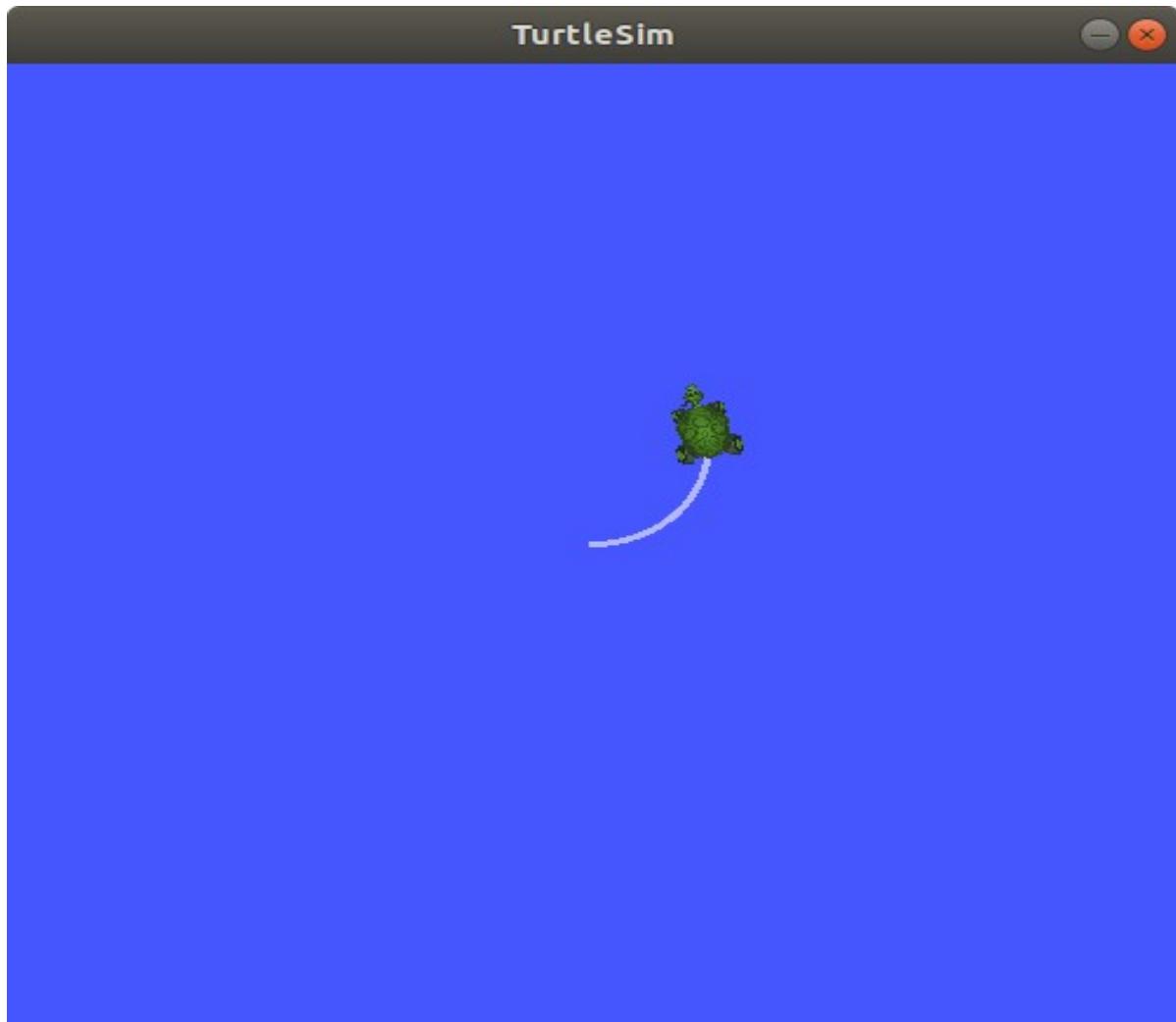


그림 34 turtlesim 실행 모습

2.4. ROS2 통합 개발환경(IDE) 구축

ROS 를 패키지를 생성하고 구동하기 위해 우분투에서 제공해주는 nano, vi, gedit 등 다양한 편집기가 있지만 보다 효과적인 개발을 위해서는 통합 개발환경 Tool을 사용 하는 것을 추천한다. 다음의 Tool 들은 다양한 편집 및 시뮬레이션 기능을 제공해줌으로 본인이 편한 개발환경 내에서 개발을 진행하는 것을 추천한다.

ROS 2 개발 환경에서 IDE는 Visual Studio Code (이하 VSCode)를 추천한다. VSCode 는 Linux, Windows, macOS 와 같이 다양한 운영체제를 지원하는 크로스 플랫폼 에디터이고 다양한 프로그래밍 언어를 지원하고 있다. 그리고 32 비트 계열은 물론 amd64, arm64 를 지원하고 있어서 ARM 계열의 임베디드 보드에서도 사용 가능하다. 더불어 텍스트 에디터 형태로 매우 가벼우면서 IntelliSense 와 Git 기능과 같은 다양한 기능을 확장(extension) 기능으로 제공하고 있다.

로그래밍 개발에는 프로그래밍 언어별 소스 코드의 구문 강조 및 자동 완성, 점프 기능 등이 매우 중요하기에 이를 위한 IntelliSense 를 설정하는 방법과 사용하는 방법에 대해 알아보고 추가로 디버깅 설정 및 사용법을 설명할 것이다. VSCode 환경에서의 디버깅 툴로 C++의 경우 GDB, Python 의 경우 debugpy 를 사용하고 있다.

Visual Studio Code 설치

VSCode 설치는 아래의 링크에서 .deb 데이안 설치 파일을 받아 설치하면 된다.

<https://code.visualstudio.com/Download>

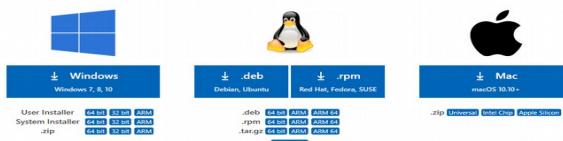


그림 35 Visual studio code 다운로드 화면

VS Code 실행

\$ code

확장 설치

좌측의 Extentions 으로 이동하여 한국어 언어팩 설치 후 재시작 버튼을 누른다.

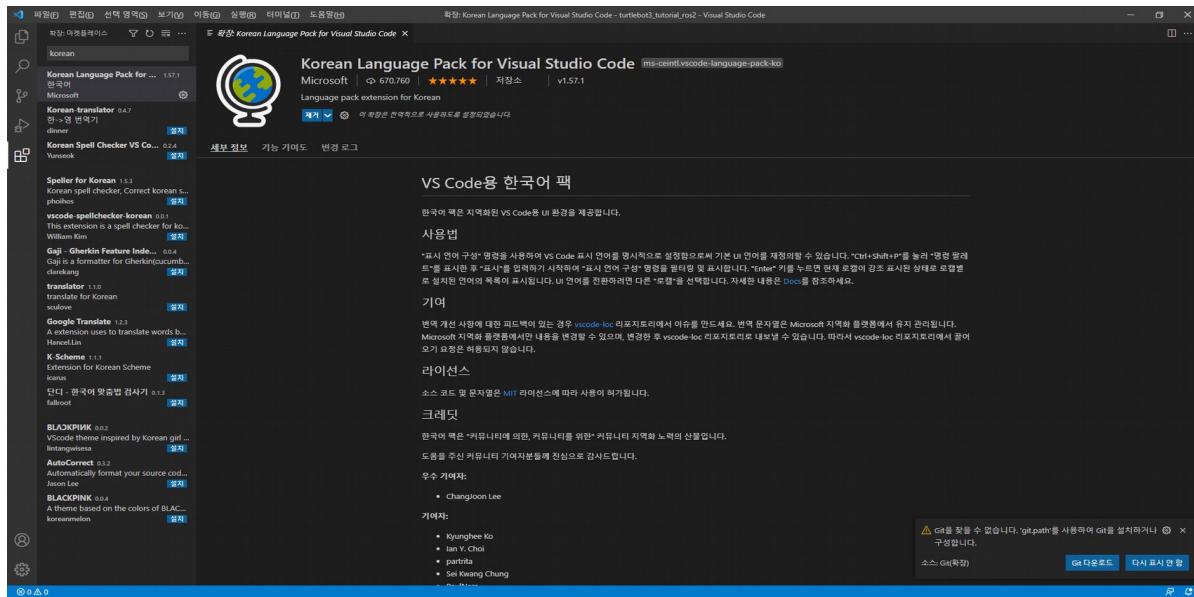


그림 36 VS Code 한국어 설치

다음으로 C/C++/Python Extentions 를 설치한다.

이름	코드명	설명
C/C++	ms-vscode.cpptools	C/C++ IntelliSense, 디버깅 및 코드 검색
CMake	twxs.cmake	CMake 언어 지원
CMake Tools	ms-vscode.cmake-tools	CMake 언어 지원 및 다양한 툴
Python	ms-python.python	린팅, 디버깅, Intellisense, 코드 서식 지정, 리팩토링, 단위 테스트 등

그림 37 C/C++/Python 확장

다음으로는 ROS 관련 확장 프로그램을 설치한다.

이름	코드명	설명
ROS	ms-iot.vscode-ros	ROS 개발 지원
URDF	smilerobotics.urdf	URDF/xacro 지원
Colcon Tasks	deitry.colcon-helper	Colcon 명령어를 위한 VSCode Task

그림 38 ROS 관련 확장 패키지

C/C++ properties 설정

C/C++ 관련 설정이다. 현재 지정한 작업 공간 (여기서는 ~/colcon_ws)의 운영체제는 무엇인지, include 폴더의 경로는 어떻게 되는지, C/C++ 규칙은 어떤 표준을 기준을 사용할지의 여부, 컴파일의 경로, intelliSense 모드 등을 설정하게 된다.

```
{  
  "configurations": [  
    {  
      "name": "Linux",  
      "includePath": [  
        "${default}",  
        "${workspaceFolder}/**",  
        "/opt/ros/foxy/include/**"  
      ],  
      "defines": [],  
      "compilerPath": "/usr/bin/g++",  
      "cStandard": "c99",  
      "cppStandard": "c++14",  
      "intelliSenseMode": "linux-gcc-x64"  
    }  
  ],  
  "version": 4  
}
```

Task 설정

VSCode 에서는 외부 프로그램을 Command Line Interface (CLI) 을 통해 연동하게 하는 기능이 있는데 이를 Task 라고 한다. 단순한 텍스트 에디터 기능이 기본인 VSCode 가 다양한 기능을 수행하고 쉽게 기능 확장을 사용할 수 있게 된 것도 이 Task 기능이 있었기 때문이다.

아래의 내용은 ROS 2 에서 빌드할 때 사용되는 colcon 과 관련한 build, test, clean 작업을 Task 로 만들었다. 이를 통해 VScode 에서 `Ctrl + Shift + b` 로 빌드할 수 있고 아래와 같이 기본 설정 이외의 Task 도 실행할 수 있다.

```
{  
  "version": "2.0.0",  
  "tasks": [  
    {  
      "label": "colcon: build",  
      "type": "shell",  
      "command": "colcon build --cmake-args '-DCMAKE_BUILD_TYPE=Debug'",  
      "problemMatcher": [],  
      "group": {  
        "kind": "build",  
        "isDefault": true  
      }  
    },  
    {  
      "label": "colcon: test",  
      "type": "shell",  
      "command": "colcon test && colcon test-result"  
    },  
    {  
      "label": "colcon: clean",  
      "type": "shell",  
      "command": "rm -rf build install log"  
    }  
  ]  
}
```

```
    }  
]  
}
```

3. 패키지의 설치와 노드 실행

3.1. Turtlesim 소개

Turtlesim은 ROS 2를 배우기 위한 경량 시뮬레이터이다. 나중에 실제 로봇이나 로봇 시뮬레이션을 통해 무엇을 할 것인지에 대한 아이디어를 제공하기 위해 가장 기본적인 수준에서 ROS 2가 무엇을 하는지 보여줄 수 있다.

`turtlesim`은 ROS 개발 초창기인 2008년에 ROS 창시자 Morgan Quigley 박사에 의해 시작된 ROS Tutorials 작업에 그 당시 ROS 툴 개발을 맡고 있던 Josh Faust에 의해 2009년에 `turtlesim`이라는 이름으로 첫 소개되었다. `turtlesim`은 해당 소스 코드의 리포지토리 같이 `ros_tutorials`라는 이름으로 쓰였고 그 하위에 속하는 패키지인 것처럼 ROS를 처음 접하는 유저들에게 튜토리얼로 제공하기 위해 ROS 패키지로 제작되었다. ROS의 학습용 패키지인 만큼 패키지, 노드, 토픽, 서비스, 액션, 파라미터에 대한 기본적인 학습 및 이 패키지와 함께 CLI 툴, rqt 툴 연동도 체험해볼 수 있다. 이 패키지는 2009년 이후 지속적으로 업데이트 되어오며 다양한 변천사를 거쳤는데 ROS 1과 2에 걸쳐서 아이콘이 배포판 이미지로 특화되어 지금의 ROS 2 Foxy 까지 지속적으로 나오고 있다.

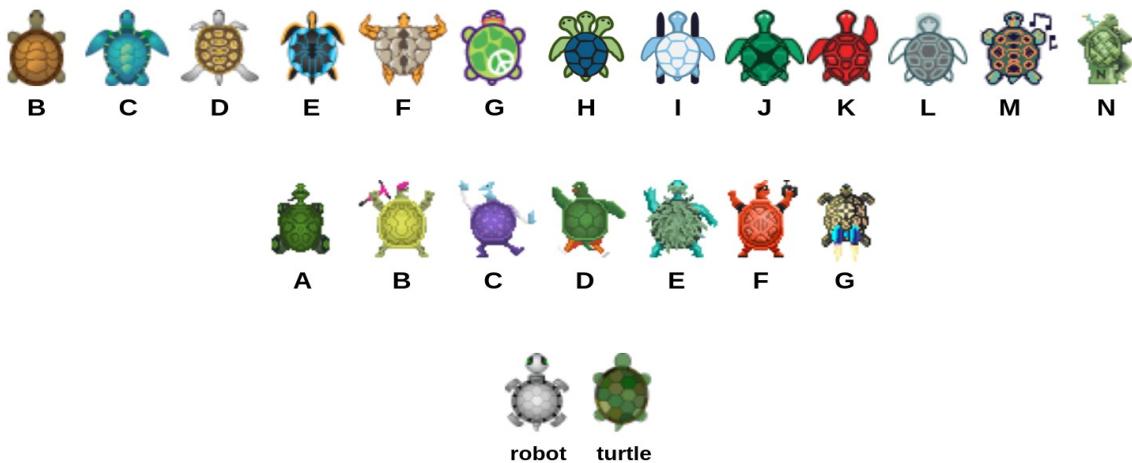


그림 39 turtlesim에 등록된 각 배포판별 커서 이미지

이 `turtlesim` 패키지의 이름에는 거북이(turtle)가 사용되었다. 그리고 `turtlesim` 패키지의 `turtlesim_node` 노드를 실행하면 나오는 거북이 아이콘에서도 모두 거북이(turtle)가 사용되었다. 이 이외에도 Open Robotics의 ROS라는 트레이드 마크 로고의 `Nine Dots`이 거북이 등껍질을 상징하고, ROS 배포판들의 포스터에서는 거북이를 빼놓을 수 없고, ROSCon의 포스터에서도 거북이가

자주 등장한다. 그리고 ROS 커뮤니티의 공식 모바일 로봇 플랫폼으로 널리 사용중인 TurtleBot이라는 이름도 `turtle`이 포함될 정도로 ROS 커뮤니티에서 거북이는 마스코트와 같은 역할이 되었다.

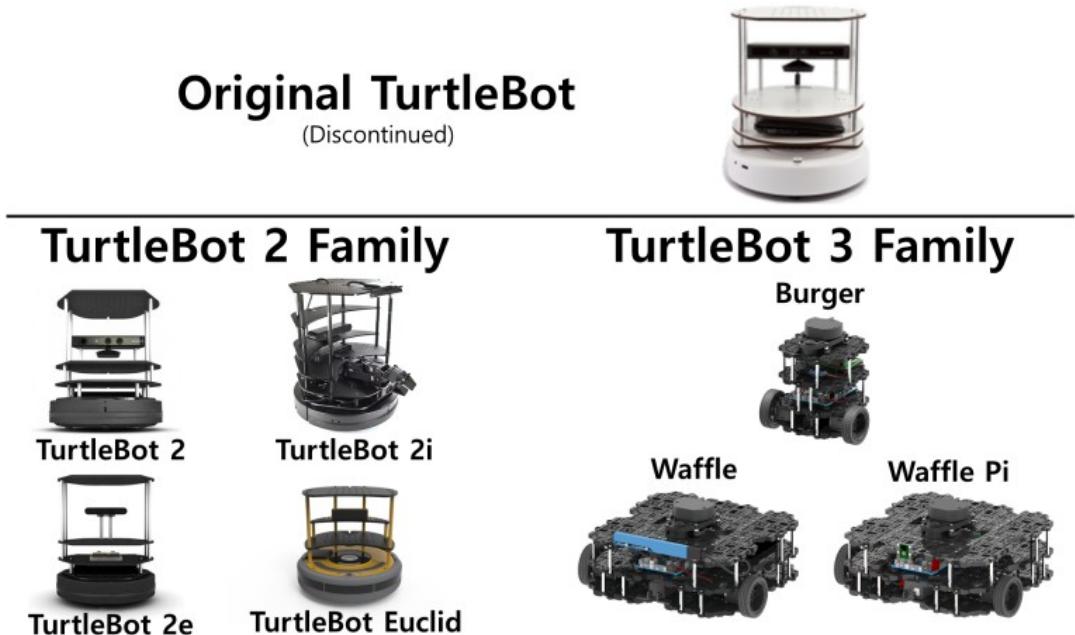


그림 40 ROS 의 공식 모바일 로봇 플랫폼 turtlebot 시리즈

3.2. Turtlesim 패키지와 노드

앞선 강좌에서 설명했던 것과 같이 ROS에서는 프로그램의 재사용성을 극대화하기 위하여 최소 단위의 실행 가능한 프로세서라고 정의하는 노드(node) 단위의 프로그램을 작성하게 된다. 이는 하나의 실행 가능한 프로그램으로 생각하면 된다. 그리고 하나 이상의 노드 또는 노드 실행을 위한 정보 등을 묶어 놓은 것을 패키지(package)라고 하며, 패키지의 묶음을 메타패키지(metapackage)라 하여 따로 분리한다.

만약 자신의 개발 환경에 어떠한 패키지가 있는지 알아보기 위해서는 아래와 같은 ROS 2의 CLI(Command Line Interface) 툴을 사용하면 된다. 그 중 `pkg` 명령어를 터미널에서 실행하면 이미 설치된 패키지 및 본인이 직접 작성한 패키지까지 포함하여 수백개의 패키지의 리스트를 확인할 수 있을 것이다. `turtlesim`을 설치하였다면 이 리스트에도 포함되어 있을 것이다.

터미널에서 아래와 같이 `ros2 pkg executables <패키지명>`을 사용하면 `turtlesim` 패키지에 포함되어 있는 다수의 노드를 확인할 수 있다. 총 4 가지로 `draw_square`, `mimic`, `turtle_teleop_key`, `turtlesim_node` 노드이다. 각 노드의 역할은 다음과 같다.

```
$ ros2 pkg executables turtlesim  
turtlesim draw_square
```

```
turtlesim mimic  
turtlesim turtle_teleop_key  
turtlesim turtlesim_node
```

- `draw_square`: 사각형 모양으로 turtle 을 움직이게하는 노드
- `mimic`: 유저가 지정한 토픽으로 동일 움직임의 `turtlesim_node`를 복수개 실행시킬 수 있는 노드
- `turtle_teleop_key`: `turtlesim_node`를 움직이게 하는 속도 값을 퍼블리시하는 노드
- `turtlesim_node`: `turtle_teleop_key` 으로부터 속도 값을 토픽으로 받아 움직이게 하는 간단 2D 시뮬레이터 노드

3.3. Turtlesim 패키지의 노드 실행

4 개의 노드 중에 가장 널리 사용하는 turtlesim_node 노드와 turtle_teleop_key 노드를 ROS 2 의 CLI(Command Line Interface) 툴의 `run` 명령어를 이용하여 실행시켜보자. turtlesim_node 노드를 실행하면 파란색 창에 거북이 한마리가 보일 것이다. 그 뒤 다른 터미널에서 turtle_teleop_key 노드를 실행시키면 화살표키로 turtlesim_node 노드의 거북이를 움질일 수 있게 된다. 더 많은 기능들이 있는데 이는 토픽, 서비스, 액션, 파라미터 강좌에서 더 자세히 다루기로 하자.

```
$ ros2 run turtlesim turtlesim_node  
$ ros2 run turtlesim turtle_teleop_key
```

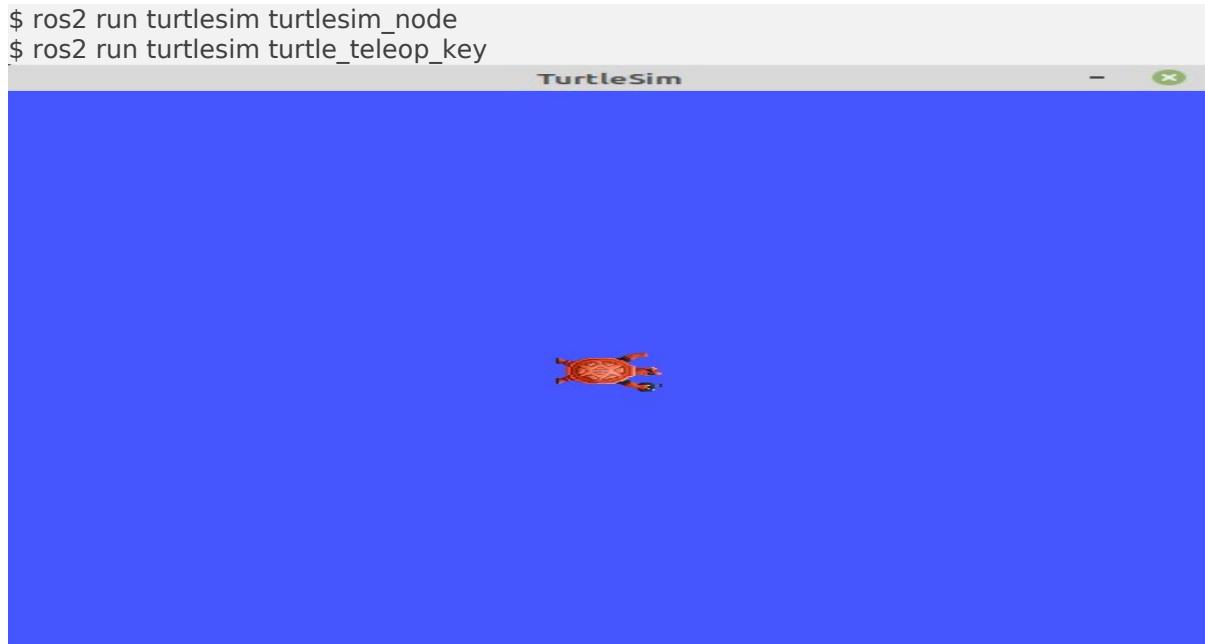


그림 41 turtlesim_node 의 실행화면

3.4. 노드 토픽, 서비스, 액션의 조회

여기서 중요한 것은 두 노드간의 동작이 단순히 키보드 값을 전달하여 움직이는게 아니라 놀려진 키보드의 키값에 해당되는 병진 속도(linear velocity)와 회전 속도(angular velocity)를 geometry_msgs 패키지의 Twist 메시지 형태로 보내고 받는다는 것이다. 참고로 turtle_teleop_key 노드를 실행시킨 후 거북이를 움직일 수 없다는 분들이 있는데 이는 해당 터미널 창을 클릭하고 활성화 시킨상태에서 화살표키를 입력값으로 받아들이기 때문이다. 주의하자.

이번에는 현재 어떠한 노드들이 실행되어 있는지, 어떻나 토픽들이 있는지, 어떠한 서비스와 액션이 있는지 알아보자. 이를 위해서는 각 명령어 `node list`, `topic list`, `service list`, `action list`를 터미널창에서 실행시키면 된다. 아래의 예제는 `turtlesim_node`와 `turtle_teleop_key`가 실행되어 있을 때의 결과 값이다.

```
$ ros2 node list
/turtlesim
/teleop_turtle

$ ros2 topic list
/parameter_events
/rosout
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose

$ ros2 service list
/clear
/kill
/reset
/spawn
/teleop_turtle/describe_parameters
/teleop_turtle/get_parameter_types
/teleop_turtle/get_parameters
/teleop_turtle/list_parameters
/teleop_turtle/set_parameters
/teleop_turtle/set_parameters_atomically
/turtle1/set_pen
/turtle1/teleport_absolute
/turtle1/teleport_relative
/turtlesim/describe_parameters
/turtlesim/get_parameter_types
/turtlesim/get_parameters
/turtlesim/list_parameters
/turtlesim/set_parameters
/turtlesim/set_parameters_atomically

$ ros2 action list
/turtle1/rotate_absolute
```

3.5. RQT_GRAPH 로 보는 노드와 토픽의 그래프 뷰

이번에는 ROS 2 CLI 툴이 아닌 ROS 2 GUI 툴을 써보자. 하기와 같이 터미널 창에서 `rqt_graph` 를 실행시키면 각 노드와 토픽, 액션이 표시된다. 이 툴을 통해 현재 개발 환경에서의 모든 노드와 토픽, 액션을 그래프 뷰로 확인할 수 있다. 그림 9 에서 보이는 동그라미가 노드이고, 네모는 토픽 또는 액션이며 화살표는 메시지의 방향을 의미한다. 여기서 서비스가 빠져있는데 서비스는 필요시 순간적으로 사용되는 형식이라 표시가 안되며, 액션은 토픽과 유사한 Pub, Sub 통신 방식의 응용이기에 표시되고 있다.

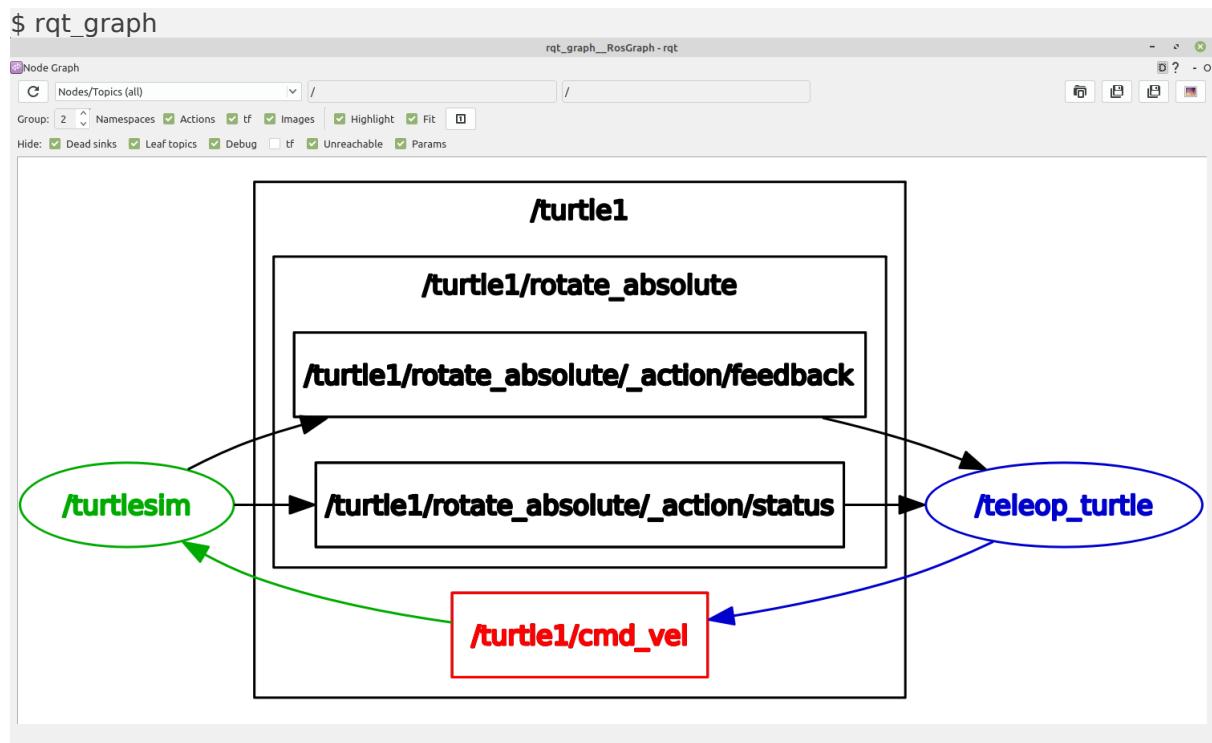


그림 42 rqt_graph 실행 화면

4. ROS2 노드와 데이터 통신

4.1. 노드(Node)와 메시지 통신(Message Communication)

노드(node)는 최소 단위의 실행 가능한 프로세스를 가리키는 용어로서 하나의 실행 가능한 프로그램을 이야기하는 것으로 ROS에서는 최소한의 실행 단위로 프로그램을 나누어 작업하게 된다.

예를 들어 노드는 원본 영상을 출력하는 카메라 드라이버, 원본 영상에 필터링하는 필터 노드, 필터링 된 영상에서 특징점을 추출하는 노드, 특징점을 이용해 물건을 검출하는 노드, 검출된 물건의 위치로 로봇의 경로를 만들어내는 노드, 모터 드라이버 노드, 경로대로 이동하는 노드 등 각 노드의 역할을 목적에 맞추어 세분화시켜 각 노드들 간의 의존성을 줄여 독립성을 높여 다른 목적의 작업을 할 때 일부 노드를 재사용할 수 있도록 하는 것이다.

이에 수많은 노드들이 연동되는 ROS 시스템을 위해서는 노드와 노드 사이에 입력과 출력 데이터를 서로 주고받게 설계해야만 한다. 여기서 주고받는 데이터를 ROS에서는 메시지(message)라고 하고 주고받는 방식을 메시지 통신(message communication)이라고 한다. 여기서 데이터에 해당되는 메시지(message)는 integer, floating point, boolean, string 와 같은 변수 형태이며 메시지 안에 메시지를 품고 있는 간단한 데이터 구조 및 메시지들의 배열과 같은 구조도 사용할 수 있다. 그리고 메시지를 주고받는 통신 방법에 따라 토픽(topic), 서비스(service), 액션(action), 파라미터(parameter)로 나누게 된다.

노드(node)는 아래 그림처럼 Node A, Node B, Node C라는 노드가 있을 때 각각의 노드들은 서로 유기적으로 Message로 연결되어 사용된다. 지금은 단순히 3개의 노드만 표시하였지만 수행하고자 하는 태스크가 많아질수록 메시지로 연결되는 노드가 늘어나며 시스템이 확장할 수 있게된다

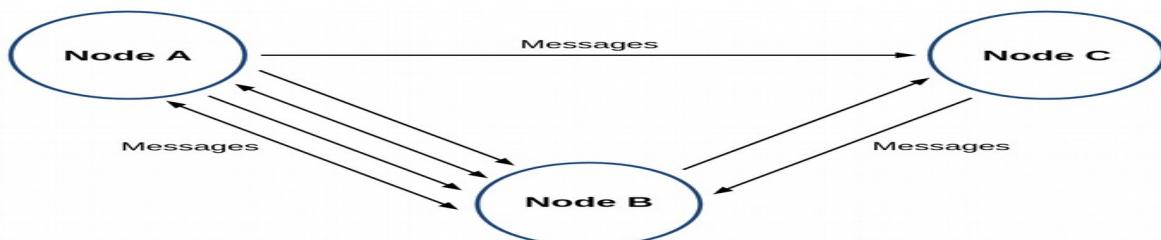


그림 43 노드들 간의 메시지 통신

4.2. 메시지 타입

4.2.1. 토픽

토픽(topic)은 그림 44의 `Node A - Node B`, `Node A - Node C`처럼 비동기식 단방향 메시지 송수신 방식으로 msg 메시지 형태의 메시지를 발간하는 Publisher 와 메시지를 구독하는 Subscriber 간의 통신이라고 볼 수 있다. 이는 1:N, N:1, N:N 통신도 가능하며 ROS 메시지 통신에서 가장 널리 사용되는 통신 방법이다.

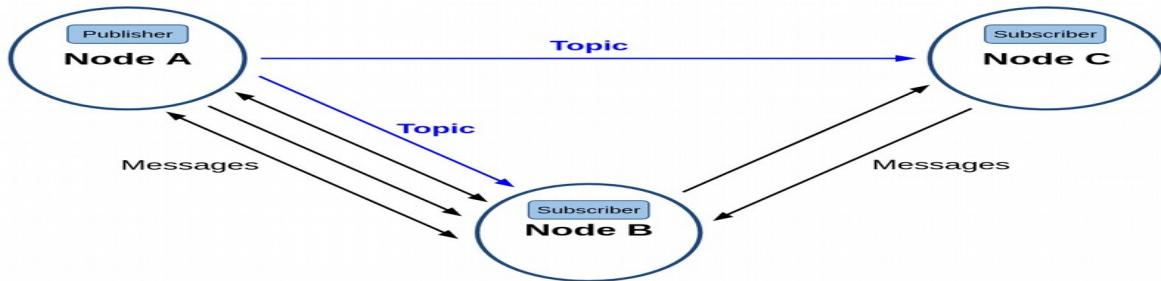


그림 44 토픽(Topic)

4.2.2. 서비스(Service)

서비스(Service)는 그림 45의 `Node B - Node C`처럼 동기식 양방향 메시지 송수신 방식으로 서비스의 요청(Request)을 하는 쪽을 Service client 라고 하며 서비스의 응답(Response)을 하는 쪽을 Service server 라고 한다. 결국 서비스는 특정 요청을 하는 클라이언트 단과 요청받은 일을 수행 후에 결괏값을 전달하는 서버 단과의 통신이라고 볼 수 있다. 서비스 요청 및 응답(Request/Response) 또한 위에서 언급한 msg 메시지의 변형으로 srv 메시지라고 한다.

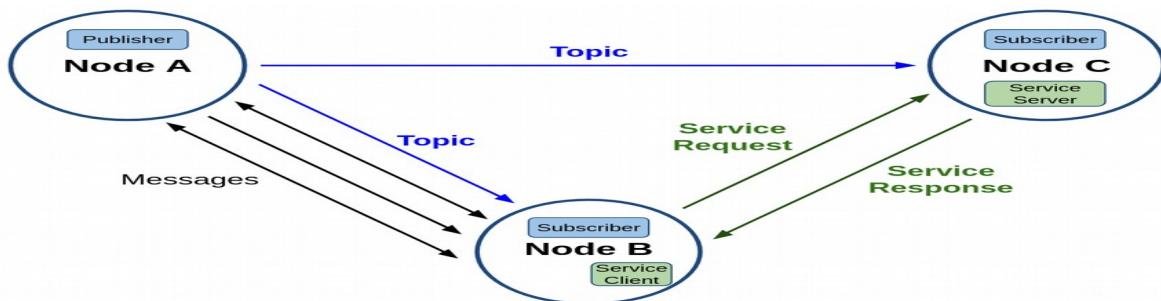


그림 45 서비스(Service)

4.2.3. 액션(Action)

액션(Action)은 그림 46의 `Node A - Node B`처럼 비동기식+동기식 양방향 메시지 송수신 방식으로 액션 목표 Goal을 지정하는 Action client과 액션 목표를 받아 특정 태스크를 수행하면서 중간 결괏값에 해당되는 액션 피드백(Feedback)과 최종 결괏값에 해당되는 액션 결과(Result)를 전송하는 Action server 간의 통신이라고 볼 수 있다. 액션의 구현 방식을 더 자세히 살펴보면 그림 47과 같이 토픽(topic)과 서비스(service)의 혼합이라고 볼 수 있는데 액션 목표 및 액션 결과를 전달하는 방식은 서비스와 같으며 액션 피드백은 토픽과 같은 메시지 전송 방식이다. 액션 목표/피드백/결과 (Goal/Feedback/Result) 메시지 또한 위에서 언급한 msg 메시지의 변형으로 action 메시지라고 한다.

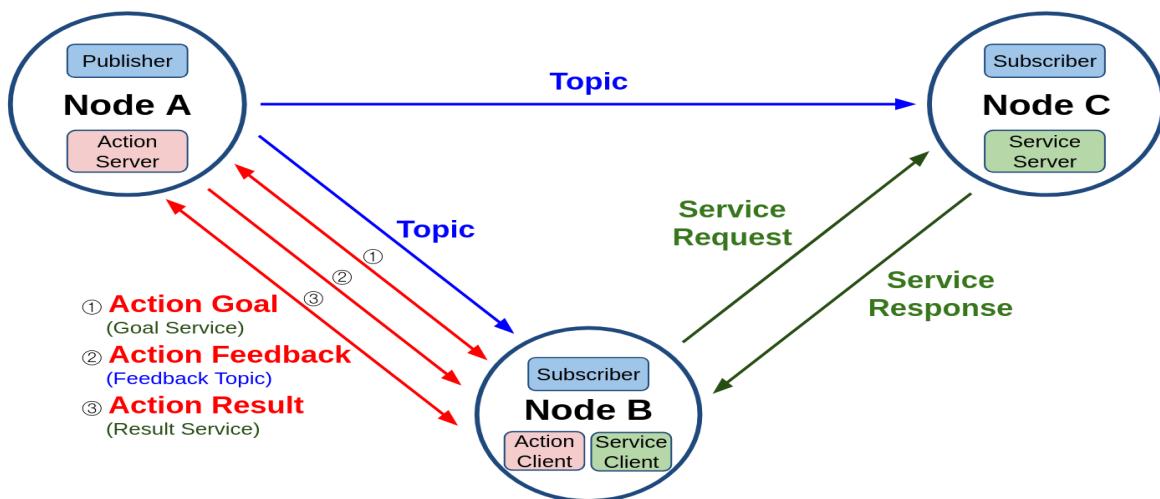


그림 46 액션(Action)

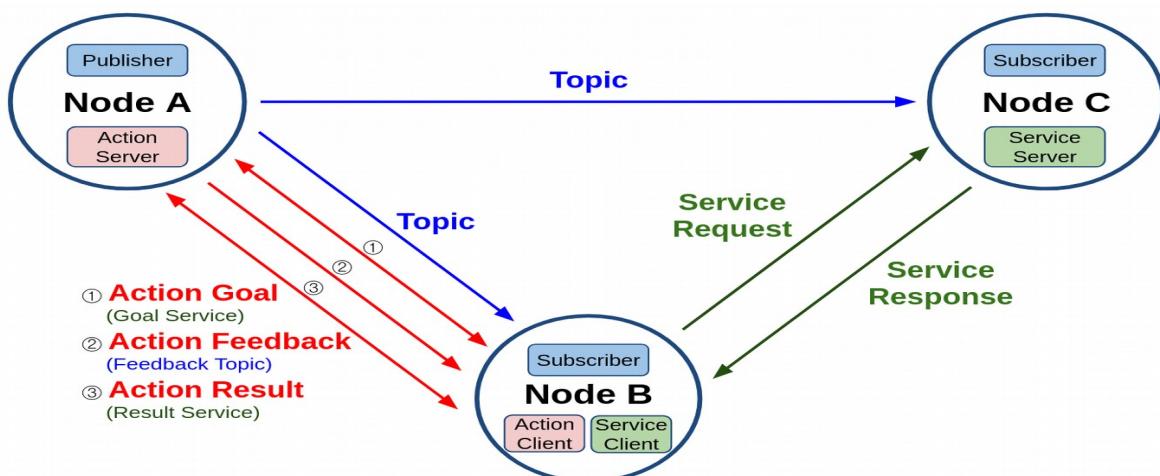


그림 47 토픽과 서비스로 구성된 액션

4.2.4. 파라미터(Parameter)

파라미터(Parameter)는 그림 48 의 각 노드에 파라미터 관련 Parameter server를 실행시켜 외부의 Parameter client 간의 통신으로 파라미터를 변경하는 것으로 서비스와 동일하다고 볼 수 있다. 단 노드 내 매개변수 또는 글로벌 매개변수를 서비스 메시지 통신 방법을 사용하여 노드 내부 또는 외부에서 쉽게 지정(Set)하거나 변경할 수 있고, 쉽게 가져(Get)와서 사용할 수 있게 하는 점에서 목적이 다르다고 볼 수 있다.

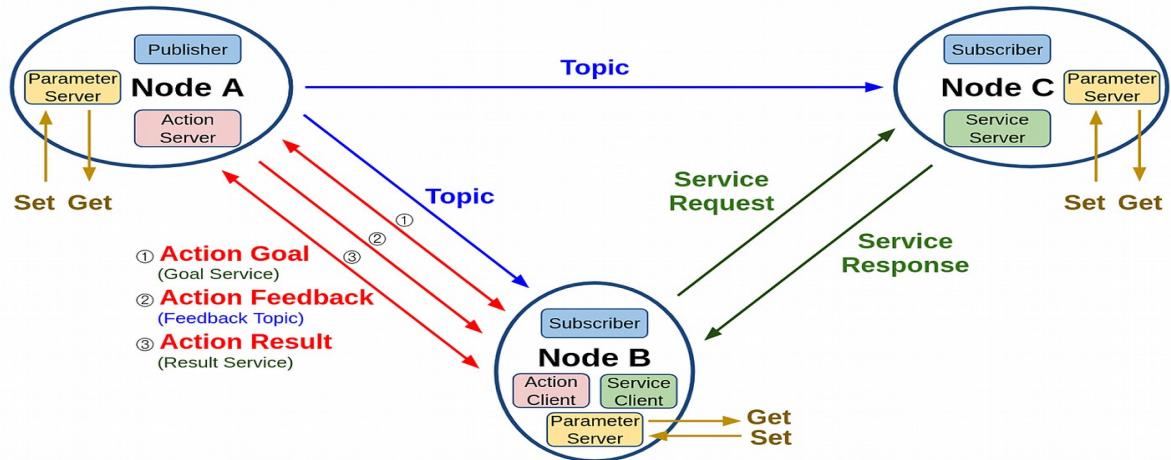


그림 48 파라미터(Parameter)

4.3. 노드의 실행

노드를 실행하기 위해서는 아래와 같이 `ros2 run` 명령어를 이용하여 특정 패키지의 특정 노드를 실행시키게 된다. 아래의 두 명령어는 각각 `turtlesim` 패키지의 `turtlesim_node`와 `turtle_teleop_key` 노드를 실행시키게 된다.

```
$ ros2 run turtlesim turtlesim_node  
$ ros2 run turtlesim turtle_teleop_key
```

기본적으로 노드를 실행시키는 방법에는 하나의 노드를 실행시키는 `ros2 run` 또는 하나 이상의 노드를 실행시키는 `ros2 launch`를 이용하는 방법 이외에도 `rqt`, `rqt_graph`, `rviz2`와 같이 지정된 실행 명령어를 실행하는 방법이 있다. 지정 실행 명령어는 주로 ROS 2의 툴을 실행시킬 때 사용된다. 다음 명령어는 노드와 노드 간의 메시지 통신을 그래프 형태로 표시하는 툴인 `rqt_graph`를 실행시키는 명령어이다.

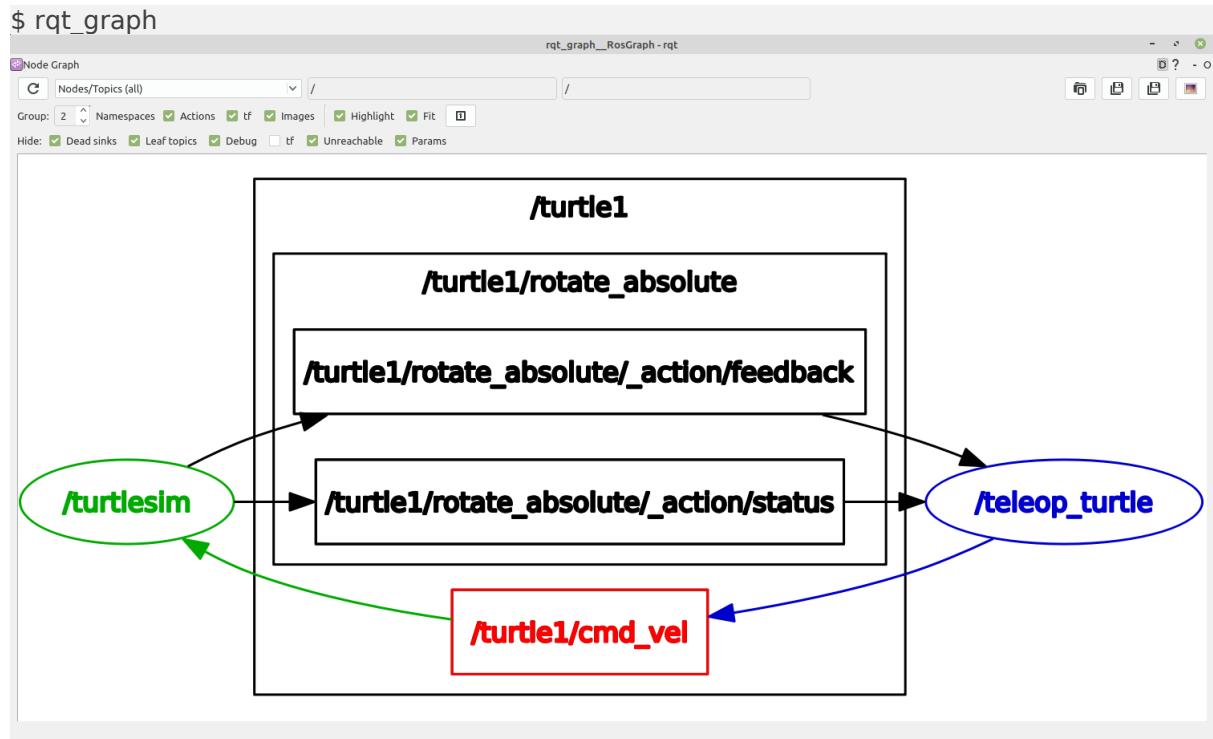


그림 49 rqt_graph 가 실행된 모습

4.4. 노드 목록

현재 개발 환경에서 동작 중인 노드의 목록을 보기를 원한다면 아래와 같이 `ros2 node list` 명령어를 통해 확인할 수 있다. 위에서 실행은 총 3 개로 rqt 툴까지 포함하여 3 개의 노드의 목록이 결괏값으로 받아 볼 수 있다. 노드 파일명과 실제 노드 이름은 다를 수 있다는 것도 알아두자. `turtlesim_node` 노드 파일은 `turtlesim`라는 노드명으로 실행되어 있고, `turtle_teleop_key` 노드 파일은 `teleop_turtle`이라는 노드명으로 실행되었다. `rqt_graph`는 rqt의 하나의 플러그인 형태로 `rqt_gui_py_node_28168`이라는 노드 이름으로 실행되었는데 rqt의 다양한 툴, 그리고 동일한 툴도 복수개 실행할 수 있도록 rqt 플러그인 류의 툴은 노드 이름에 임의의 번호가 붙게 된다는 것도 알아두자.

```
$ ros2 node list  
/rqt_gui_py_node_28168  
/teleop_turtle  
/turtlesim
```

만약 동일 노드를 복수개 실행하려고 한다면 `ros2 run turtlesim turtlesim_node` 같이 이전과 같이 그대로 실행시켜도 되지만 동일한 노드 이름으로 실행되어 된다는 점을 기억해야 한다. 동일 이름으로 실행되게 되면 시스템을 구성하는 데에 있어서 복잡하게 된다. 만약 노드명을 달리하고 싶다면 아래와 같이 노드 명을 변경하여 실행시키면 된다.

```
$ ros2 run turtlesim turtlesim_node:=new_turtle
```

이때에 실행 중인 노드 목록을 확인하는 아래 명령어로 다시 확인하면 `new_turtle`이 새롭게 추가되었음을 알 수 있다. `rqt_graph` 상에서도 그림 9와 같이 지정된 이름을 실행되었음을 확인할 수 있다. 현재에는 그림 9와 같이 노드의 이름만 변경되었을 뿐 토픽인 `turtle1/cmd_vel`은 변경되지 않았음을 알 수 있다. 그리고 `teleop_turtle` 노드를 이용하여 거북이를 움직여보면 두 개의 노드의 거북이들이 모두 동일하게 움직이는 것을 볼 수 있는데 동일한 토픽을 이용하기 때문이다. 이 토픽 또한 토픽명을 변경하거나 name_space를 통해 바꿀 수 있는데 이는 해당 내용을 설명하는 토픽 및 launch 강좌 때에 자세히 다루도록 하겠다.

```
$ ros2 node list  
/rqt_gui_py_node_29017  
/teleop_turtle  
/new_turtle  
/turtlesim
```

4.5. 노드 정보 확인하기

노드의 정보를 확인하기 위해서는 아래와 같이 `ros2 node info` 명령어에 정보를 보기 위한 노드명을 지정하면 된다. 이 정보에는 지정된 노드의 Publishers, Subscriber, Service, Action, Parameter 정보를 확인할 수 있다.

```
$ ros2 node info /turtlesim
/turtlesim
Subscribers:
/parameter_events: rcl_interfaces/msg/ParameterEvent
/turtle1/cmd_vel: geometry_msgs/msg/Twist
Publishers:
/parameter_events: rcl_interfaces/msg/ParameterEvent
/rosout: rcl_interfaces/msg/Log
/turtle1/color_sensor: turtlesim/msg/Color
/turtle1/pose: turtlesim/msg/Pose
Service Servers:
/clear: std_srvs/srv/Empty
/kill: turtlesim/srv/Kill
/reset: std_srvs/srv/Empty
/spawn: turtlesim/srv/Spawn
/turtle1/set_pen: turtlesim/srv/SetPen
/turtle1/teleport_absolute: turtlesim/srv/TeleportAbsolute
/turtle1/teleport_relative: turtlesim/srv/TeleportRelative
/turtlesim/describe_parameters: rcl_interfaces/srv/DescribeParameters
/turtlesim/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
/turtlesim/get_parameters: rcl_interfaces/srv/GetParameters
/turtlesim/list_parameters: rcl_interfaces/srv/ListParameters
/turtlesim/set_parameters: rcl_interfaces/srv/SetParameters
/turtlesim/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
Service Clients:
Action Servers:
/turtle1/rotate_absolute: turtlesim/action/RotateAbsolute
Action Clients:
```

```
$ ros2 node info /teleop_turtle
/teleop_turtle
Subscribers:
/parameter_events: rcl_interfaces/msg/ParameterEvent
Publishers:
/parameter_events: rcl_interfaces/msg/ParameterEvent
/rosout: rcl_interfaces/msg/Log
/turtle1/cmd_vel: geometry_msgs/msg/Twist
Service Servers:
/teleop_turtle/describe_parameters: rcl_interfaces/srv/DescribeParameters
/teleop_turtle/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
/teleop_turtle/get_parameters: rcl_interfaces/srv/GetParameters
/teleop_turtle/list_parameters: rcl_interfaces/srv/ListParameters
/teleop_turtle/set_parameters: rcl_interfaces/srv/SetParameters
/teleop_turtle/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
Service Clients:
Action Servers:
Action Clients:
```

```
/turtle1/rotate_absolute: turtlesim/action/RotateAbsolute
```

5. ROS2 Message Type

5.1. 토픽(Topic)

5.1.1. 토픽이란?

토픽(topic)은 그림 50의 `Node A - Node B`처럼 비동기식 단방향 메시지 송수신 방식으로 msg 메시지 형태의 메시지를 발행하는 `Publisher`와 메시지를 구독하는 `Subscriber` 간의 통신이라고 볼 수 있다. 이는 1:1 통신을 기본으로 하지만 그림 51의 `Node A - Node B`, `Node A - Node C`와 같이 하나의 토픽(예: Topic C)을 송수신하는 1:N도 가능하고 그 구성 방식에 따라 N:1, N:N 통신도 가능하며 ROS 메시지 통신에서 가장 널리 사용되는 통신 방법이다.

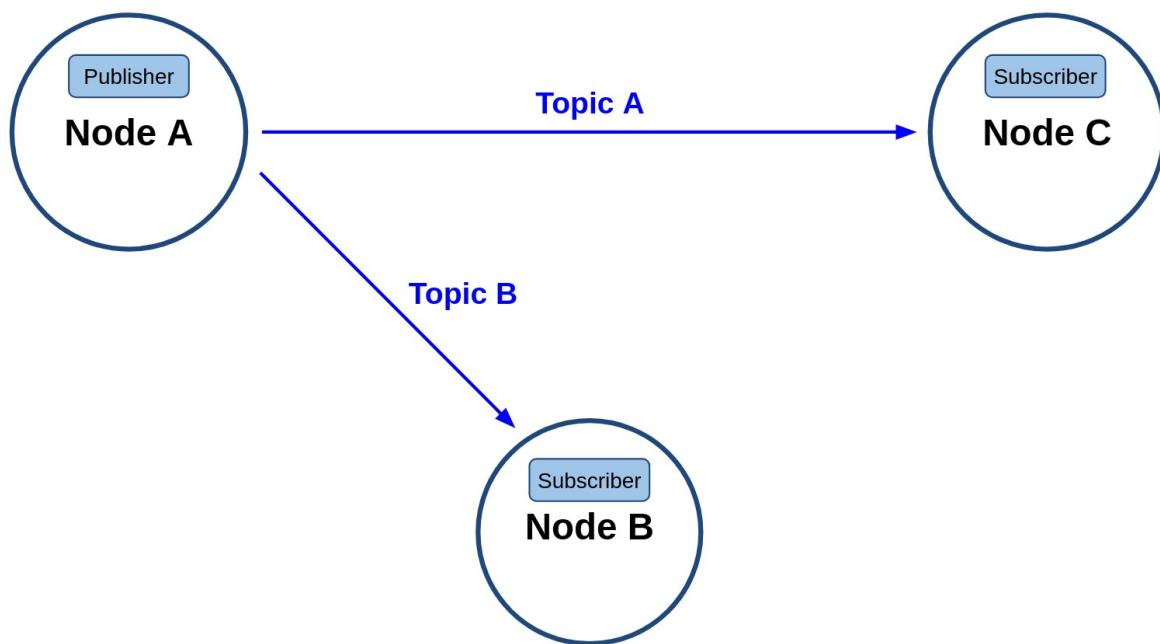


그림 50 퍼블리셔와 서브스크라이버

그리고 그림 51의 `Node A`처럼 하나의 이상의 토픽을 발행할 수 있을 뿐만이 아니라 `Publisher` 기능과 동시에 토픽(예: Topic D)을 구독하는 `Subscriber` 역할도 동시에 수행할 수 있다. 원한다면 자신이 발행한 토픽을 셀프 구독할 수 있게 구성할 수도 있다. 이처럼 토픽 기능은 목적에 따라 다양한 방법으로 사용할 수 있는데 이러한 유연성으로 다양한 곳에 사용중에 있다. 경험상 ROS 프로그래밍시에 70% 이상이 토픽으로 사용될 정도로 통신 방식 중에 가장 기본이 되며 가장 널리쓰이는 방법이다. 기본 특징으로 비동기성과 연속성을 가지기에 센서 값 전송 및 항시 정보를 주고 받아야하는 부분에 주로 사용된다.

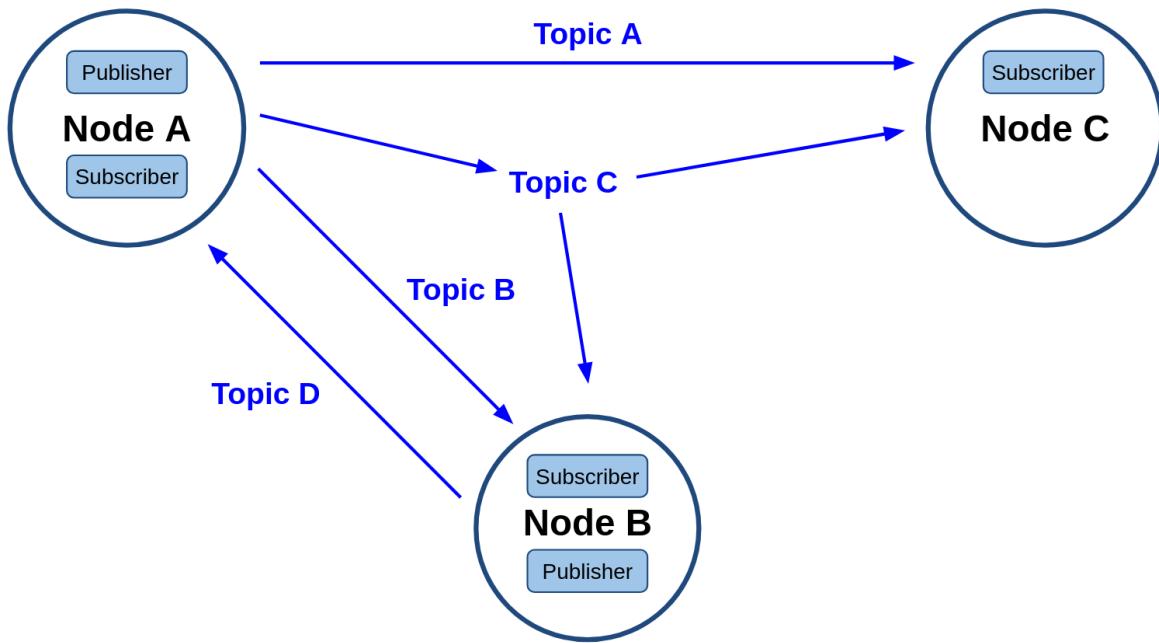


그림 51 다자간 Topic 통신

5.1.2. 토픽 목록 확인

좀 더 간단하게 메시지들을 확인하고 싶다면 아래 명령어처럼 `ros2 topic list -t`를 이용하면 된다. 이전의 명령어가 turtlesim 노드만의 정보를 확인하였다면 `ros2 topic list -t` 명령어는 현재 개발 환경에서 동작 중인 모든 노드들의 토픽 정보를 볼 수 있는 것으로 지금은 turtlesim 노드만이 실행된 상태이기에 turtlesim 노드가 발행, 구독하는 메시지만 표시되고 있다. 참고로 `-t` 옵션은 부가적인 것으로 각 메시지의 형태(type)를 함께 표시해준다.

```
$ ros2 topic list -t
/parameter_events [rcl_interfaces/msg/ParameterEvent]
/rosout [rcl_interfaces/msg/Log]
/turtle1/cmd_vel [geometry_msgs/msg/Twist]
/turtle1/color_sensor [turtlesim/msg/Color]
/turtle1/pose [turtlesim/msg/Pose]
```

현재 상태는 turtlesim 노드만 실행되었기에 토픽을 주고 받는 상황은 아니다. 아래와 같이 rqt_graph를 실행시키면 토픽을 주고 받는 어떠한 상황도 아닌 것임을 그림 52에서 확인할 수 있을 것이다.

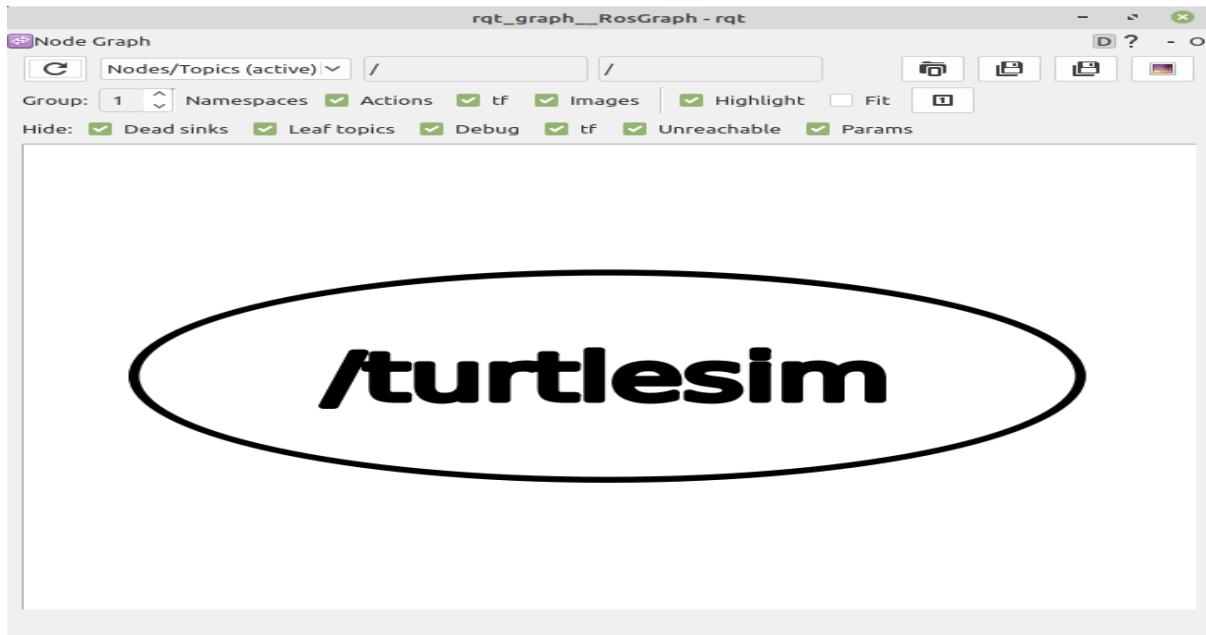


그림 52 rqt_graph 실행 모습

여기서 추가로 turtle_teleop_key (노드명: teleop_turtle, 이하 teleop_turtle 라고 표기함)를 실행해보자. 그 뒤 실행시켜둔 rqt_graph 화면에서 그림 52 와 같이 프로그램의 좌측 상단의 `Refresh ROS graph` 버튼을 클릭하면 최신 상태로 갱신되는데 그림 53 과 같이 teleop_turtle 노드에서 발행한 cmd_vel 토픽을 turtlesim 에서 구독하고 있는 모습을 볼 수 있다.

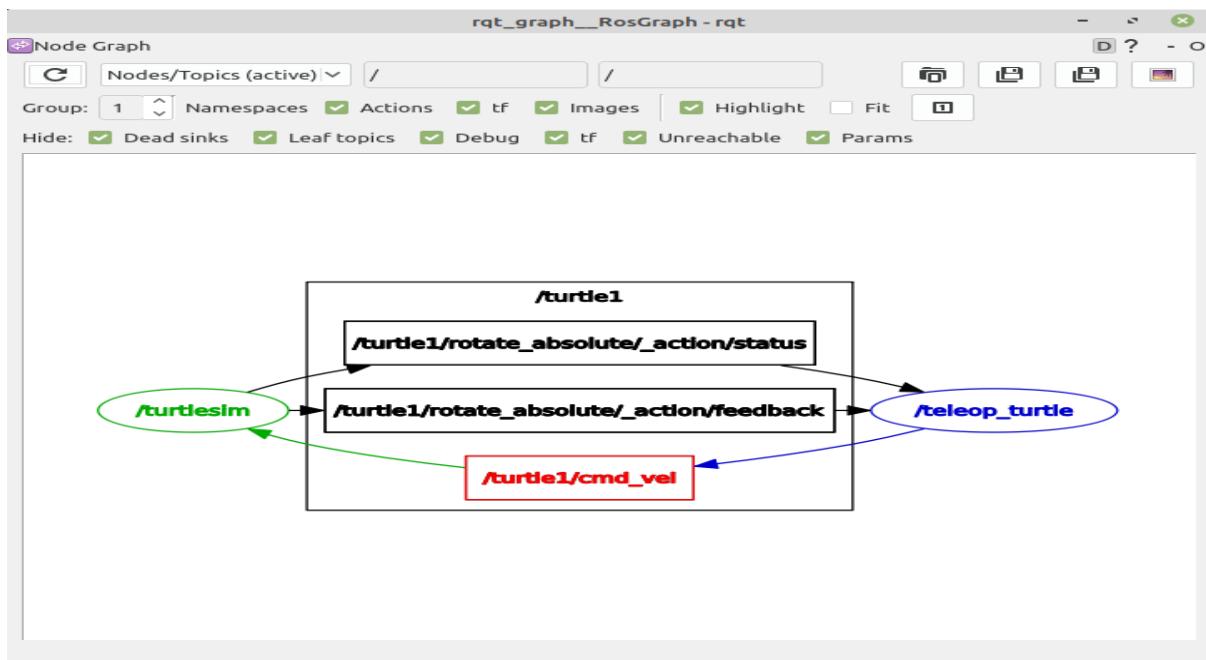


그림 53 rqt_graph 로 알아본 turtlesim_node 와 teleop_turtle 노드의 관계

여기서 궁금증이 있을 것이다. 분명 위에서는 color_sensor 및 pose 토픽도 있다고 했는데 여기서는 왜 표시가 되지 않는 것일까? 분명히 turtlesim 노드는 이 두 개의 토픽을 발행하고 있다. 하지만 teleop_turtle 를 실행하기 전에 cmd_vel 토픽이 활성화 되지 않았던 것과 마찬가지로 아직 이를 구독하는 노드가 없기 때문이다. 만약에 발행하는 노드, 구독하는 노드와 상관없이 모든 토픽을 보기 위해서는 rqt_graph 화면에서 `Dead sinks` 와 `Leaf topics` 를 해제하여 모두 보이게 하자. 그러면 그림 54 와 같이 확인할 수 있을 것이다.

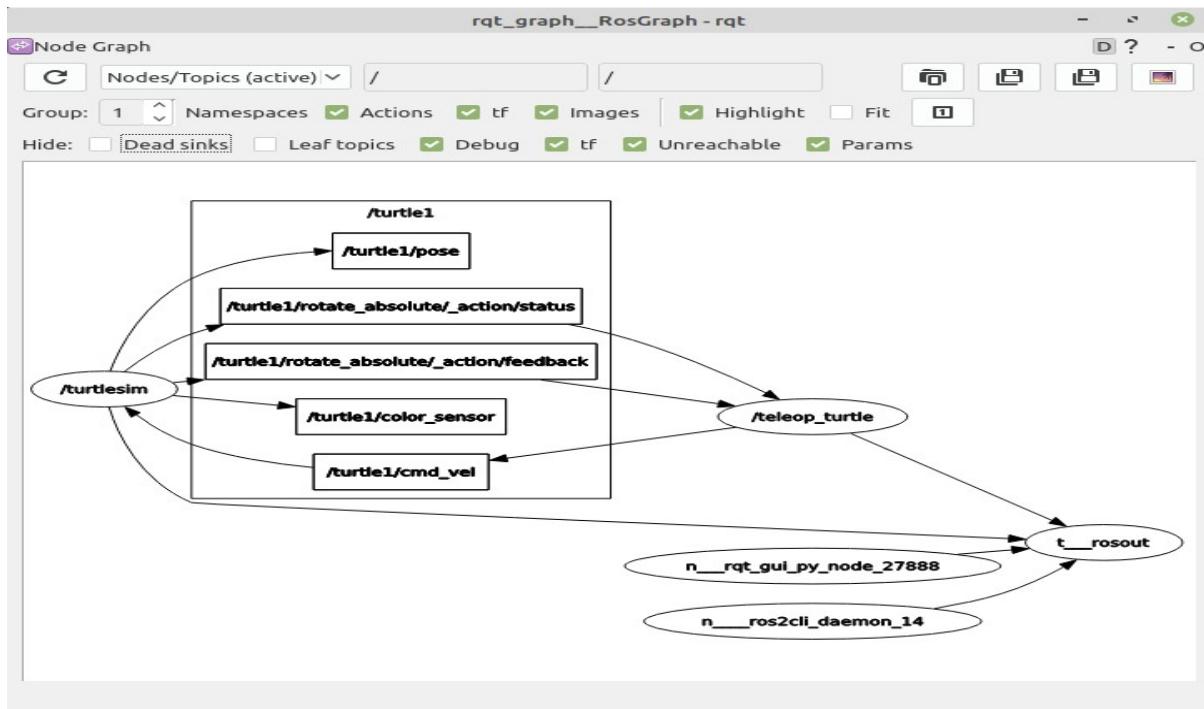


그림 54 rqt_graph 의 옵션

5.1.3. 토픽 정보 확인

rqt_graph 를 이용하여 토픽의 pub/sub 상태를 확인하는 방법이외에도 ROS 2 CLI 툴을 이용하여 하기와 같이 토픽 메시지 형태, 토픽의 발행 및 구독 정보를 확인할 수도 있다. 경험상 이 명령어는 잘 사용하지는 않는다.

```
$ ros2 topic info /turtle1/cmd_vel
Type: geometry_msgs/msg/Twist
Publisher count: 1
Subscriber count: 1
```

5.1.4. 토픽 내용 확인

이번에는 특정 토픽의 메시지 내용을 실시간으로 표시하는 `ros2 topic echo` 를 사용해보자. 다음 명령어와 같이 `/turtle1/cmd_vel` 라고 토픽을 지정하게 되면 해당 토픽의 값을 확인해볼 수 있다.

참고로 `/turtle1/cmd_vel` 토픽을 발행하는 teleop_turtle 노드를 실행한 터미널 창에서 방향 키보드 키($\leftarrow \uparrow \downarrow \rightarrow$)를 눌러 명령을 내려야지만 토픽 값을 확인할 수 있다. 아래의 결과를 보자면 `/turtle1/cmd_vel` 토픽의 linear에 x, y, z 값이 있으며, angular에 x, y, z 값이 존재한다는 것을 알 수 있다. 총 6 개의 값으로 구성되어 있으며 현재 linear.x 값으로 1.0 m/s 임을 확인할 수 있다.

```
$ ros2 topic echo /turtle1/cmd_vel
linear:
x: 1.0
y: 0.0
z: 0.0
angular:
x: 0.0
y: 0.0
z: 0.0
```

5.1.5. 토픽 대역폭 확인

이번에는 메시지의 대역폭, 즉 송수신받는 토픽 메시지의 크기를 확인해보자. 크기 확인은 아래 명령어와 같이 `ros2 topic bw`으로 지정된 토픽 메시지의 송수신되는 토픽의 초당 대역폭을 알 수 있다. teleop_turtle 노드에서 지속적으로 메시지를 보내는 상황이라면 평균 1.74KB/s 의 대역폭으로 `/turtle1/cmd_vel` 토픽이 사용되는 것을 확인할 수 있다. 이는 사용하는 메시지 형태 및 주기에 따라 달라질 수 있다.

```
$ ros2 topic bw /turtle1/cmd_vel
Subscribed to [/turtle1/cmd_vel]
average: 1.74KB/s
mean: 0.05KB min: 0.05KB max: 0.05KB window: 100
(생략)
```

5.1.6. 토픽 주기 확인

토픽의 전송 주기를 확인하려면 `ros2 topic hz` 명령어를 이용하면 된다. teleop_turtle 노드에서 지속적으로 `/turtle1/cmd_vel` 토픽을 발행한다면 아래와 같이 평균 33.2 Hz 정도가 나올 것이다. 즉 0.03 초에 한번씩 토픽을 발행하고 있다는 것이다. 이는 teleop_turtle 노드에서 얼마나 자주 `/turtle1/cmd_vel` 토픽을 발행하는지에 따라 달라질 수 있다.

```
$ ros2 topic hz /turtle1/cmd_vel
average rate: 33.212
min: 0.029s max: 0.089s std dev: 0.00126s window: 2483
(생략)
```

5.1.7. 토픽 지연 시간 확인

토픽은 RMW 및 네트워크 장비를 거치기 때문에 latency 즉 지연 시간이 반드시 존재하게 된다. 이 지연 시간을 체크하는 방식으로 유저가 직접 코드로 구현하는 방법도 있겠지만 메시지내에 header[6]라는

stamp 메시지를 사용하고 있다면 `ros2 topic delay`를 명령어를 이용하여 메시지는 발행한 시간과 구독한 시간의 차를 계산하여 지연 시간을 확인할 수 있다.

```
$ ros2 topic delay /TOPIC_NAME  
average delay: xxx.xxx  
min: xxx.xxxx max: xxx.xxxx std dev: xxx.xxxx window: 10
```

5.1.8. 토픽 발행

토픽의 발행(publish)은 ROS 프로그램에 내장하는게 기본이다. 이는 ROS 프로그래밍 시간에 다루도록 하고 여기서는 `ros2 topic pub` 명령어를 통해 간단히 토픽을 발행하는 테스트를 해보자. 이 명령어의 사용은 다음과 같다. `ros2 topic pub` 명령어에 토픽 이름, 토픽 메시지 타입, 메시지 내용을 기술하면 된다.

```
ros2 topic pub <topic_name> <msg_type> "<args>"
```

즉, 아래와 같이 기술하면 되는데 아래 명령어를 풀어 해석하자면 `--once` 옵션을 사용하여 단 한번의 발행만을 수행하도록 하였으며, 토픽 이름으로는 /turtle1/cmd_vel 을 사용하였고, 토픽 메시지 타입은 geometry_msgs/msg/Twist 을 사용하였다. 메시지 내용으로는 병진 속도 linear.x 값으로 2.0 m/s 를 넣었고, 회전 속도 angular.z 값으로 1.8 rad/s 를 입력하였다.

```
$ ros2 topic pub --once /turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 1.8}}"
```

이를 실행시키면 그림 55 와 같이 거북이가 정해진 병진 속도 및 회전 속도로 이동되었음을 확인할 수 있다.

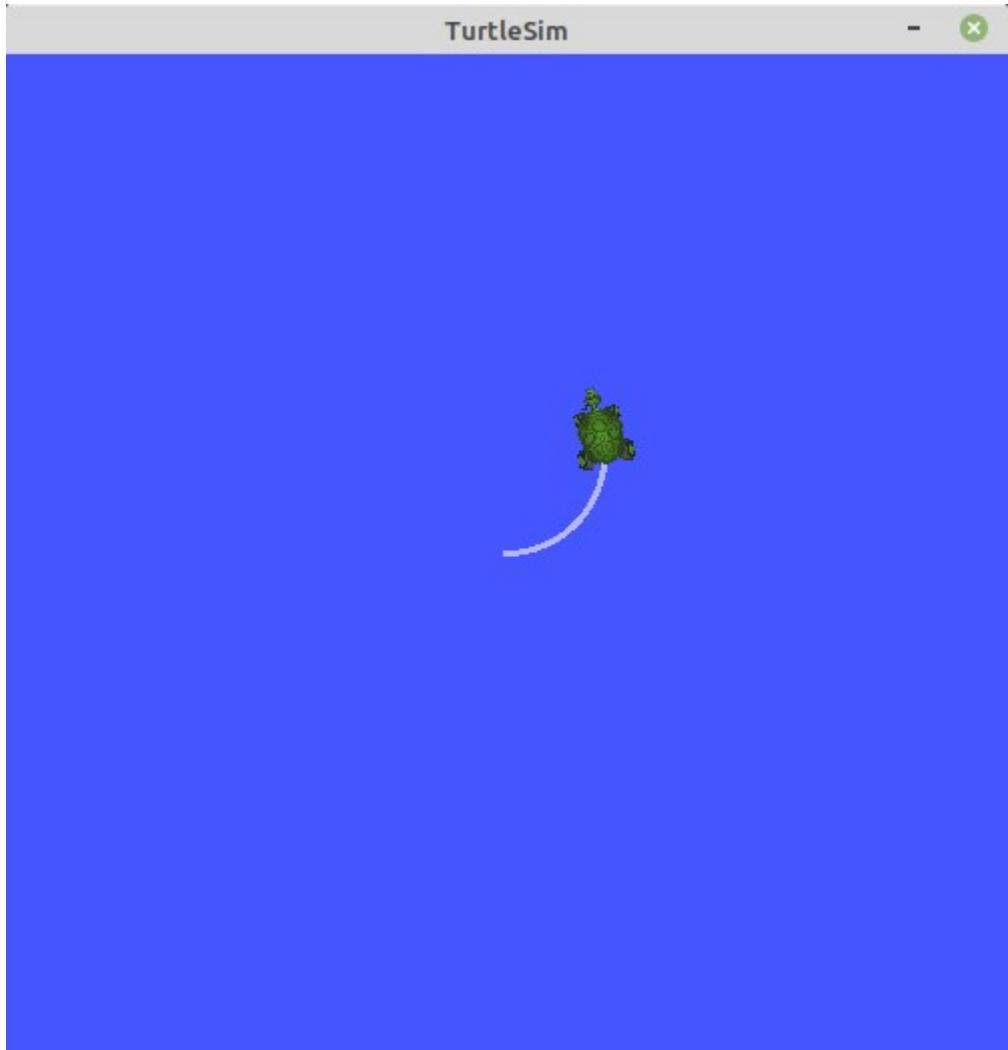


그림 55 거북이가 병진/회전 속도 토픽을 받아 이동하는 모습

지속적인 발행을 원한다면 `--once` 옵션 제거하고 대신 `--rate 1` 옵션처럼 주기 1Hz의 발행도 할 수 있다.

```
$ ros2 topic pub --rate 1 /turtle1/cmd_vel geometry_msgs/msg/Twist "{\"linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 1.8}}"
```

5.1.9. 토픽 기록 및 저장

bag 기록

ROS에는 발행하는 토픽을 파일 형태로 저장하고 필요할 때 저장된 토픽을 다시 불러와 동일 타이밍으로 재생할 수 있는 기능이 있다. 이를 `rosbag`이라고 한다. 이는 매우 유용한 ROS의 기능으로 디버깅에 큰 도움을 준다.

예를 들어 내가 SLAM 알고리즘을 개발한다면 알고리즘 개발에 집중하기 위해서는 SLAM에 필요한 라이다와 같은 센서 정보와 로봇의 위치 정보인 오도메트리와 같은 상태 정보가 필요한데 매번 로봇을

구동시켜 데이터를 취득하기도 힘들고 데이터 취득을 매번하여 테스트를 하더라도 센서 정보 및 로봇 상태 값에 따라 결괏값이 상이해져 알고리즘이 좋아진 것인지 테스트할 때 운이 좋게 센서 및 로봇 상태 값이 좋았는지 구분하기 어렵다. 이럴 때에는 알고리즘의 입력 값을 고정하고 반복하여 테스트할 수 있다면 알고리즘만의 개선 작업 및 성능 검증 테스트를 할 수 있게 된다. `rosbag`은 이러한 상황에서 원하는 토픽을 기록하고 재생하는 도구이다.

`rosbag`은 다음과 같이 bag 기록 (ros2 bag record) 명령어에 내가 기록하고자 하는 토픽 이름을 기재하면 된다. 예를 들어 `/turtle1/cmd_vel` 토픽을 기록하려면 다음 명령어와 같이 실행해주면 된다. 기록 종료는 프로그램을 종료하면 되는데 일반적인 노드 종료와 마찬가지로 해당 터미널 창에서 `Ctrl + c`를 눌러주면 된다. 기록이 종료되면 `rosbag2_2020_09_04-08_31_06`이라는 이름으로 폴더가 생성된다. 참고로 원하는 이름이 별도로 있다면 `ros2 bag record -o 이름 /turtle1/cmd_vel`과 같이 `-o` (output) 옵션을 이용하여 특정 이름을 지정해도 된다.

```
ros2 bag record <topic_name1> <topic_name2> <topic_name3>
```

```
$ ros2 bag record /turtle1/cmd_vel
[INFO]: Opened database 'rosbag2_2020_09_04-08_31_06'.
[INFO]: Listening for topics...
[INFO]: Subscribed to topic '/turtle1/cmd_vel'
```

bag 정보

저장된 `rosbag` 파일의 정보를 확인하려면 아래 예제와 같이 bag 정보 (ros2 bag info) 명령어를 이용하면 된다. 내용을 살펴보면 방금 전 우리가 기록한 이 `rosbag` 파일은 84.4 KiB 크기에 31.602s 시간 동안 기록되었고, 기록이 언제 시작되고 언제 끝났는지 타임스탬프와 취득한 토픽의 이름 메시지 형태 메시지 별 갯수와 총 갯수 등이 기록되어 있다.

```
$ ros2 bag info rosbag2_2020_09_04-08_31_06/
Files:          rosbag2_2020_09_04-08_31_06.db3
Bag size:      84.4 KiB
Storage id:    sqlite3
Duration:      31.602s
Start:         Sep 4 2020 08:31:09.952 (1599175869.952)
End:          Sep 4 2020 08:31:41.554 (1599175901.554)
Messages:       355
Topic information: Topic: /turtle1/cmd_vel | Type: geometry_msgs/msg/Twist | Count: 355 |
Serialization Format: cdr
```

bag 재생

`rosbag` 파일을 기록하고 정보를 확인해봤으니 이제는 재생을 해보자. 일단 `turtlesim` 노드를 종료한 후 다시 시작하여 초기화를 해준 후 아래의 예제처럼 `rosbag`를 재생하면 기록 시간 타이밍에 따라 토픽이 재생됨을 확인할 수 있다. 이는 위에서 설명한 `ros2 topic echo /turtle1/cmd_vel` 명령어를 이용하여 터미널 창에서 확인해도 되고, 그림 11와 같이 `turtlesim` 노드위의 움직임을 비교해도 된다.

```
$ ros2 bag play rosbag2_2020_09_04-08_31_06/
[INFO]: Opened database 'rosbag2_2020_09_04-08_31_06'.
```

5.2. 서비스(Service)

5.2.1. 서비스란?

서비스(service)는 그림 56의 `Node A - Node B`처럼 동기식 양방향 메시지 송수신 방식으로 서비스의 요청(Request)을 하는 쪽을 Service Client라고 하며 요청받은 서비스를 수행한 후 서비스의 응답(Response)을 하는 쪽을 Service Server라고 한다. 결국 서비스는 특정 요청을 하는 클라이언트 단과 요청받은 일을 수행 후에 결과값을 전달하는 서버 단과의 통신이라고 볼 수 있다. 서비스 요청 및 응답(Request/Response) 또한 위에서 언급한 msg 인터페이스의 변형으로 srv 인터페이스라고 한다.



그림 56 서비스 통신에서의 서버와 클라이언트

서비스는 그림 57과 같이 동일 서비스에 대해 복수의 클라이언트를 가질 수 있도록 설계되었다. 단, 서비스 응답은 서비스 요청이 있었던 서비스 클라이언트에 대해서만 응답을 하는 형태로 그림 2의 구성에서 예를 들자면 Node C의 Service Client가 Node B의 Service Server에게 서비스 요청을 하였다면 Node B의 Service Server는 요청받은 서비스를 수행한 후 Node C의 Service Client에게만 서비스 응답을 하게된다.

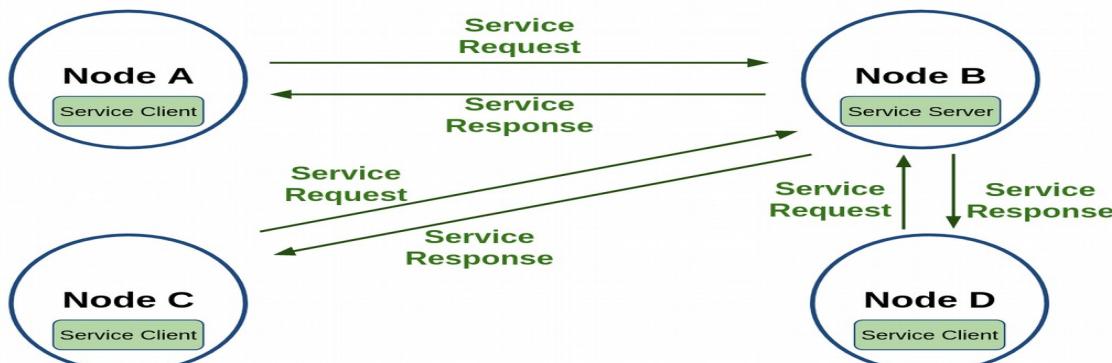


그림 57 서비스와 클라이언트의 관계

5.2.2. 서비스 목록 확인

turtlesim 패키지의 turtlesim_node (노드명: turtlesim) 노드를 이용할 것이며 아래와 같이 실행한다. 그 뒤 아래와 같이 서비스 목록 확인 (ros2 service list) 명령어로 현재 실행 중인 노드들의 서비스 목록을 확인해본다.

```
$ ros2 service list  
/clear  
/kill  
/reset  
/spawn  
/turtle1/set_pen  
/turtle1/teleport_absolute  
/turtle1/teleport_relative  
/turtlesim/describe_parameters  
/turtlesim/get_parameter_types  
/turtlesim/get_parameters  
/turtlesim/list_parameters  
/turtlesim/set_parameters  
/turtlesim/set_parameters_atomically
```

다양한 서비스들이 포함되어 있는데 parameters 가 붙어있는 서비스는 파라미터와 관련된 내용으로 모든 노드의 기본 기능으로 포함되어 있다.

5.2.3. 서비스 형태 확인

특정 서비스의 형태가 궁금하다면 서비스 형태 확인 명령어인 `ros2 service type`을 이용하면 된다. 사용 방법은 다음과 같이 명령어 뒤에 특정 서비스명을 입력하면 어떤 형태의 서비스인지 확인할 수 있다. 예를 들어 아래와 같이 clear 서비스는 std_srvs/srv/Empty 형태이고, kill 서비스는 turtlesim/srv/Kill 형태임을 확인할 수 있다.

```
$ ros2 service type /clear  
std_srvs/srv/Empty  
$ ros2 service type /kill  
turtlesim/srv/Kill  
$ ros2 service type /spawn  
turtlesim/srv/Spawn
```

참고로 서비스 형태 확인 명령어가 아니더라도 위에서 설명하였던 서비스 목록 확인 (ros2 service list) 명령어에 옵션으로 `-t`를 붙여주면 형태를 서비스 목록과 함께 볼 수 있다.

```
$ ros2 service list -t  
/clear [std_srvs/srv/Empty]  
/kill [turtlesim/srv/Kill]  
/reset [std_srvs/srv/Empty]  
/spawn [turtlesim/srv/Spawn]  
/turtle1/set_pen [turtlesim/srv/SetPen]  
/turtle1/teleport_absolute [turtlesim/srv/TeleportAbsolute]  
/turtle1/teleport_relative [turtlesim/srv/TeleportRelative]  
(생략)
```

5.2.4. 서비스 찾기

앞의 서비스 형태 확인`에서 언급한 서비스 형태 확인 명령어와 반대로 특정 형태를 입력하면 해당 형태의 서비스를 사용하는 서비스명을 확인하는 명령어도 있다. 사용법은 아래와 같이 `ros2 service find` 명령어에 매개변수로 특정 서비스 형태를 적어주면 된다.

```
$ ros2 service find std_srvs/srv/Empty  
/clear  
/reset  
$ ros2 service find turtlesim/srv/Kill  
/kill
```

5.2.5. 서비스 요청

지금까지 서비스에 대해 알아봤으니 실제 서비스 서버에게 서비스 요청(Request)을 해보도록 하자. 서비스 요청은 다음과 같이 `ros2 service call` 명령어를 이용하며 매개 변수로 서비스명, 서비스 형태, 서비스 요청 내용을 기술하면 된다.

```
ros2 service call <service_name> <service_type> "<arguments>"
```

첫번째로 다룰 서비스는 /clear 서비스로 turtlesim 노드를 동작할 때 표시되는 이동 궤적을 지우는 서비스이다. 이를 확인하기 위해서는 아래와 같이 teleop_turtle 노드를 실행시켜서 그림 58 처럼 미리 실행시켜둔 turtlesim 을 움직여보자.

```
$ ros2 run turtlesim turtle_teleop_key
```

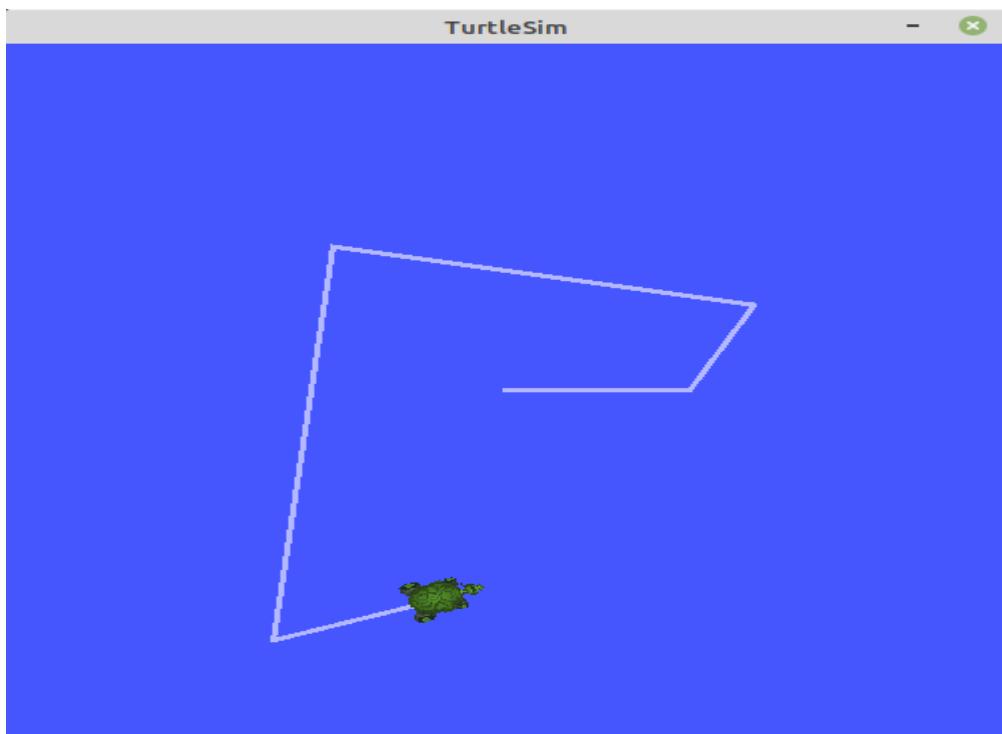


그림 58 turtlesim 이동 궤적

그 뒤 아래의 명령어로 /clear 서비스를 요청해보자. 그림 3의 이동 궤적이 서비스 요청 후에는 그림 59처럼 모두 지워짐을 확인할 수 있다. 아래 명령어에서 "<arguments>" 가 생략되었는데 이는 std_srvs/srv/Empty이라는 서비스 형태가 아무런 내용이 없는 형태로 사용할 수 있기 때문이다.

```
$ ros2 service call /clear std_srvs/srv/Empty  
requester: making request: std_srvs.srv.Empty_Request()  
response:  
std_srvs.srv.Empty_Response()
```



그림 59 이동 궤적이 지워진 모습

이번에는 /kill 서비스를 요청해보자. /kill 서비스는 죽이고자하는 거북이 이름을 서비스 요청의 내용으로 입력하면되는데 아래와 같이 turtle1이라고 이름을 지정하면 그림 60 과 같이 거북이가 사라졌음을 확인할 수 있다.

```
$ ros2 service call /kill turtlesim/srv/Kill "name: 'turtle1'"  
requester: making request: turtlesim.srv.Kill_Request(name='turtle1')  
response:  
turtlesim.srv.Kill_Response()
```



그림 60 kill 서비스에 의해 turtle 이 사라진 모습

이번에는 /reset 서비스를 요청해보자. 이 서비스는 그림 61 과 같이 모든 것이 리셋이 되면서 없어졌던 거북이가 다시 등장하거나 이동한 후에 모든 궤적도 사라지고 원점에 재위치 시킨다는 것을 확인할 수 있다.

```
$ ros2 service call /reset std_srvs/srv/Empty  
requester: making request: std_srvs.srv.Empty_Request()  
response:  
std_srvs.srv.Empty_Response()
```

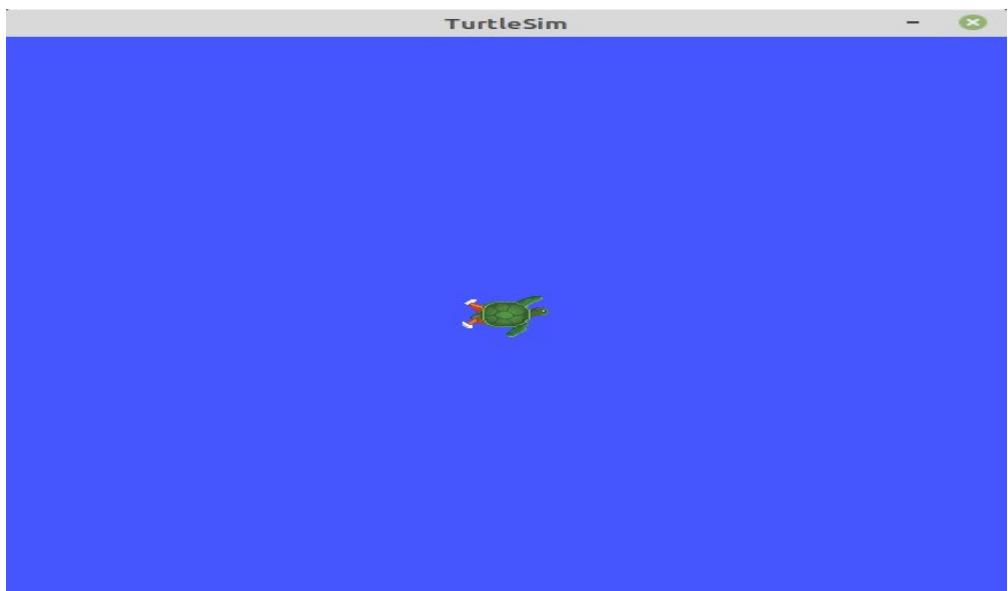


그림 61 리셋된 모습

이번에는 `/set_pen` 서비스를 요청해보자. 이 서비스는 지정한 거북이의 궤적 색과 크기를 변경하는 것으로 아래와 같이 `r`, `g`, `b` 값을 조합하여 색을 지정하고, `width`로 궤적의 크기를 지정할 수 있다. 이를 여러번 값을 변경해보며 테스트하면 그림 62 와 같다.

```
$ ros2 service call /turtle1/set_pen turtlesim/srv/SetPen "{r: 255, g: 255, b: 255, width: 10}"
requester: making request: turtlesim.srv.SetPen_Request(r=255, g=255, b=255, width=10,
off=0)
response:
turtlesim.srv.SetPen_Response()
```

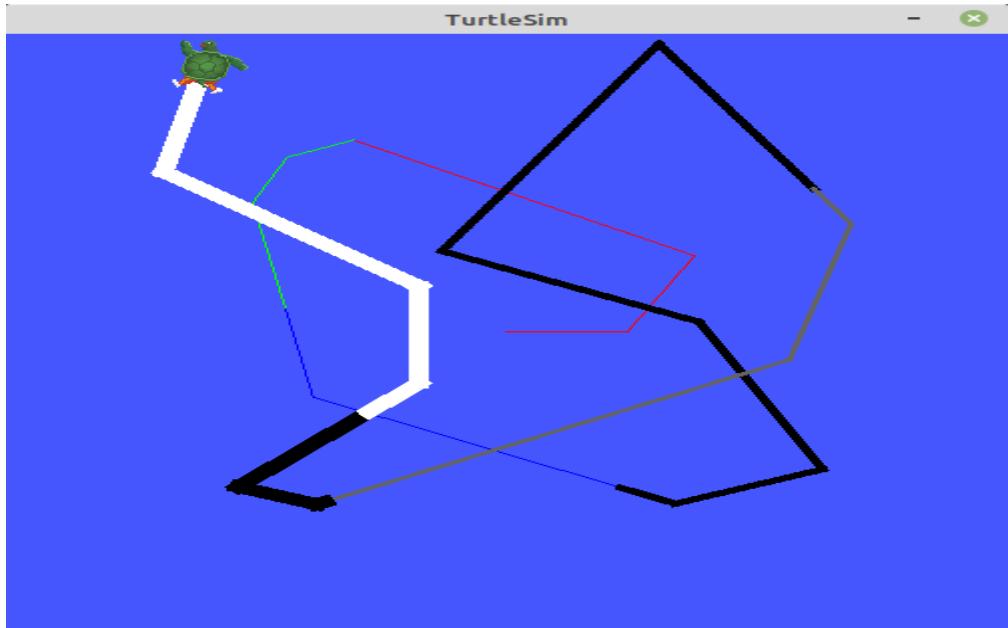


그림 62 이동 궤적의 색상 및 크기를 변경할 경우

마지막으로 `/spawn` 서비스를 요청해보자. 이 서비스는 지정한 위치 및 자세에 지정한 이름으로 거북이를 추가시키게된다. 이름은 옵션으로 지정하지 않으면 `turtle2`처럼 자동으로 지정되며 동일한 이름을 사용할 수는 없다.

아래 예제에서는 기본으로 지정된 `turtle1`을 `/kill` 서비스를 이용해 없애고 닌자거북이 4 총사인 `leonardo`, `raffaello`, `michelangelo`, `donatello`를 생성한 모습이다. 이를 수행하면 그림 63과 같이 4 마리의 거북이를 볼 수 있다.

```
$ ros2 service call /kill turtlesim/srv/Kill "name: 'turtle1'"
requester: making request: turtlesim.srv.Kill_Request(name='turtle1')
response:
turtlesim.srv.Kill_Response()
$ ros2 service call /spawn turtlesim/srv/Spawn "{x: 5.5, y: 9, theta: 1.57, name: 'leonardo'}"
requester: making request: turtlesim.srv.Spawn_Request(x=5.5, y=9.0, theta=1.57,
name='leonardo')
response:
turtlesim.srv.Spawn_Response(name='leonardo')
$ ros2 service call /spawn turtlesim/srv/Spawn "{x: 5.5, y: 7, theta: 1.57, name: 'raffaello'}"
requester: making request: turtlesim.srv.Spawn_Request(x=5.5, y=7.0, theta=1.57,
name='raffaello')
response:
turtlesim.srv.Spawn_Response(name='raffaello')
$ ros2 service call /spawn turtlesim/srv/Spawn "{x: 5.5, y: 5, theta: 1.57, name:
'michelangelo'}"
requester: making request: turtlesim.srv.Spawn_Request(x=5.5, y=5.0, theta=1.57,
name='michelangelo')
response:
turtlesim.srv.Spawn_Response(name='michelangelo')
$ ros2 service call /spawn turtlesim/srv/Spawn "{x: 5.5, y: 3, theta: 1.57, name: 'donatello'}"
requester: making request: turtlesim.srv.Spawn_Request(x=5.5, y=3.0, theta=1.57,
name='donatello')
response:
turtlesim.srv.Spawn_Response(name='donatello')
```



그림 63 닌자 거북이 4 총사

```
$ ros2 topic list
/donatello/cmd_vel
/donatello/color_sensor
/donatello/pose
/leonardo/cmd_vel
/leonardo/color_sensor
/leonardo/pose
/michelangelo/cmd_vel
/michelangelo/color_sensor
/michelangelo/pose
/new_turtle/cmd_vel
/new_turtle/pose
/parameter_events
/raffaello/cmd_vel
/raffaello/color_sensor
/raffaello/pose
/rosout
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
(생략)
```

5.3. 액션(Action)

5.3.1. 액션이란?

액션(action)은 그림 64의 `Node A - Node B`처럼 비동기식+동기식 양방향 메시지 송수신 방식으로 액션 목표(goal)를 지정하는 Action Client과 액션 목표를 받아 특정 태스크를 수행하면서

중간 결괏값에 해당되는 액션 피드백(feedback)과 최종 결괏값에 해당되는 액션 결과(result)를 전송하는 Action Server 간의 통신이라고 볼 수 있다. 이는 ROS 1의 액션 컨셉과 동일하다.



그림 64 액션 서버와 클라이언트

추가로 액션의 구현 방식을 더 자세히 살펴보면 그림 65와 같이 토픽과 서비스의 혼합이라고 볼 수 있는데 ROS 1이 토픽만을 사용하였다면 ROS 2에서는 액션 목표, 액션 결과, 액션 피드백은 토픽과 서비스가 혼합되어 있다.

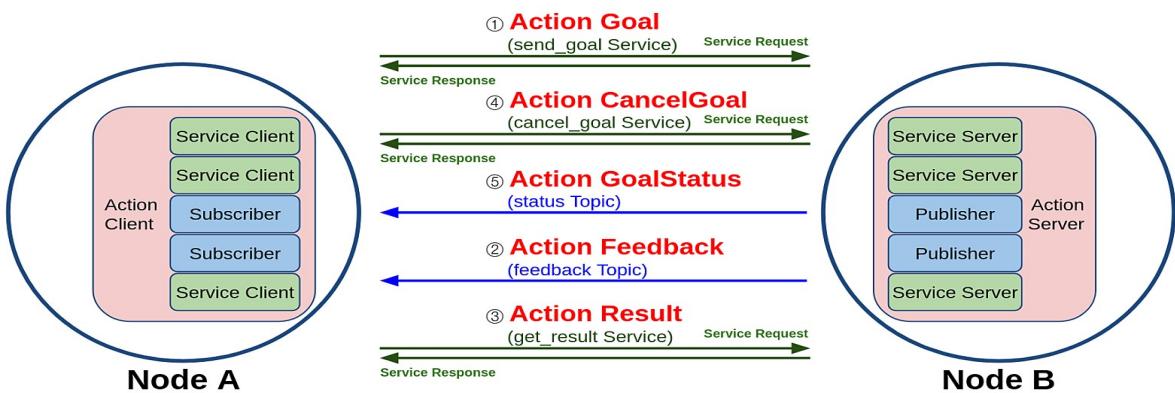


그림 65 액션 목표, 피드백, 결과

즉, 그림 65와 같이 Action Client는 Service Client 3개와 Topic Subscriber 2개로 구성되어있으며, Action Server는 Service Server 3개와 Topic Publisher 2개로 구성된다. 액션 목표/피드백/결과 (goal/feedback/result) 데이터는 msg 및 srv 인터페이스의 변형으로 action 인터페이스라고 한다.

ROS 1에서의 액션은 목표, 피드백, 결과 값을 토픽으로만 주고 받았는데 ROS 2에서는 토픽과 서비스 방식을 혼합하여 사용하였다. 그 이유로 토픽으로만 액션을 구성하였을 때 토픽의 특징인 비동기식 방식을 사용하게 되어 ROS 2 액션[10]에서 새롭게 선보이는 목표 전달(send_goal), 목표 취소 (cancel_goal), 결과 받기 (get_result)를 동기식인 서비스를 사용하기 위해서이다. 이런 비동기 방식을 이용하다보면 원하는 타이밍에 적절한 액션을 수행하기 어려운데 이를 원활히 구현하기 위하여 목표 상태(goal_state)라는 것이 ROS 2에서 새롭게 선보였다. 목표 상태는 목표 값을 전달 한 후의 상태 머신을 구동하여 액션의 프로세스를 쭋는 것이다. 여기서 말하는 상태머신은 Goal State

Machine 으로 그림 66 과 같이 액션 목표 전달 이후의 액션의 상태 값을 액션 클라이언트에게 전달할 수 있어서 비동기, 동기 방식이 혼재된 액션의 처리를 원활하게 할 수 있게 되어 있다.

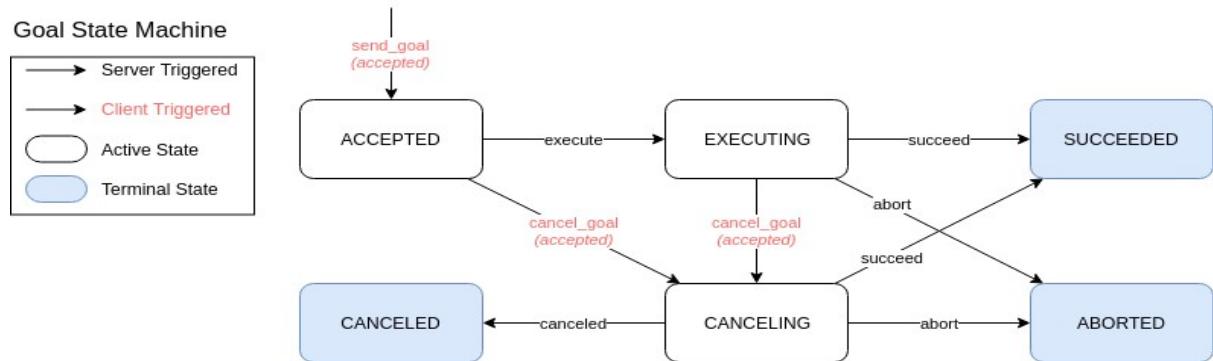


그림 66 Goal State Machine

5.3.2. 액션 서버 및 클라이언트

이번 ROS 2 액션 (action) 실습에도 turtlesim 노드와 teleop_turtle 노드를 이용하여 테스트해보겠다. 아래와 같이 두개의 노드를 각각 실행하자.

```
$ ros2 run turtlesim turtlesim_node
$ ros2 run turtlesim turtle_teleop_key
Reading from keyboard
-----
Use arrow keys to move the turtle.
Use G|B|V|C|D|E|R|T keys to rotate to absolute orientations. 'F' to cancel a rotation.
'Q' to quit.
```

지금까지는 teleop_turtle 노드가 실행된 터미널 창에서 $\leftarrow \uparrow \downarrow \rightarrow$ 키와 같이 화살표 키를 눌러 turtlesim 의 거북이를 움직여봤는데 이번에는 G, B, V, C, D, E, R, T 키를 사용해보겠다. 이 키들은 각 거북이들의 rotate_absolute 액션을 수행함에 있어서 액션의 목표 값을 전달하는 목적으로 사용된다. F 키를 중심으로 주위의 8 개의 버튼을 사용하는 것으로 그림 67 과 같이 각 버튼은 거북이를 절대 각도로 회전하도록 목표 값이 설정되어 있다. 그리고 F 키를 누르면 전달한 목표 값을 취소하여 동작을 바로 멈추게 된다. 여기서 G 키는 시계 방향 3 시를 가르키는 theta 값인 0.0 값에 해당되고 rotate_absolute 액션의 기준 각도가 된다. 다른 키는 위치별로 0.7854 radian 값씩 정회전 방향(반시계 방향)으로 각 각도 값이 할당되어 있다.

예를 들어 R 키를 누르면 1.5708 radian 목표 값이 전달되어 거북이는 12 기 방향으로 회전하게 된다. 각 키에 할당된 라디안 값은 코드를 확인하면 자세히 살펴볼 수 있다.

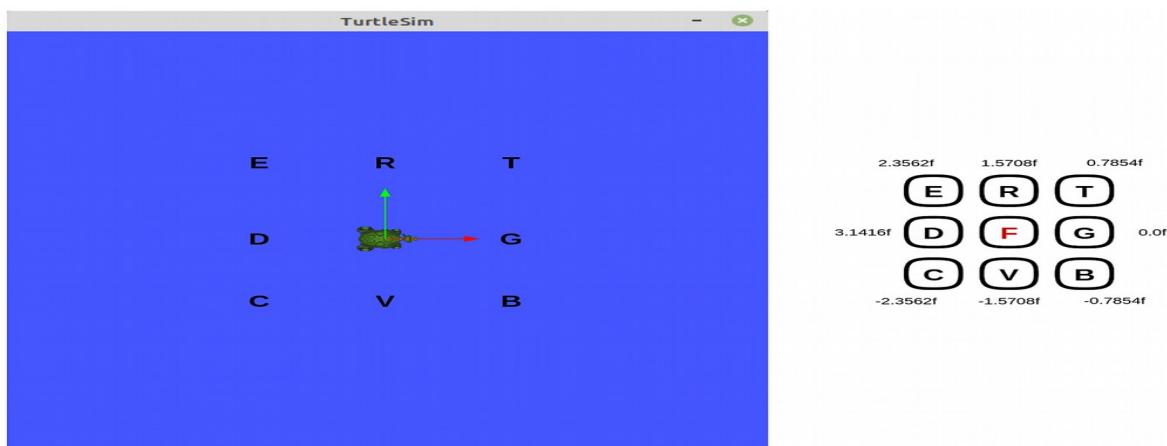


그림 67 각 키에 할당된 절대 자세 값

액션 목표는 도중에 취소할 수도 있는데 turtlesim_node 에도 그 상황을 터미널 창에 표시해준다. 예를 들어 액션 목표의 취소 없이 목표 theta 값에 도달하면 아래와 같이 표시된다.

[INFO]: Rotation goal completed successfully

하지만 액션 목표 theta 값에 도달하기 전에 turtle_teleop_key 가 실행된 터미널 창에서 F 키를 눌러 액션 목표를 취소하게 되면 turtlesim_node 이 실행된 터미널 창에 아래와 같이 목표가 취소 되었음을 알리면서 거북이는 그 자리에서 멈추게된다.

[INFO]: Rotation goal canceled

5.3.3. 노드 정보

각 실행된 노드의 액션 정보를 확인하기 위해서는 노드 정보 (ros2 node info) 명령어를 사용하는데 `008 ROS 2 노드와 메시지 통신` 강좌에서 이미 한번 다루었다. 액션 부분만 발췌하면 아래와 같음을 확인할 수 있다.

즉, turtlesim 노드는 Action Server 역할을 하며 turtlesim/action/RotateAbsolute 이라는 action 인터페이스를 사용하는 /turtle1/rotate_absolute 이라는 이름의 액션 서버임을 확인할 수 있다. teleop_turtle 노드는 /turtle1/rotate_absolute 의 액션의 Action Client 역할을 하고있다.

```
$ ros2 node info /turtlesim
/turtlesim
(생략)
Action Servers:
/turtle1/rotate_absolute: turtlesim/action/RotateAbsolute
Action Clients:

$ ros2 node info /teleop_turtle
/teleop_turtle
(생략)
Action Servers:

Action Clients:
```

```
/turtle1/rotate_absolute: turtlesim/action/RotateAbsolute
```

여기에 추가로 더 설명하자면 `rotate_absolute` 액션은 하기와 같이 `send_goal`, `cancel_goal`, `status`, `feedback`, `get_result` 의 5 가지로 더 세분화되어 있다. 이 각 액션 목표, 피드백, 결과에 대한 처리는 프로그래밍시 작성하게 된다.

```
/turtle1/rotate_absolute/_action/send_goal: turtlesim/action/RotateAbsolute_SendGoal
```

```
/turtle1/rotate_absolute/_action/cancel_goal: action_msgs/srv/CancelGoal
```

```
/turtle1/rotate_absolute/_action/status: action_msgs/msg/GoalStatusArray
```

```
/turtle1/rotate_absolute/_action/feedback: turtlesim/action/RotateAbsolute_FeedbackMessage
```

```
/turtle1/rotate_absolute/_action/get_result: turtlesim/action/RotateAbsolute_GetResult
```

5.3.4. 액션 목록

액션 정보를 확인하는 방법으로는 위와 같이 특정 노드의 정보를 확인하는 방법 이외에도 액션 목록 (`ros2 action list -t`) 명령어를 이용하여 현재 개발 환경에서 실행 중인 액션의 목록을 확인하는 방법도 있다.

```
$ ros2 action list -t  
/turtle1/rotate_absolute [turtlesim/action/RotateAbsolute]
```

5.3.5. 액션 정보

검색된 액션 목록의 더 자세한 정보를 확인하기 위해서는 액션 정보 (`ros2 action info`) 명령어를 이용하면 되는데 그 결과로 사용되는 액션 이름과 해당 액션의 서버 및 클라이언트 노드 이름 및 갯수를 표시해준다.

```
$ ros2 action info /turtle1/rotate_absolute  
Action: /turtle1/rotate_absolute  
Action clients: 1  
  /teleop_turtle  
Action servers: 1  
  /turtlesim
```

5.3.6. 액션 목표 전달

위 액션 서버 및 클라이언트`에서는 `teleop_turtle` 를 Action client로 하여 액션 목표(action goal)를 전달해보았다. 이번에는 ``ros2 action send_goal`` 명령어를 통하여 액션 목표(action goal)를 전달해보겠다. 이 명령어는 아래와 같이 ``ros2 action send_goal`` 명령어에 액션 이름, 액션 형태, 목표 값을 차례로 입력하면 된다.

```
ros2 action send_goal <action_name> <action_type> "<values>"
```

예를 들어 거북이를 12 시 방향인 theta: 1.5708 값을 목표로 주게되면 그림 68 같이 이동하게 되며 아래 처럼 전달한 목표 값과 액션 목표의 UID(Unique ID), 결괏값으로 이동 시작 위치로의 변위 값인 delta 를 결과로 보여주며 마지막으로 상태를 표시하게 된다.

```
$ ros2 action send_goal /turtle1/rotate_absolute turtlesim/action/RotateAbsolute "{theta: 1.5708}"
Waiting for an action server to become available...
Sending goal:
  theta: 1.5708

Goal accepted with ID: b991078e96324fc994752b01bc896f49

Result:
  delta: -1.5520002841949463

Goal finished with status: SUCCEEDED
```

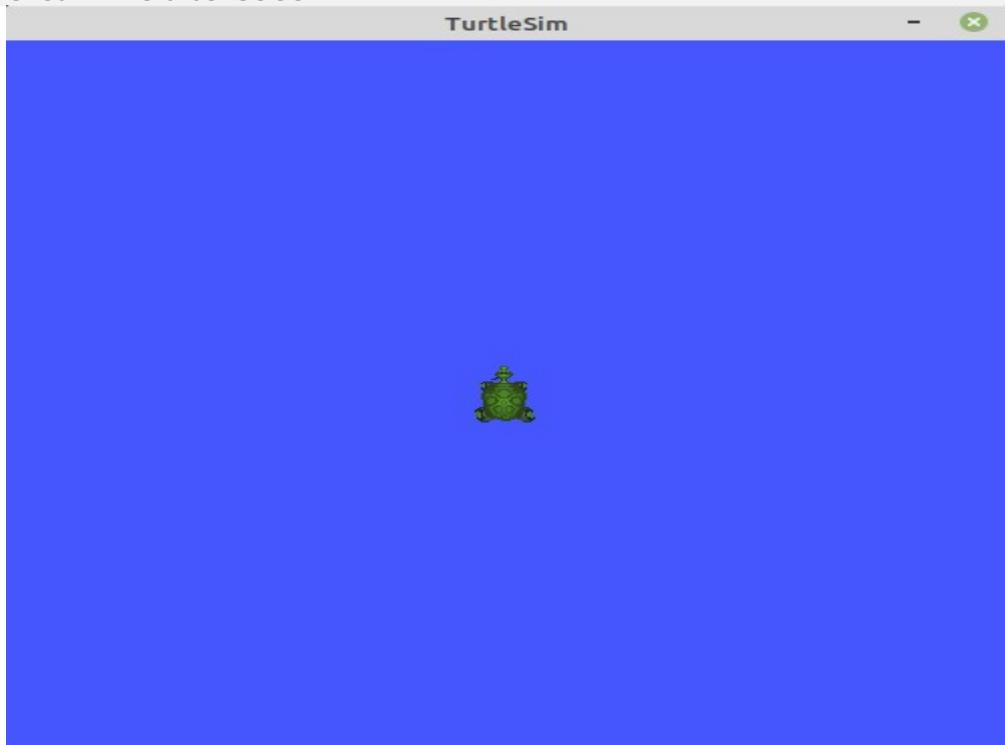


그림 68 thetha: 1.5708 으로 회전한 모습

위 명령어에서는 피드백이 포함되어 있지 않았는데 이를 화면에 표시하기 위해서는 아래와 같이 --feedback 옵션을 붙여주면 된다. 이를 통해 거북이가 정해진 theta (-1.5708, 시계 6 시 방향)으로 회전하며 위에서 언급한 목표 값, UID, 결괏값, 상태 값 이외에도 Feedback 값으로 총 회전 값의 남은 회전량을 피드백으로 표시하게 된다. 이를 수행하면 그림 6 과 같이 시계 6 시 방향으로 회전하게 된다.

```
$ ros2 action send_goal /turtle1/rotate_absolute turtlesim/action/RotateAbsolute "{theta: -1.5708}" --feedback
Waiting for an action server to become available...
Sending goal:
  theta: -1.5708
Goal accepted with ID: ad7dc695c07c499782d3ad024fa0f3d2
```

```
Feedback:  
  remaining: -3.127622127532959  
Feedback:  
  remaining: -3.111621856689453  
Feedback:  
  remaining: -3.0956220626831055  
(생략)  
  
Feedback:  
  remaining: -0.03962135314941406  
Feedback:  
  remaining: -0.023621320724487305  
Feedback:  
  remaining: -0.007621288299560547  
Result:  
  delta: 3.1040005683898926  
Goal finished with status: SUCCEEDED
```



그림 69 theta : -1.5708 으로 회전한 모습

5.4. 파라미터

ROS 2에서의 파라미터(parameter)는 그림 70과 같이 각 노드에서 파라미터 관련 Parameter server를 실행시켜 그림 2와 같이 외부의 Parameter client 와 통신으로 파라미터를 변경하는 것으로 `010 ROS 2 서비스 (service)`[4] 강좌에서 다루었던 서비스(service)와 동일하다고 볼 수 있다. 단 서비스가 서비스 요청과 응답이라는 RPC(remote procedure call)가 목적이었다면 파라미터는 노드 내 매개변수를 서비스 데이터 통신 방법을 사용하여 노드 내부 또는 외부에서 쉽게 지정(Set)하거나 변경할 수 있고, 쉽게 가져(Get)와서 사용할 수 있게 하는 점에서 그 사용 목적이 다르다고 볼 수 있다.

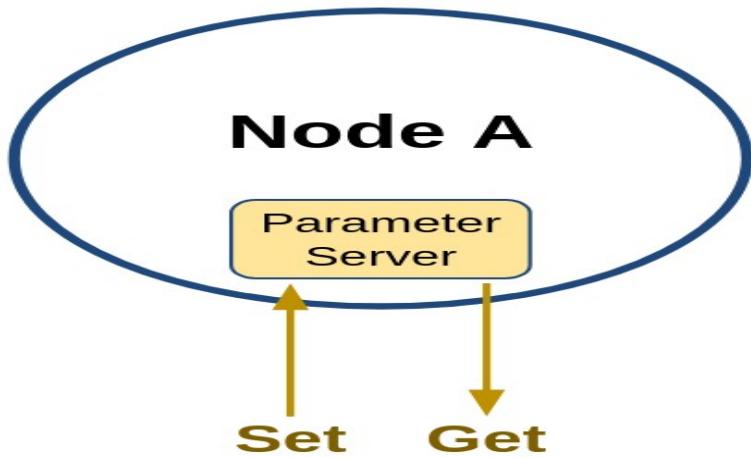


그림 70 파라미터 서버

파라미터 관련 기능은 RCL(ROS Client Libraries)의 기본 기능으로 그림 71과 같이 모든 노드가 자신만의 Parameter server를 가지고 있고, 그림 71과 같이 각 노드는 Parameter client로 포함시킬 수 있어서 자기 자신의 파라미터 및 다른 노드의 파라미터를 읽고 쓸 수 있게 된다. 이를 활용하면 각 노드의 다양한 매개변수를 글로벌 매개변수처럼 사용할 수 있게 되어 추가 프로그래밍이나 컴파일 없이 능동적으로 변화 가능한 프로세스를 만들 수 있게 된다. 그리고 각 파라미터는 yaml 파일 형태의 파라미터 설정 파일을 만들어 초기 파라미터 값 설정 및 노드 실행시에 파라미터 설정 파일을 불러와서 사용할 수 있기에 ROS 2 프로그래밍에 매우 유용하게 사용할 수 있다.

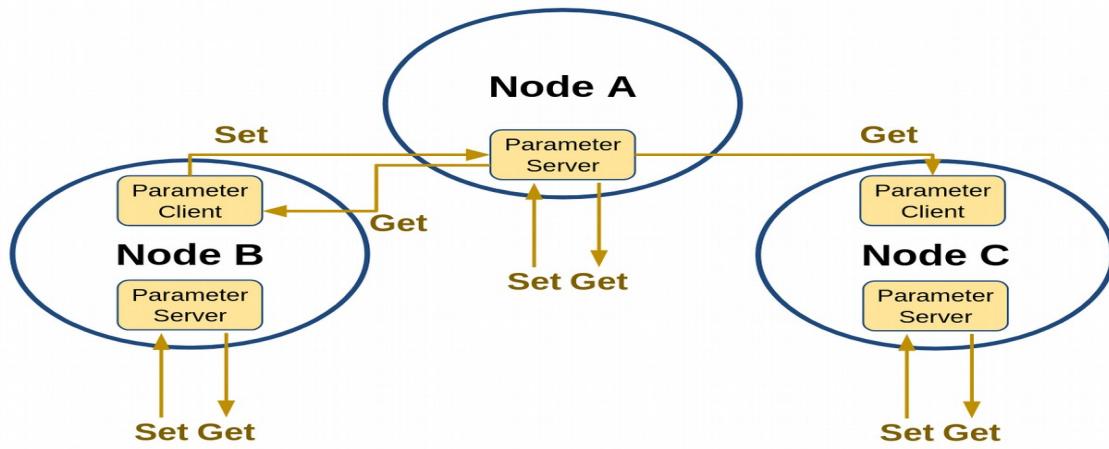


그림 71 파라미터 서버와 클라이언트

5.4.1. 파라미터 목록 확인

우선 파라미터 사용 방법 실습에 앞서서 우리 ROS 의 친구인 거북이를 불러오도록 하겠다. 다음과 같이 2 개의 터미널 창에서 각각 turtlesim 노드와 teleop_turtle 노드를 실행시키자.

```
$ ros2 run turtlesim turtlesim_node
$ ros2 run turtlesim turtle_teleop_key
```

그 뒤 아래와 같이 파라미터 목록을 확인할 수 있는 파라미터 목록 확인 (ros2 param list) 명령어를 사용하면 현재 개발 환경에서 실행 중인 노드(2 개, turtlesim 와 teleop_turtle)의 Parameter server 에 접근하여 현재 사용 가능한 파라미터를 노드별로 목록화 하여 표시하게 된다. 여기서 `use_sim_time` 이라는 파라미터가 공통으로 보이는데 이는 모든 노드의 기본 파라미터로 나중에 다를 예정인 ROS 의 3 차원 시뮬레이터 Gazebo 등의 시뮬레이션 때 사용하는 파라미터임을 알아두자.

```
$ ros2 param list
/teleop_turtle:
  scale_angular
  scale_linear
  use_sim_time
/turtlesim:
  background_b
  background_g
  background_r
  use_sim_time
```

5.4.2. 파라미터 내용 확인

이번에는 파라미터 목록 확인을 통해 알아본 파라미터가 어떤 형태, 목적, 데이터 형태, 최소/최댓값을 가지는지 알아보자. 이를 알아보기 위해서는 파라미터 내용 확인 (ros2 param describe) 명령어를 이용하면 되고 실제 사용 방법은 다음 명령어와 같이 `ros2 param describe` 명령어에 노드 이름과

파라미터 이름을 지정하면 된다. 아래 예제는 turtlesim 노드의 background_b 파라미터에 대해 알아본 결과로 파라미터 이름은 background_b이고, 데이터 형태는 integer이며, 배경 컬러의 파랑 채널이라는 설명과 함께 최소, 최대, 스텝 값을 확인할 수 있다.

```
$ ros2 param describe /turtlesim background_b
Parameter name: background_b
Type: integer
Description: Blue channel of the background color
Constraints:
  Min value: 0
  Max value: 255
  Step: 1
```

5.4.3. 파라미터 읽기

파라미터 읽기 (ros2 param get) 명령어는 아래와 같이 파라미터 읽기 (ros2 param get) 명령어에 노드 이름과 파라미터 이름을 함께 적어주면 된다.

```
ros2 param get <node_name> <parameter_name>
```

예를 들어 위에서 사용했던 파라미터 목록 확인 (ros2 param list) 명령어를 통해 확인된 background_r, background_g, background_b 파라미터의 값을 읽어 오면 아래와 같은 결과를 확인할 수 있다. 즉, 현재 실행 중인 turtlesim 노드의 배경색은 r = 69, g = 86, b = 255 값의 색상으로 Neon Blue(#4556FF, <https://www.htmlcsscolor.com/hex/4556FF>)에 해당되게 된다. 지금까지 강좌에서 파란색 화면 상의 거북이라고 했었는데 그 파란색이 정확히는 r = 69, g = 86, b = 255 이였던 것이다.

```
$ ros2 param get /turtlesim background_r
Integer value is: 69
```

```
$ ros2 param get /turtlesim background_g
Integer value is: 86
```

```
$ ros2 param get /turtlesim background_b
Integer value is: 255
```

5.4.4. 파라미터 쓰기

위에서 파라미터 읽기를 해봤으니 다음은 파라미터 쓰기를 해보자. 파라미터 쓰기 (ros2 param set) 명령어는 아래와 같이 파라미터 쓰기 (ros2 param set) 명령어에 노드 이름, 파라미터 이름, 변경할 파라미터의 값을 지정하면 된다.

```
ros2 param set <node_name> <parameter_name> <value>
```

예를 들어 배경색을 Dark Violet(#9400D3, <https://www.htmlcsscolor.com/hex/9400D3>)으로 변경하기 위하여 r = 148, g = 0, b = 211 값으로 변경해보자.

```
$ ros2 param set /turtlesim background_r 148
Set parameter successful
```

```
$ ros2 param set /turtlesim background_g 0  
Set parameter successful
```

```
$ ros2 param set /turtlesim background_b 211  
Set parameter successful
```

이 명령어가 문제없이 실행되었다면 그림 72의 기본 Neon Blue의 배경색에서 Dark Violet 배경색으로 변경되었음을 확인할 수 있을 것이다. 이렇게 파라미터는 별도의 추가 프로그래밍 작업 없이도 정해놓은 파라미터를 외부에서 읽고, 쓰기를 하여 프로그래밍, 컴파일 과정이 없이도 다양한 기능을 구현할 수 있게 된다. 위 예제에서는 파라미터의 외부 접근만 다루었지만 해당 노드의 내부에서도 이 파라미터를 자유롭게 접근 가능하기 때문에 개발자가 원하는 형태로 쓸 수 있는 글로벌 매개변수로 생각하고 이용해도 좋을 것이다. 그리고 이 강좌에서는 `ros2 param`이라는 ROS 2 CLI 툴을 이용하였지만 이 이외에도 GUI 형태의 rqt 플러그인을 이용하면 더욱 편하다. 이 부분은 다음 강좌에서 다루기로 하겠다. 그리고 파라미터 관련 툴이 아닌 각 연관된 노드들의 코드에 파라미터 클라이언트 형태로 사용할 수 있게 API를 갖추고 있기에 다양한 목적으로, 다양한 프로그래밍 언어로, 다양한 구현 방식으로 사용할 수 있게 되어 있다.

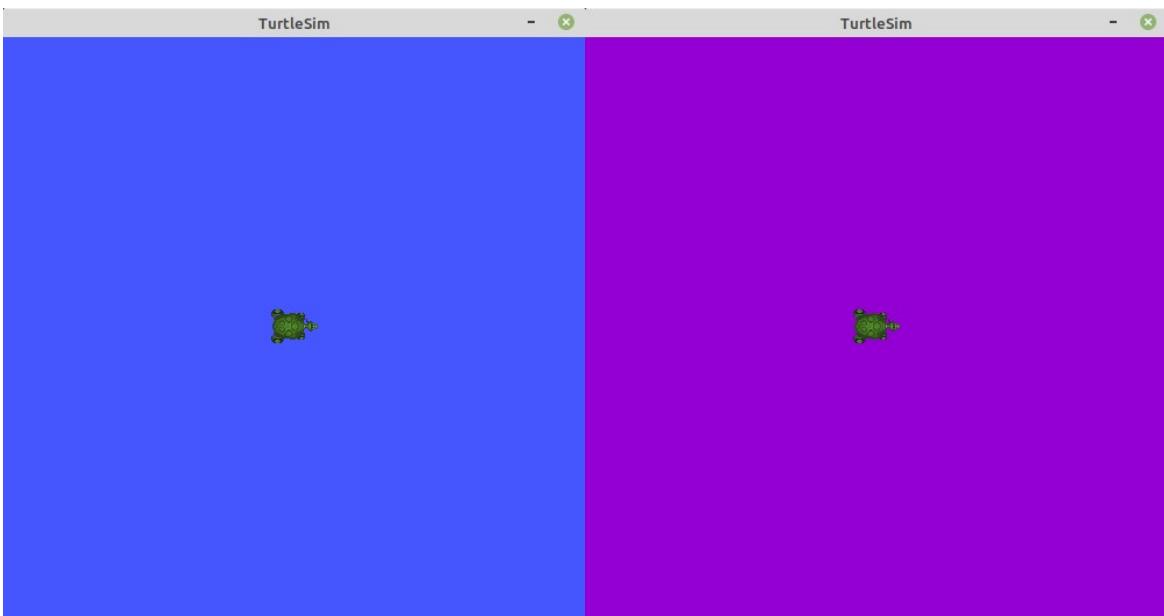


그림 72 Neoblue 배경색(좌측), Dark Violet 배경색(우측)

5.4.5. 파라미터 저장

위 내용에서 파라미터 쓰기 (ros2 param set)를 사용하면 기본 파라미터의 값을 바꾸어 사용할 수 있다는 것을 알았다. 이 명령어를 이용하면 파라미터를 변경할 수 있지만 turtlesim 노드를 종료하였다가 다시 시작하면 모든 파라미터는 초기값으로 다시 설정된다. 이에 필요하다면 현재 파라미터를 저장하고 다시 불러와야 하는데 이 때에는 파라미터 저장 (ros2 param dump) 명령어를 이용하면 된다.

파라미터 저장 (ros2 param dump)는 아래와 같이 `ros2 param dump` 명령어에 노드 이름을 적어주면 현재 폴더에 해당 노드 이름으로 설정 파일이 yaml 형태로 저장된다. 예를 들어 아래 예제에서는 `turtlesim.yaml` 파일로 저장되었다.

```
$ ros2 param dump /turtlesim  
Saving to: ./turtlesim.yaml
```

이 `turtlesim.yaml` 파일에는 turtlesim 노드가 파라미터 서버로하여 가지고 있는 background_r, background_g, background_b, use_sim_time 과 같이 4 개의 파라미터의 현재 값이 저장되게 된다.

```
$ cat ./turtlesim.yaml  
turtlesim:  
  ros_parameters:  
    background_b: 211  
    background_g: 0  
    background_r: 148  
    use_sim_time: false
```

노드 실행시 저장된 파라미터 값을 사용하려면 아래 예제와 같이 --ros-args --params-file 옵션과 함께 지정된 yaml 파일의 위치를 적어주면 된다. 이 명령어 처럼 노드 실행시 파라미터 파일을 함께 기재하면 파라미터의 기본 값이 지정된 파일에 기재된 내용으로 초기화하게 된다.

```
$ ros2 run turtlesim turtlesim_node --ros-args --params-file ./turtlesim.yaml
```

5.4.6. 파라미터 삭제

특정 파라미터 삭제는 아래 예제와 같이 `ros2 param delete` 명령어에 노드 이름과 파라미터 이름을 적어주면 된다. 일반적인 경우에는 많이 사용되지는 않는 명령어이다.

```
$ ros2 param delete /turtlesim background_b  
Deleted parameter successfully
```

파라미터가 삭제된 상태를 확인해보면 아래 예제와 같다.

```
$ ros2 param list /turtlesim  
background_g  
background_r  
use_sim_time
```

6. ROS2 기본 프로그래밍

6.1. ROS 프로그래밍 규칙

6.1.1. ROS 프로그래밍 코드 스타일

오픈소스 코드는 해당 커뮤니티의 공동의 결과물로 협업이 기반이 된다. 협업 프로그래밍 작업 시에는 일관된 규칙을 만들고 이를 준수하며 프로그래밍하고 혹시 모를 실수에 대비하여 자동화 툴로 자가 검토를 실시하고 공개 후에도 여러 동료들로부터 코드 리뷰도 받게 된다. 이러한 코드 스타일 가이드 준수는 처음에는 귀찮고 버거울 수 있으나 소스 코드 작업 시 빈번히 생기는 개발자의 부가적인 선택을 줄여주고, 다른 협업 개발자 및 이용자의 코드 이해도를 높이며 상호 간의 코드 리뷰를 쉽게 할 수 있으며, 프로그래밍 언어의 특정 기능으로 인해 생길 수 있는 오류와 다양한 이슈를 피할 수 있게 해준다.

로봇운영체제 ROS도 ROS 커뮤니티의 공동의 결과물로 협업이 기반이 된다. ROS 커뮤니티도 이를 위해 `ROS 2 developer guide`과 `ROS Enhancement Proposals (REPs)`과 같은 가이드와 규칙도 만들고 `ROS 2 Code style`와 같이 일관된 코드 스타일을 지키기 위하여 사용되는 각 언어에 대해 스타일 가이드라인을 세우고 구성원들의 합의하에 이를 따르고 있다. 그렇다고 독단적인 패키지 레이아웃이나 문서 레이아웃이 아닌 오픈소스 커뮤니티들에서 가장 많이 사용되고 있는 인기 있는 스타일을 바탕으로 자체 가이드라인을 만들어 사용하고 있다. 또한 일관된 코드 스타일을 관리하기 위하여 개발 툴을 제공하고 있으며 개발자가 커뮤니티의 프로그래밍 지침을 준수하는지 확인할 수 있게 하고 있다.

6.1.2. 기본 이름 규칙

하기와 같이 3 종류의 네이밍을 기본으로 하며 파일 이름은 모두 소문자로 `snake_case` 이름 규칙을 사용한다. 가독성을 해치는 축약어는 가능한 사용하지 않으며 확장자명은 모두 소문자로 표기한다.

단, `ROS` 인터페이스 류의 파일은 /msg 및 /srv 또는 /action 에 폴더에 위치시키며 인터페이스 파일명은 `CamelCased` 규칙을 따른다. 그 이유는 *.msg 및 *.srv 또는 *.action는 *.h(pp) 변환 후 인터페이스 타입으로 구조체 및 타입으로 사용되기 때문이다.

- CamelCased
- snake_case
- ALL_CAPITALS

그 이외에 특정 목적에 의해 만들어지는 하기 파일 이름은 예외적으로 대소문자 규칙을 따르지 않고 하기와 같이 고유의 이름을 사용한다.

- package.xml
- CMakeLists.txt
- README.md
- LICENSE
- CHANGELOG.rst
- .gitignore
- .travis.yml
- *.repos

6.1.3. C++ 개발 규칙

ROS 2 Developer Guide 및 ROS 2 Code style에서 다루고 있는 C++ 코드 스타일은 오픈소스 커뮤니티에서 가장 널리 사용 중인 Google C++ Style Guide를 사용하고 있으며 ROS의 특성에 따라 일부를 수정하여 사용하고 있다. 하기에 기재되지 않는 부분은 REPs 및 Google C++ Style Guide 문서를 참고하도록 하자.

(1) 기본 규칙 : C++14 Standard [6]를 준수한다.

(2) 라인 길이 : 최대 100 문자

(3) 이름 규칙 (Naming) : `CamelCased`, `snake_case`, `ALL_CAPITALS` 만을 사용한다.

- ✓ CamelCased: 타입, 클래스, 구조체, 열거형
- ✓ snake_case: 파일, 패키지, 인터페이스, 네임스페이스, 변수, 함수, 메소드
- ✓ ALL_CAPITALS: 상수, 매크로
- ✓ 소스 파일은 `.cpp` 확장자를 사용한다.
- ✓ 헤더 파일은 `.hpp` 확장자를 사용한다.
- ✓ 전역변수(global variable)는 사용이 피치 못한 경우에는 `g_` 접두어를 붙인다.
- ✓ 클래스 멤버 변수(class member variable)는 마지막에 밑줄(`_`)을 붙인다.

(4) 공백 문자 대 탭 (Spaces vs. Tabs)

- ✓ 기본 들여쓰기(indent)는 공백 문자(space) `2 개`를 사용한다. (탭(tab)문자 사용 금지)
- ✓ `Class`의 `public:`, `protected:`, `private:`은 들여쓰기를 사용하지 않는다.

- ✓ 괄호 (Brace) : 모든 if, else, do, while, for 구문에 괄호를 사용한다.

(5) 주석 (Comments)

- ✓ 문서 주석에는 `/** */` 을 사용한다.
- ✓ 구현 주석에는 `//` 을 사용한다.

(6) 린터 (Linters)

- ✓ C++ 코드 스타일의 자동 오류 검출을 위하여 ament_cpplint, ament_uncrustify 를 사용하고 정적 코드 분석이 필요한 경우 ament_cppcheck [9]를 사용한다.

(7) 기타

- ✓ Boost 라이브러리의 사용은 가능한 피하고 어쩔 수 없을 경우에만 사용한다.
- ✓ 포인트 구문은 `char * c;` 처럼 사용한다. (`char* c;` 이나 `char *c;` 처럼 사용하지 않는다.)
- ✓ 중첩 템플릿은 `set<list<string>>` 처럼 사용한다. (`set<list<string> >` 또는 `set< list<string> >` 처럼 사용하지 않는다.)

(8) C++ 개발 규칙에 따른 문법 예시

```
int main(int argc, char **argv)
{
    if (condition) {
        return 0;
    } else {
        return 1;
    }
}

if (this && that || both) {
    ...
}

// Long condition; open brace
if (
    this && that || both && this && that || both && this && that ||
    both && this && that)
{
    ...
}

// Short function call
call_func(foo, bar);

// Long function call; wrap at the open parenthesis
call_func(
    foo, bar, foo, bar, foo, bar, foo, bar, foo, bar, foo, bar,
    foo, bar, foo, bar, foo, bar, foo, bar, foo, bar, foo, bar, foo, bar);

// Very long function argument; separate it for readability
```

```

call_func(
    bang,
    fooooooooooooooooooooooooooooooooooooo,
    bar, bat);

ReturnType LongClassName::ReallyReallyReallyLongFunctionName(
    Type par_name1, // 2 space indent
    Type par_name2,
    Type par_name3)
{
    DoSomething(); // 2 space indent
    ...
}

MyClass::MyClass(int var)
: some_var_(var),
  some_other_var_(var + 1)
{
    ...
    DoSomething();
    ...
}

```

6.1.4. Python 개발 규칙

ROS 2 Developer Guide 및 ROS 2 Code style에서 다루고 있는 Python 코드 스타일은 Python Enhancement Proposals (PEPs)의 PEP 8를 준수한다.

(1) 기본 규칙 : Python 3 (Python 3.5 이상)를 사용한다.

(2) 라인 길이 : 최대 100 문자

(3) 이름 규칙 (Naming)

- ✓ `CamelCased`, `snake_case`, `ALL_CAPITALS` 만을 사용한다.
- ✓ CamelCased: 타입, 클래스
- ✓ snake_case: 파일, 패키지, 인터페이스, 모듈, 변수, 함수, 메소드
- ✓ ALL_CAPITALS: 상수

(4) 공백 문자 대 탭 (Spaces vs. Tabs)

- ✓ 기본 들여쓰기(indent)는 공백 문자(space) `4 개`를 사용한다. (탭(tab)문자 사용 금지)
- ✓ `Hanging indent` (문장 중간에 들여쓰기를 사용하는 형식)의 사용 방법은 아래 예제를 참고.
- ✓ 괄호 및 공백 사용은 아래 예제를 참고하자.
- ✓ 예제 (올바른 사용법)

```
foo = function_name(var_one, var_two, var_three, var_four)
```

```
def long_long_long_long_function_name(  
    var_one,  
    var_two,  
    var_three,  
    var_four):  
    print(var_one)
```

- ✓ 예제 (잘못된 사용법)

```
foo = long_function_name(var_one, var_two,  
                        var_three, var_four)  
  
foo = long_function_name(var_one, var_two,  
                        var_three, var_four)  
  
def long_function_name(  
    var_one, var_two, var_three,  
    var_four):  
    print(var_one)
```

(5) 괄호 (Brace)

- ✓ 괄호는 계산식 및 배열 인덱스로 사용하며, 자료형에 따라 적절한 괄호(대괄호 `[]`, 중괄호 `{ }`, 소괄호 `()`)를 사용한다.

```
list = [1, 2, 3, 4, 5]  
dictionary = {'age': 30, 'name': '홍길동'}  
tupple = (1, 2, 3, 4, 5)
```

(6) 주석 (Comments)

- ✓ 문서 주석에는 `"""`을 사용하며 Docstring Conventions을 기술한 PEP 257]을 준수한다.
- ✓ 구현 주석에는 `#`을 사용한다.

(7) 린터 (Linters) : Python 코드 스타일의 자동 오류 검출을 위하여 ament_flake8 를 사용.

(8) 기타 : 모든 문자는 큰 따옴표(`"`, double quotes)가 아닌 작은 따옴표(`'', single quotes)를 사용하여 표현한다.

6.2. ROS2 패키지 생성 및 빌드

패키지는 ROS2 를 활용한 소스코드의 컨테이너로 다른 사람들이 코드를 설치하거나 공유를 하기 위해 구성하고 있다. ROS2 패키지를 활용하여 다른 사람들이 쉽게 코드를 수정하고 구현할 수 있는 것이. ROS2 에서 패키지 생성은 ament 를 빌드 시스템으로, build tool 은 colcon 을 사용하게 된다. 물론 기존 ROS1 에서 사용하는 CMake 또는 Python 을 사용하여 패키지를 생성할 수도 있다. 본 교재는 ament 기반의 패키지 생성 시스템과 colcon 기반의 빌드 시스템을 활용하여 패키지를 생성하게 된다.

6.2.1. 패키지 생성하기

ROS 2 패키지를 생성하는 방법으로는 두 가지인데 하나는 직접 패키지 폴더를 만들고 그 안에 파일 시스템에 필수적인 `package.xml`이나 `CMakeLists.txt` 또는 `setup.py` 등을 포함시켜주고 소스 코드를 작성하는 것과 ros2cli 명령어를 이용하는 것이다. 경험적으로는 자신이 주로 사용하는 기능들을 중심으로 기본 패키지를 만들어두고 복사/붙여넣기로 패키지를 만든 후 이름과 몇 가지 설정만 변경하여 사용하는 전자의 방법이 편하다. 하지만 처음에는 후자의 ros2cli 명령어를 사용하여 소스 코드 작업하는 게 편할 수 있다. 이에 연습 삼아 후자의 방법으로 간단한 패키지를 생성해보자.

패키지 생성 명령어는 다음과 같다. `ros2 pkg create` 명령어를 사용하고 그 뒤에 옵션을 붙여 주게 된다. 참고로 하기 명령어를 실행하는 폴더 위치는 `사용자 작업 폴더`임을 잊지 말자.

```
$ ros2 pkg create [패키지이름] --build-type [빌드 타입] --dependencies [의존하는패키지 1] [  
의존하는패키지 n]
```

우선 빌드 타입인데 RCL으로 C++을 사용한다면 ament_cmake을 설정하고, Python을 사용한다면 ament_python을 기입해 주자. 참고로 GUI 프로그램을 작성해야 한다면 python을 사용한다고 하더라도 rqt plugin 계열을 써야 하기에 ament_cmake을 기입하면 된다.

```
$ ros2 pkg create test_pkg_rclcpp --build-type ament_cmake  
$ ros2 pkg create test_pkg_rclpy --build-type ament_python
```

'ros2 pkg create'는 사용자가 패키지를 작성할 때 ament 빌드 시스템에 꼭 필요한 CMakeLists.txt와 package.xml을 포함한 패키지 폴더를 생성한다. 이해를 돋기 위해 간단한 패키지를 작성해 보자. 먼저 새로운 터미널 창을 열고(Ctrl + Alt + t) 다음 명령어를 실행하여 작업 폴더로 이동한다

```
$ cd ~/colcon_ws/src
```

생성할 패키지 이름은 'my_first_ros_rclcpp_pkg'이다. ROS에서 패키지 이름은 모두 소문자를 사용하며 공백이 있으면 안 된다. 그리고 붙임표(-) 대신에 밑줄(_)을 사용해 각 단어를 이어붙이는 것을 스타일 규칙으로 삼고 있다. ROS 프로그래밍에서 코딩 스타일과 이름 규칙은 추후 이어지는 강좌에서 더 자세히 다루도록 하겠다. 그럼 다음 명령어로 my_first_ros_rclcpp_pkg 이름의 패키지를 생성해보자.

```
$ ros2 pkg create my_first_ros_rclcpp_pkg --build-type ament_cmake --dependencies rclcpp  
std_msgs
```

만약 Python을 사용한다면 하기와 같이 지정하면 된다.

```
$ ros2 pkg create my_first_ros_rclpy_pkg --build-type ament_python --dependencies rclpy  
std_msgs
```

앞의 명령어에서 의존하는 패키지로 'std_msgs'와 사용되는 클라이언트 라이브러리에 따라 'rclcpp' 또는 'rclpy'를 옵션으로 달아주었다. ROS의 표준 메시지 패키지인 std_msgs와 ROS에서 C/C++를 사용하기 위하여 클라이언트 라이브러리인 rclcpp 또는 Python을 사용하기 위한 클라이언트 라이브러리 rclpy를 사용하겠다는 것으로 패키지 생성에 앞서 미리 설치해야 한다는 의미이다. 이러한 의존하는 패키지 설정은 패키지를 생성할 때 지정할 수도 있지만, 생성한 다음 package.xml에서 직접 입력해도 된다.

패키지를 생성하였다면 '~/colcon_ws/src'에 'my_first_ros_xxxxx_pkg' 패키지 폴더와 ROS 패키지가 갖추어야 할 기본 내부 폴더 그리고 package.xml 파일들이 생성된다. ament_cmake이나 ament_python이나에 따라 기본 구성 파일 시스템이 좀 상이한데 기본적으로는 아래와 같이 구성된다.

```
(my_first_ros_rclcpp_pkg)
.
└── include
    └── my_first_ros_rclcpp_pkg
└── src
└── CMakeLists.txt
└── package.xml
```

3 directories, 2 files

```
(my_first_ros_rclpy_pkg)
.
└── my_first_ros_rclpy_pkg
    └── __init__.py
└── resource
    └── my_first_ros_rclpy_pkg
└── test
    ├── test_copyright.py
    ├── test_flake8.py
    └── test_pep257.py
└── package.xml
└── setup.cfg
└── setup.py
```

3 directories, 8 files

6.2.2. ROS2 빌드 툴

ROS 1의 경우 여러 가지 다른 도구, 즉 catkin_make, catkin_make_isolated 및 catkin_tools가 지원되었다. ROS 2에서는 알파, 베타, 그리고 Ardent 릴리스까지 빌드 도구로 ament_tools가 이용되었고 ROS 2 Bouncy부터는 colcon을 추천하고 있다. colcon (collective construction)은 ROS 2 패키지를 작성, 테스트, 빌드 등 ROS 2 기반의 프로그램할 때 빼놓을 수 없는 툴로 작업 흐름을 향상시키는 CLI 타입의 명령어 도구이다. 사용 방법은 `colcon build`와 같은 CLI 형태의 명령어로 터미널창에서 수행하게 되며 다양한 옵션과 함께 사용할 수 있다.

catkin_make

catkin_make는 ROS 1 빌드 시스템을 포함하는 ROS 패키지 catkin에서 제공하던 기본 툴로 ROS Fuerte 버전 이후 rosbuild의 대체 툴로서 오랜 기간 사용되어온 ROS 1의 대표 빌드 툴이다.

catkin_make_isolated

catkin_make과 마찬가지로 ROS 1 빌드 시스템을 포함하는 ROS 패키지 catkin에서 제공하던 기본 툴로 하나의 CMake으로 복수의 패키지를 빌드 할 수 있었으며 격리 빌드를 지원함으로써 모든 패키지를 별도로 빌드하게 되었다. 이 기능 변화를 통해 설치용 폴더를 분리하거나 병합할 수 있게 되었다.

catkin_tools

catkin_make, catkin_make_isolated의 독립 사용에 불편함을 해결하고 Python으로 구성된 패키지도 관리할 수 있게 해주는 툴로 catkin_make의 부족한 부분을 제공하였다.

ament_tools

ROS 2의 ament_cmake 및 ament_python, 순수 CMake 패키지를 모두 지원하는 툴로 catkin_make, catkin_make_isolated, catkin_tools 모두의 기능을 사용할 수 있으며 ROS 2 Bouncy 버전 이전까지 사용되었다.

colcon

ROS 1과 ROS 2 모두를 지원하기 위하여 통합된 빌드 툴로서 소개되었으며 ROS 2 Bouncy 이후 ROS 2의 기본 빌드 툴로 사용 중에 있다.

6.2.3. 패키지 빌드하기

ROS 2 특정 패키지 또는 전체 패키지를 빌드할 때에는 colcon 빌드 툴을 사용한다. 사용 방법은 매우 간단한 편인데 우선 소스 코드가 있는 workspace로 이동하고 colcon build 명령어로 전체를 빌드하게 된다. 여기서 빌드 옵션을 추가하여 사용하는게 일반적인데 특정 패키지만 선택하여 빌드하고자 할 때에는 `--packages-select` 옵션을 이용하고 symlink를 이용하려면 `--symlink-install` 옵션을 붙여주면 된다.

경험상 가장 많이 사용하게 되는 빌드 명령어는 아래와 같다. 첫 번째 명령어가 전체 패키지를 빌드할 때 사용되며 두 번째 명령어는 명령어 마지막에 특정 패키지 이름을 기재하여 그 해당 패키지만 빌드할 때 사용한다.

```
$ cd ~/colcon_ws && colcon build --symlink-install  
$ cd ~/colcon_ws && colcon build --symlink-install --packages-select [패키지 이름]
```

ROS 2에서는 빌드 관련 내용들이 모두 변경되면서 빌드 옵션에도 새로운 변화[3]가 생겼다. 그중 사용하면서 가장 좋았던 3 가지를 꼽자면 아래와 같다.

우선 `Multiple workspace`이다. 이는 ROS 1에서는 `catkin_ws`와 같이 특정 워크스페이스를 확보하고 하나의 워크스페이스에서 모든 작업을 다 했는데 ROS 2에서는 복수의 독립된 워크스페이스를 사용할 수 있어서 작업 목적 및 패키지 종류별로 관리할 수 있게 되었다.

둘째는 `No non-isolated build`이다. ROS 1에서는 하나의 CMake 파일로 여러 개의 패키지를 동시에 빌드 할 수 있었다. 이렇게 하면 빌드 속도가 빨라지지만 모든 패키지의 종속성에 신경을 많이 써야 하고 빌드 순서가 매우 중요하게 된다. 또한 모든 패키지가 동일 네임스페이스 사용하게 되므로 이름에서 충돌이 발생할 수 있었다. ROS 2에서는 이전 빌드 시스템인 catkin에서 일부 기능으로 사용되었던 `catkin_make_isolated` 형태와 같은 격리 빌드만을 지원함으로써 모든 패키지를 별도로 빌드하게 되었다. 이 기능 변화를 통해 설치용 폴더를 분리하거나 병합할 수 있게 되었다.

셋째는 `No devel space`이다. catkin은 패키지를 빌드 한 후 devel이라는 폴더에 코드를 저장한다. 이 폴더는 패키지를 설치할 필요 없이 패키지를 사용할 수 있는 환경을 제공한다. 이를 통해 파일 복사를 피하면서 사용자는 파이썬 코드를 편집하고 즉시 코드 실행할 수 있었다. 단 이러한 기능은 매우 편리한 기능이지만 패키지를 관리하는 측면에서 복잡성을 크게 증가시켰다. 이에 ROS 2에서는 패키지를 빌드 한 후 설치해야 패키지를 사용할 수 있도록 바뀌었다. 단 쉬운 사용성도 고려하여 colcon

사용 시에 `colcon build --symlink-install` 와 같은 옵션[8]을 사용하여 심벌릭 링크 설치의 선택적 기능을 사용하여 동일한 이점을 제공하고 있다.

6.3. Simple Publisher / Subscriber 작성하기(C++)

프로그래밍 언어를 배울 때 처음에 등장하는 Hello World는 화면에 Hello World라는 문구를 출력하는 것으로 시작한다. ROS의 Hello World 또한 다르지 않지만 출력보다는 메시지 전송에 더 초점을 둔다. C++ 언어로 ROS 2의 가장 간단한 구조의 토픽(topic) 퍼블리셔(publisher)와 서브스크라이버(subscriber)를 작성하고 동작시켜 보겠다.

6.3.1. 패키지 생성

ROS 2 패키지 생성 명령어는 다음과 같다. `ros2 pkg create` 명령어를 사용하고 그 뒤에 옵션을 붙여 주게 된다. 참고로 하기 명령어를 실행하는 폴더 위치는 ~/colcon_ws/src 폴더이다.

```
$ ros2 pkg create [패키지이름] --build-type [빌드 타입] --dependencies [의존하는 패키지 1] [  
의존하는 패키지 n]  
$ cd ~/colcon_ws/src/  
$ ros2 pkg create my_first_ros_rclcpp_pkg --build-type ament_cmake --dependencies rclcpp  
std_msgs
```

앞의 명령어에서 의존하는 패키지로 'rclcpp'과 'std_msgs'를 옵션으로 달아주었다. ROS의 표준 메시지 패키지인 std_msgs 와 ROS에서 C++을 사용하기 위한 클라이언트 라이브러리 rclcpp 를 사용하겠다는 것으로 패키지 생성에 앞서 미리 설치해야 한다는 의미이다. 이러한 의존하는 패키지 설정은 패키지를 생성할 때 지정할 수도 있지만, 생성한 다음 package.xml 에서 직접 입력해도 된다.

패키지를 생성하였다면 '~/colcon_ws/src'에 'my_first_ros_rclcpp_pkg' 패키지 폴더와 ROS 패키지가 갖추어야 할 기본 내부 폴더 그리고 package.xml 파일들이 생성된다. ament_cmake 이나 ament_python 이나에 따라 기본 구성 파일 시스템이 좀 상이한데 기본적으로는 아래와 같이 구성된다.

```
.  
└── include  
    └── my_first_ros_rclcpp_pkg  
└── src  
└── CMakeLists.txt  
└── package.xml  
  
3 directories, 2 files
```

6.3.2. Package.xml 파일 수정

패키지 설정 파일 (package.xml)은 사용할 RCL(ROS 2 client libraries)에 따라 달라지는데 C++ 이라면 build_type 으로 ament_cmake 이 사용되고 Python 이라면 ament_python 으로 설정하면

된다. 그 이외에는 각기 다른 개발 환경에 맞춘 의존성 패키지 설정을 해주면 여느 패키지이나 대동소이하다.

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd"
schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>my_first_ros_rclcpp_pkg</name>
  <version>0.0.1</version>
  <description>ROS 2 rclcpp basic package for the ROS 2 seminar</description>
  <maintainer email="pyo@robotis.com">Pyo</maintainer>
  <license>Apache License 2.0</license>
  <author>Mikael Arguedas</author>
  <author>Morgan Quigley</author>
  <author email="jacob@openrobotics.org">Jacob Perron</author>
  <author email="pyo@robotis.com">Pyo</author>

  <buildtool_depend>ament_cmake</buildtool_depend>

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

  <test_depend>ament_lint_auto</test_depend>
  <test_depend>ament_lint_common</test_depend>

  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
```

6.3.3. CMakeLists.txt

빌드 설정 파일 (CMakeLists.txt)에서의 주의점은 의존성 패키지의 설정과 빌드 및 설치 관련 설정이다. 이 패키지에서는 빌드시에 필요한 ament_cmake 패키지, 클라이언트 라이브러리 rclcpp 패키지, 메시지 인터페이스 std_msgs 패키지가 의존성 패키지로 사용됨을 명시하였다. 그리고 `helloworld_publisher`과 `helloworld_subscriber` 실행 파일의 빌드 및 설치를 위한 설정등이 포함되어 있다.

```
# Set minimum required version of cmake, project name and compile options
cmake_minimum_required(VERSION 3.5)
project(my_first_ros_rclcpp_pkg)

if(NOT CMAKE_CXX_STANDARD)
  set(CMAKE_CXX_STANDARD 14)
endif()

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()

# Find dependencies
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)

# Build
add_executable(helloworld_publisher src/helloworld_publisher.cpp)
ament_target_dependencies(helloworld_publisher rclcpp std_msgs)
```

```

add_executable(helloworld_subscriber src/helloworld_subscriber.cpp)
ament_target_dependencies(helloworld_subscriber rclcpp std_msgs)

# Install
install(TARGETS
    helloworld_publisher
    helloworld_subscriber
    DESTINATION lib/${PROJECT_NAME})

# Test
if(BUILD_TESTING)
    find_package(ament_lint_auto REQUIRED)
    ament_lint_auto_find_test_dependencies()
endif()

# Macro for ament package
ament_package()

```

6.3.4. Hello_word_publisher.cpp

퍼블리셔 노드의 C++ 코드는 `~/colcon_ws/src/my_first_ros_rclcpp_pkg/src/` 폴더에 `helloworld_publisher.cpp`라는 이름으로 소스 코드 파일을 직접 생성하여 넣어주고 퍼블리셔 노드의 전체 코드는 다음과 같이 작성하면 된다.

```

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

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

using namespace std::chrono_literals;

class HelloworldPublisher : public rclcpp::Node
{
public:
    HelloworldPublisher()
        : Node("helloworld_publisher"), count_(0)
    {
        auto qos_profile = rclcpp::QoS(rclcpp::KeepLast(10));
        helloworld_publisher_ = this->create_publisher<std_msgs::msg::String>(
            "helloworld", qos_profile);
        timer_ = this->create_wall_timer(
            1s, std::bind(&HelloworldPublisher::publish_helloworld_msg, this));
    }

private:
    void publish_helloworld_msg()
    {
        auto msg = std_msgs::msg::String();
        msg.data = "Hello World: " + std::to_string(count_++);
        RCLCPP_INFO(this->get_logger(), "Published message: '%s'", msg.data.c_str());
        helloworld_publisher_->publish(msg);
    }
    rclcpp::TimerBase::SharedPtr timer_;
    rclcpp::Publisher<std_msgs::msg::String>::SharedPtr helloworld_publisher_;
}

```

```

size_t count_;
};

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

```

첫 구절은 include 및 namespace 구문이다. 코드에서 사용되는 std 계열의 헤더를 우선 선언하고 있으며, 이어서 rclcpp 의 Node 클래스를 사용하기 위한 rclcpp.hpp 헤더파일과 퍼블리시하는 메시지의 타입인 String 메시지 인터페이스를 사용하고자 string.hpp 헤더파일을 포함시켰다. chrono_literals 은 추후에 500ms, 1s 과 같이 시간을 가식성을 높인 문자로 표현하기 위하여 namespace 를 사용할 수 있도록 선언하였다.

```

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

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

using namespace std::chrono_literals;

```

이 노드의 메인 클래스는 HelloworldPublisher 으로 rclcpp 의 Node 클래스를 상속하여 사용할 예정이다.

```
class HelloworldPublisher : public rclcpp::Node
```

다음은 클래스 생성자의 정의로 `Node("helloworld_publisher")`, count_(0) 와 같이 Node 클래스의 생성자를 호출하고 노드 이름을 `helloworld_publisher` 으로 지정하였다. count_ 변수는 0 으로 초기화 한다.

그 다음 퍼블리셔의 QoS 설정을 위하여 `rclcpp::QoS(rclcpp::KeepLast(10))` 과 같이 기본 QoS 에서 KeepLast 형태로 `depth` 를 `10` 으로 설정하여 통신 상태가 원활하지 못한 상황 등 예기치 못한 경우 퍼블리시할 데이터를 버퍼에 10 개까지 저장하라는 설정이다.

그 다음으로는 Node 클래스의 create_publisher 함수를 이용하여 `helloworld_publisher_` 라는 퍼블리셔를 설정하고 있다. 매개변수로는 토픽에 사용할 토픽 메시지 타입과 토픽의 이름, QoS 설정을 기입하도록 되어 있으며 여기서는 토픽 메시지 타입으로 `String` , 토픽 이름으로 `helloworld` , QoS 설정으로 좀전에 설정한 `qos_profile` 으로 설정하였다.

마지막으로 Node 클래스의 create_wall_timer 함수를 이용하여 콜백함수를 수행하는 구문인데 첫번째 매개변수는 period 으로 1s(1 초)로 설정하였다. 이는 1 초마다 지정한 콜백함수를 실행하라는 것으로 아래 코드와 같이 설정하면 1 초마다 publish_helloworld_msg 함수를 실행하게 된다.

```

public:
HelloworldPublisher()
: Node("helloworld_publisher"), count_(0)
{

```

```

auto qos_profile = rclcpp::QoS(rclcpp::KeepLast(10));
helloworld_publisher_ = this->create_publisher<std_msgs::msg::String>(
    "helloworld", qos_profile);
timer_ = this->create_wall_timer(
    1s, std::bind(&HelloworldPublisher::publish_helloworld_msg, this));
}

```

다음은 위에서 지정한 콜백함수인 `publish_helloworld_msg` 함수이다. 퍼블리시할 메시지는 `String` 타입으로 `msg`이라는 이름으로 선언하였으며 보낼 메시지는 `msg.data`에 저장하게 되는데 여기서는 `Hello World: 1`과 같이 매번 콜백함수가 실행될때마다 1씩 증가하는 `count_` 값을 문자열에 포함시켜 `publish` 함수를 통해 퍼블리시하게 된다. `RCLCPP_INFO` 함수는 콘솔창에 출력하는 함수로 로거의 종류에 따라 `RCLCPP_DEBUG`, `RCLCPP_INFO`, `RCLCPP_WARN`, `RCLCPP_ERROR`, `RCLCPP_FATAL`과 같이 5 가지 종류가 있다. 일반적인 정보 전달에는 `RCLCPP_INFO`를 사용하고 있기에 `RCLCPP_INFO` 함수를 통해 현재 퍼블리시되는 메시지를 콘솔창에 출력시키는 구문을 마지막에 넣어주었다. `RCLCPP_XXXX` 계열의 함수는 프로그래밍에서 흔히 사용되는 `print` 함수라고 생각하면 이해하기 편할 것이다. 이하 클래스에서 `private` 변수로 사용되는 `timer_`, `helloworld_publisher_`, `count_`을 선언하였다.

참고로 콜백함수의 구현에는 member function, lambda, local function 방법이 있는데 이 예제 코드에서는 member function 방식을 택하였다. 시퀀스에 의해 처리되는 local function 방법은 잘 사용되지 않고 member function 또는 lambda 방식이 많이 사용된다. 자신이 좋아하는 구현 방법이 있다면 그 방법으로 작성하도록 하자.

```

private:
void publish_helloworld_msg()
{
    auto msg = std::msgs::msg::String();
    msg.data = "Hello World: " + std::to_string(count_++);
    RCLCPP_INFO(this->get_logger(), "Published message: '%s'", msg.data.c_str());
    helloworld_publisher_->publish(msg);
}
rclcpp::TimerBase::SharedPtr timer_;
rclcpp::Publisher<std::msgs::msg::String>::SharedPtr helloworld_publisher_;
size_t count_;

```

만약 `lambda`로 구현하고자 한다면 `publish_helloworld_msg` 함수를 삭제하고 위의 `HelloworldPublisher` 클래스 생성자 구문의 `timer_ = this->create_wall_timer()` 함수에 람다 표현식을 추가하면 된다. 경험적으로 콜백함수 부분이 비교적 간단한 구현 내용이면 람다 표현식을 쓰고 구현이 복잡하고 양이 많다면 멤버 함수 방식이 디버깅하기 편했다.

```

timer_ = this->create_wall_timer(
    1s,
    [this]() -> void
    {
        auto msg = std::msgs::msg::String();
        msg.data = "Hello World2: " + std::to_string(count_++);
        RCLCPP_INFO(this->get_logger(), "Published message: '%s'", msg.data.c_str());
        helloworld_publisher_->publish(msg);
    }
);

```

마지막은 `main` 함수로 `rclcpp::init`를 이용하여 초기화하고 위에서 작성한 `HelloworldPublisher` 클래스를 `node`라는 이름으로 생성한 다음 `rclpy::spin` 함수를 이용하여 생성한 노드를 `spin` 시켜

지정된 콜백함수가 실행될 수 있도록 하고 있다. 종료 `Ctrl + c`와 같은 인터럽트 시그널 예외 상황에서는 rclcpp::shutdown 함수로 노드를 소멸하고 프로세스를 종료하게 된다.

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

6.3.5. Helloworld_subscriber.cpp

서브스크라이버 노드의 C++ 코드도 퍼블리셔 노드와 마찬가지로 `~/colcon_ws/src/my_first_ros_rcl_cpp_pkg/src/` 폴더에 `helloworld_subscriber.cpp`라는 이름으로 소스 코드 파일을 직접 생성하여 넣어주고 서브스크라이버 노드의 전체 코드는 다음과 같이 작성하면 된다.

```
#include <functional>
#include <memory>

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

using std::placeholders::_1;

class HelloworldSubscriber : public rclcpp::Node
{
public:
    HelloworldSubscriber()
    : Node("Helloworld_subscriber")
    {
        auto qos_profile = rclcpp::QoS(rclcpp::KeepLast(10));
        helloworld_subscriber_ = this->create_subscription<std_msgs::msg::String>(
            "helloworld",
            qos_profile,
            std::bind(&HelloworldSubscriber::subscribe_topic_message, this, _1));
    }

private:
    void subscribe_topic_message(const std_msgs::msg::String::SharedPtr msg) const
    {
        RCLCPP_INFO(this->get_logger(), "Received message: '%s'", msg->data.c_str());
    }
    rclcpp::Subscription<std_msgs::msg::String>::SharedPtr helloworld_subscriber_;
};

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

첫 구절은 include 및 namespace 구문으로 퍼블리셔 노드와 비슷하다. placeholders 클래스는 bind 함수의 대체자 역할을 위하여 _1로 선언하였다.

```
#include <functional>
#include <memory>

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

using std::placeholders::_1;
```

이 노드의 메인 클래스는 HelloworldSubscriber으로 rclcpp 의 Node 클래스를 상속하여 사용할 예정이다.

```
class HelloworldSubscriber : public rclcpp::Node
```

다음은 클래스 생성자의 정의로 `Node("Helloworld_subscriber")`과 같이 Node 클래스의 생성자를 호출하고 노드 이름을 `Helloworld_subscriber`으로 지정하였다.

그 다음 퍼블리셔의 QoS 설정을 위하여 `rclcpp::QoS(rclcpp::KeepLast(10))`과 같이 기본 QoS에서 KeepLast 형태로 `depth`를 `10`으로 설정하여 통신 상태가 원활하지 못한 상황 등 예기치 못한 경우 서브스크라이브 데이터를 버퍼에 10 개까지 저장하라는 설정이다.

그 다음으로는 Node 클래스의 create_subscription 함수를 이용하여 `helloworld_subscriber_`라는 서브스크라이버를 설정하고 있다. 매개변수로는 토픽에 사용할 토픽 메시지 타입과 토픽의 이름, QoS 설정, 수신받은 메시지를 처리할 콜백함수를 기입하도록 되어 있다. 여기서는 토픽 메시지 타입으로 `String`, 토픽 이름으로 `helloworld`, QoS 설정으로 좀전에 설정한 `qos_profile`, 콜백함수는 `subscribe_topic_message`으로 설정하였다.

```
public:
    HelloworldSubscriber()
    : Node("Helloworld_subscriber")
    {
        auto qos_profile = rclcpp::QoS(rclcpp::KeepLast(10));
        helloworld_subscriber_ = this->create_subscription<std_msgs::msg::String>(
            "helloworld",
            qos_profile,
            std::bind(&HelloworldSubscriber::subscribe_topic_message, this, _1));
    }
```

다음은 위에서 지정한 콜백함수인 `subscribe_topic_message` 함수이다. 서브스크라이브한 메시지는 String 타입으로 msg이라는 이름을 사용하며 받은 메시지는 msg.data에 저장하게 되어 있다. 여기서는 `Hello World: 1`과 같은 메시지를 서브스크라이브하게 된다. 이 msg.data 를 RCLCPP_INFO 함수를 이용하여 서브스크라이브된 메시지를 콘솔창에 출력시키는 구문을 마지막에 넣어주었다. 이하 클래스에서 private 변수로 사용되는 `helloworld_subscriber_`를 선언하였다.

```
private:
    void subscribe_topic_message(const std_msgs::msg::String::SharedPtr msg) const
    {
        RCLCPP_INFO(this->get_logger(), "Received message: '%s'", msg->data.c_str());
    }
    rclcpp::Subscription<std_msgs::msg::String>::SharedPtr helloworld_subscriber_;
```

마지막은 main 함수로 HelloworldSubscriber을 node로 선언하여 사용한다는 것 이외에는 위에서 설명한 퍼블리셔 노드의 main 함수와 동일하다.

```

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

```

6.3.6. 빌드 및 실행

ROS 2 특정 패키지 또는 전체 패키지를 빌드할 때에는 colcon 빌드 툴을 사용한다. 사용 방법은 매우 간단한 편인데 우선 소스 코드가 있는 workspace로 이동하고 colcon build 명령어로 전체를 빌드하게 된다. 여기서 빌드 옵션을 추가하여 사용하는게 일반적인데 특정 패키지만 선택하여 빌드하고자 할 때에는 `--packages-select` 옵션을 이용하고 symlink를 이용하려면 `--symlink-install` 옵션을 붙여주면 된다.

(워크스페이스내의 모든 패키지 빌드하는 방법)

```
$ cd ~/colcon_ws && colcon build --symlink-install
```

(특정 패키지만 빌드하는 방법)

```
$ cd ~/colcon_ws && colcon build --symlink-install --packages-select [패키지 이름 1] [패키지 이름 2] [패키지 이름 N]
```

(특정 패키지 및 의존성 패키지를 함께 빌드하는 방법)

```
$ cd ~/colcon && colcon build --symlink-install --packages-up-to [패키지 이름]
```

위에서 작성해둔 my_first_ros_rclcpp_pkg 패키지만 빌드하려면 하기와 같은 명령어를 통해 가능하다.

```

$ cd ~/colcon_ws
$ colcon build --symlink-install --packages-select my_first_ros_rclcpp_pkg
Starting >>> my_first_ros_rclcpp_pkg
Finished <<< my_first_ros_rclcpp_pkg [2.53s]

```

Summary: 1 package finished [2.76s]

특정 패키지의 첫 빌드 때에는 빌드 후에 하기 명령어와 같이 환경 설정 파일을 불러와서 실행 가능한 패키지의 노드 설정들을 해줘야 빌드된 노드를 실행할 수 있으니 아래와 같이 실행해주자.

```
. ~/colcon_ws/install/local_setup.bash
```

각 노드의 실행은 `ros2 run` 명령어를 통해 아래와 같이 실행하면 된다.

```

$ ros2 run my_first_ros_rclcpp_pkg helloworld_subscriber
[INFO]: Received message: 'Hello World: 0'
[INFO]: Received message: 'Hello World: 1'
[INFO]: Received message: 'Hello World: 2'
[INFO]: Received message: 'Hello World: 3'
[INFO]: Received message: 'Hello World: 4'
[INFO]: Received message: 'Hello World: 5'

```

```

$ ros2 run my_first_ros_rclcpp_pkg helloworld_publisher
[INFO]: Published message: 'Hello World: 0'
[INFO]: Published message: 'Hello World: 1'
[INFO]: Published message: 'Hello World: 2'
[INFO]: Published message: 'Hello World: 3'

```

```
[INFO]: Published message: 'Hello World: 4'  
[INFO]: Published message: 'Hello World: 5'
```

6.4. Simple Publisher / Subscriber 작성하기(Python)

C++ 언어로 ROS 2 의 가장 간단한 구조의 토픽(topic) 퍼블리셔(publisher)와 서브스크라이버(subscriber)를 Python 언어로 작성하고 동작시켜보자.

6.4.1. 패키지 생성

ROS 2 패키지 생성 명령어는 다음과 같다. `ros2 pkg create` 명령어를 사용하고 그 뒤에 옵션을 붙여 주게 된다.

```
$ ros2 pkg create [패키지이름] --build-type [빌드 타입] --dependencies [의존하는패키지 1] [  
의존하는패키지 n]  
$ cd ~/colcon_ws/src/  
$ ros2 pkg create my_first_ros_rclpy_pkg --build-type ament_python --dependencies rclpy  
std_msgs
```

앞의 명령어에서 의존하는 패키지로 'rclpy'와 'std_msgs'를 옵션으로 달아주었다. ROS 의 표준 메시지 패키지인 std_msgs 와 ROS 에서 Python 을 사용하기 위한 클라이언트 라이브러리 rclpy 를 사용하겠다는 것으로 패키지 생성에 앞서 미리 설치해야 한다는 의미이다. 이러한 의존하는 패키지 설정은 패키지를 생성할 때 지정할 수도 있지만, 생성한 다음 package.xml 에서 직접 입력해도 된다.

패키지를 생성하였다면 '~/colcon_ws/src'에 'my_first_ros_rclpy_pkg' 패키지 폴더와 ROS 패키지가 갖추어야 할 기본 내부 폴더 그리고 package.xml 파일들이 생성된다. ament_cmake 이나 ament_python 이나에 따라 기본 구성 파일 시스템이 좀 상이한데 기본적으로는 아래와 같이 구성된다.

```
.  
└── my_first_ros_rclpy_pkg  
    └── __init__.py  
└── resource  
    └── my_first_ros_rclpy_pkg  
└── test  
    ├── test_copyright.py  
    ├── test_flake8.py  
    └── test_pep257.py  
└── package.xml  
└── setup.cfg  
└── setup.py
```

3 directories, 8 files

6.4.2. Package.xml 작성

패키지 설정 파일 (package.xml)은 사용할 RCL(ROS 2 client libraries)에 따라 달라지는데 C++

이라면 build_type 으로 ament_cmake 이 사용되고 Python 이라면 ament_python 으로 설정하면 된다. 그 이외에는 각기 다른 개발 환경에 맞춘 의존성 패키지 설정을 해주면 여느 패키지이나 대동소이하다.

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd"
schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>my_first_ros_rclpy_pkg</name>
  <version>0.0.2</version>
  <description>ROS 2 rclpy basic package for the ROS 2 seminar</description>
  <maintainer email="pyo@robotis.com">Pyo</maintainer>
  <license>Apache License 2.0</license>
  <author email="mikael@osrfoundation.org">Mikael Arguedas</author>
  <author email="pyo@robotis.com">Pyo</author>

  <depend>rclpy</depend>
  <depend>std_msgs</depend>

  <test_depend>ament_copyright</test_depend>
  <test_depend>ament_flake8</test_depend>
  <test_depend>ament_pep257</test_depend>
  <test_depend>python3-pytest</test_depend>

  <export>
    <build_type>ament_python</build_type>
  </export>
</package>
```

6.4.3. Setup.py 작성

패키지 설정 파일 (setup.py)에서의 주의점은 entry_points 옵션의 console_scripts 키를 사용한 실행 파일 설정이다. 예를 들어 `helloworld_publisher`과 `helloworld_subscriber` 콘솔 스크립트는 각각 my_first_ros_rclpy_pkg.helloworld_publisher 모듈과 my_first_ros_rclpy_pkg.helloworld_subscriber 모듈의 main 함수가 호출되게 된다. 이를 통해 `ros2 run` 또는 `ros2 launch`를 이용하여 해당 스크립트를 실행시킬 수 있다.

```
from setuptools import find_packages
from setuptools import setup

package_name = 'my_first_ros_rclpy_pkg'

setup(
    name=package_name,
    version='0.0.2',
    packages=find_packages(exclude=['test']),
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    author='Mikael Arguedas, Pyo',
    author_email='mikael@osrfoundation.org, pyo@robotis.com',
    maintainer='Pyo',
```

```

maintainer_email='pyo@robotis.com',
keywords=['ROS'],
classifiers=[
    'Intended Audience :: Developers',
    'License :: OSI Approved :: Apache Software License',
    'Programming Language :: Python',
    'Topic :: Software Development',
],
description='ROS 2 rclpy basic package for the ROS 2 seminar',
license='Apache License, Version 2.0',
tests_require=['pytest'],
entry_points={
    'console_scripts': [
        'helloworld_publisher = my_first_ros_rclpy_pkg.helloworld_publisher:main',
        'helloworld_subscriber = my_first_ros_rclpy_pkg.helloworld_subscriber:main',
    ],
},
)

```

6.4.4. Setup.cfg 작성

패키지 환경 설정 파일 (setup.cfg)에서의 주의점은 `my_first_ros_rclpy_pkg`와 같이 패키지 이름을 기재해야한다는 것과 나중에 colcon를 이용하여 빌드하게 되면 `/home/[유저이름]/colcon_ws/install/_my_first_ros_rclpy_pkg/lib/my_first_ros_rclpy_pkg`와 같은 지정 폴더에 실행 파일이 생성된다는 점이다.

```

[develop]
script-dir=$base/lib/my_first_ros_rclpy_pkg
[install]
install-scripts=$base/lib/my_first_ros_rclpy_pkg

```

6.4.5. Helloworld_publisher.py

퍼블리셔	노드의	Python	코드는
`~/colcon_ws/src/my_first_ros_rclpy_pkg/my_first_ros_rclpy_		pkg/`	폴더에
`helloworld_publisher.py`라는 이름으로 소스 코드 파일을 직접 생성하여 넣어주고 퍼블리셔 노드의			
전체 코드는 다음과 같이 작성하면 된다.			

```

import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile
from std_msgs.msg import String

class HelloworldPublisher(Node):

    def __init__(self):
        super().__init__('helloworld_publisher')
        qos_profile = QoSProfile(depth=10)
        self.helloworld_publisher = self.create_publisher(String, 'helloworld', qos_profile)
        self.timer = self.create_timer(1, self.publish_helloworld_msg)
        self.count = 0

    def publish_helloworld_msg(self):

```

```

msg = String()
msg.data = 'Hello World: {}'.format(self.count)
self.helloworld_publisher.publish(msg)
self.get_logger().info('Published message: {}'.format(msg.data))
self.count += 1

def main(args=None):
    rclpy.init(args=args)
    node = HelloworldPublisher()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        node.get_logger().info('Keyboard Interrupt (SIGINT)')
    finally:
        node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()

```

첫 구절은 import 구문이다. rclpy 의 Node 클래스를 사용하여, 퍼블리셔의 QoS 설정을 위하여 QoSProfile 클래스를 사용한다. 그리고 퍼블리시하는 메시지의 타입은 std_msgs.msg 모듈의 String 메시지 인터페이스를 사용하고자 import 하였다.

```

import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile
from std_msgs.msg import String

```

이 노드의 메인 클래스는 HelloworldPublisher 으로 Node 클래스를 상속하여 사용할 예정이다.

```
class HelloworldPublisher(Node):
```

다음은 클래스 생성자의 정의로 `super().__init__('helloworld_publisher')` 를 이용하여 Node 클래스의 생성자를 호출하고 노드 이름을 `helloworld_publisher` 으로 지정하였다.

그 다음 퍼블리셔의 QoS 설정을 위하여 `QoSProfile` 호출하고 기본 `depth` 를 `10` 으로 설정하여 통신 상태가 원활하지 못한 상황 등 예기치 못한 경우 퍼블리시할 데이터를 버퍼에 10 개까지 저장하라는 설정이다.

그 다음으로는 Node 클래스의 `create_publisher` 함수를 이용하여 `helloworld_publisher` 라는 퍼블리셔를 설정하고 있다. 매개변수로는 토픽에 사용할 토픽 메시지 타입과 토픽의 이름, QoS 설정을 기입하도록 되어 있으며 여기서는 토픽 메시지 타입으로 `String` , 토픽 이름으로 `helloworld` , QoS 설정으로 좀전에 설정한 `qos_profile` 으로 설정하였다.

마지막으로 Node 클래스의 `create_timer` 함수를 이용하여 콜백함수를 수행하는 구문인데 첫번째 매개변수는 `timer_period_sec` 으로 1 을 설정하였다. 이는 1 초마다 지정한 콜백함수를 실행하라는 것으로 아래 코드와 같이 설정하면 1 초마다 `publish_helloworld_msg` 함수를 실행하게 된다. `count` 는 콜백함수에 사용되는 카운터 값이다.

```

def __init__(self):
    super().__init__('helloworld_publisher')
    qos_profile = QoSProfile(depth=10)
    self.helloworld_publisher = self.create_publisher(String, 'helloworld', qos_profile)
    self.timer = self.create_timer(1, self.publish_helloworld_msg)

```

```
    self.count = 0
```

다음은 위에서 지정한 콜백함수인 `publish_helloworld_msg` 함수이다. 퍼블리시할 메시지는 String 타입으로 `msg`이라는 이름으로 선언하였으며 보낼 메시지는 `msg.data`에 저장하게 되는데 여기서는 `Hello World: 1`과 같이 매번 콜백함수가 실행될때마다 1씩 증가하는 `count` 값을 문자열에 포함시켜 `publish` 함수를 통해 퍼블리시하게 된다. `get_logger` 함수는 콘솔창에 출력하는 함수로 로거의 종류에 따라 `debug`, `info`, `warning`, `error`, `fatal`와 같이 5 가지 종류가 있다. 일반적인 정보 전달에는 `info`를 사용하고 있기에 `info` 함수를 통해 현재 퍼블리시되는 메시지를 콘솔창에 출력시키는 구문을 마지막에 넣어주었다. `get_logger`는 프로그래밍에서 흔히 사용되는 `print` 함수라고 생각하면 이해하기 편할 것이다.

참고로 콜백함수의 구현에는 member function, lambda, local function 방법이 있는데 이 예제 코드에서는 member function 방식을 택하였다. 시퀀스에 의해 처리되는 local function 방법은 잘 사용되지 않고 member function 또는 lambda 방식이 많이 사용된다. 자신이 좋아하는 구현 방법이 있다면 그 방법으로 작성하도록 하자.

```
def publish_helloworld_msg(self):
    msg = String()
    msg.data = 'Hello World: {0}'.format(self.count)
    self.helloworld_publisher.publish(msg)
    self.get_logger().info('Published message: {0}'.format(msg.data))
    self.count += 1
```

마지막은 `main` 함수로 `rclpy.init`를 이용하여 초기화하고 위에서 작성한 `HelloworldPublisher` 클래스를 `node`라는 이름으로 생성한 다음 `rclpy.spin` 함수를 이용하여 생성한 노드를 spin 시켜 지정된 콜백함수가 실행될 수 있도록 하고 있다. 종료 `Ctrl + c`와 같은 인터럽트 시그널 예외 상황에서는 `node`를 소멸시키고 `rclpy.shutdown` 함수로 노드를 종료하게 된다.

```
def main(args=None):
    rclpy.init(args=args)
    node = HelloworldPublisher()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        node.get_logger().info('Keyboard Interrupt (SIGINT)')
    finally:
        node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()
```

6.4.6. Helloworld_subscriber.py

서브스크라이버 노드의 Python 코드도 퍼블리셔 노드와 마찬가지로 `~/colcon_ws/src/my_first_ros_rclpy_pkg/my_first_ros_rclpy_pkg/` 폴더에 `helloworld_subscriber.py`라는 이름으로 소스 코드 파일을 직접 생성하여 넣어주고 서브스크라이버 노드의 전체 코드는 다음과 같이 작성하면 된다.

```
import rclpy
from rclpy.node import Node
```

```

from rclpy.qos import QoSProfile
from std_msgs.msg import String

class HelloworldSubscriber(Node):

    def __init__(self):
        super().__init__('Helloworld_subscriber')
        qos_profile = QoSProfile(depth=10)
        self.helloworld_subscriber = self.create_subscription(
            String,
            'helloworld',
            self.subscribe_topic_message,
            qos_profile)

    def subscribe_topic_message(self, msg):
        self.get_logger().info('Received message: {}'.format(msg.data))

def main(args=None):
    rclpy.init(args=args)
    node = HelloworldSubscriber()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        node.get_logger().info('Keyboard Interrupt (SIGINT)')
    finally:
        node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()

```

첫 구절은 import 구문으로 퍼블리셔 노드와 완전히 동일하다.

```

import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile
from std_msgs.msg import String

```

이 노드의 메인 클래스는 `HelloworldSubscriber` 으로 `Node` 클래스를 상속하여 사용할 예정이다.

```
class HelloworldSubscriber(Node):
```

다음은 클래스 생성자의 정의로 `super().__init__('Helloworld_subscriber')` 를 이용하여 `Node` 클래스의 생성자를 호출하고 노드 이름을 `Helloworld_subscriber` 으로 지정하였다.

그 다음 서브스크라이버의 QoS 설정을 위하여 `QoSProfile` 호출하고 기본 `depth` 를 `10` 으로 설정하여 통신 상태가 원활하지 못한 상황 등 예기치 못한 경우 서브스크라이브 데이터를 버퍼에 10 개까지 저장하라는 설정이다.

그 다음으로는 `Node` 클래스의 `create_subscription` 함수를 이용하여 `helloworld_subscriber` 라는 서브스크라이버를 설정하고 있다. 매개변수로는 토픽에 사용할 토픽 메시지 타입과 토픽의 이름, 수신받은 메시지를 처리할 콜백함수과 QoS 설정을 기입하도록 되어 있다. 여기서는 토픽 메시지 타입으로 `String`, 토픽 이름으로 `helloworld`, 콜백함수는 `subscribe_topic_message`, QoS 설정으로 좀전에 설정한 `qos_profile` 으로 설정하였다.

```
def __init__(self):
```

```

super().__init__('Helloworld_subscriber')
qos_profile = QoSProfile(depth=10)
self.helloworld_subscriber = self.create_subscription(
    String,
    'helloworld',
    self.subscribe_topic_message,
    qos_profile)

```

다음은 위에서 지정한 콜백함수인 `subscribe_topic_message` 함수이다. 서브스크라이브한 메시지는 `String` 타입으로 `msg`이라는 이름을 사용하며 받은 메시지는 `msg.data`에 저장하게 되어 있다. 여기서는 `Hello World: 1`과 같은 메시지를 서브스크라이브하게 된다. 이 `msg.data`를 `get_logger`의 `info` 함수를 이용하여 서브스크라이브된 메시지를 콘솔창에 출력시키는 구문을 마지막에 넣어주었다.

```

def subscribe_topic_message(self, msg):
    self.get_logger().info('Received message: {}'.format(msg.data))

```

마지막은 `main` 함수로 `HelloworldSubscriber`을 `node`로 선언하여 사용한다는 것 이외에는 위에서 설명한 퍼블리셔 노드의 `main` 함수와 동일하다.

```

def main(args=None):
    rclpy.init(args=args)
    node = HelloworldSubscriber()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        node.get_logger().info('Keyboard Interrupt (SIGINT)')
    finally:
        node.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()

```

6.4.7. 빌드 및 실행

ROS 2 특정 패키지 또는 전체 패키지를 빌드할 때에는 `colcon` 빌드 툴을 사용한다. 사용 방법은 매우 간단한 편인데 우선 소스 코드가 있는 `workspace`로 이동하고 `colcon build` 명령어로 전체를 빌드하게 된다. 여기서 빌드 옵션을 추가하여 사용하는게 일반적인데 특정 패키지만 선택하여 빌드하고자 할 때에는 `--packages-select` 옵션을 이용하고 `symlink`를 이용하려면 `--symlink-install` 옵션을 붙여주면 된다.

(워크스페이스내의 모든 패키지 빌드하는 방법)
`$ cd ~/colcon_ws && colcon build --symlink-install`

(특정 패키지만 빌드하는 방법)
`$ cd ~/colcon_ws && colcon build --symlink-install --packages-select [패키지 이름 1] [패키지 이름 2] [패키지 이름 N]`

(특정 패키지 및 의존성 패키지를 함께 빌드하는 방법)
`$ cd ~/colcon_ws && colcon build --symlink-install --packages-up-to [패키지 이름]`

위에서 작성해둔 `my_first_ros_rclpy_pkg` 패키지만 빌드하려면 하기와 같은 명령어를 통해 가능하다.

```

$ cd ~/colcon_ws
$ colcon build --symlink-install --packages-select my_first_ros_rclpy_pkg
Starting >>> my_first_ros_rclpy_pkg
Finished <<< my_first_ros_rclpy_pkg [0.66s]

```

```
Summary: 1 package finished [0.87s]
```

특정 패키지의 첫 빌드 때에는 빌드 후에 하기 명령어와 같이 환경 설정 파일을 불러와서 실행 가능한 패키지의 노드 설정들을 해줘야 빌드된 노드를 실행할 수 있으니 아래와 같이 실행해주자.

```
. ~/colcon_ws/install/local_setup.bash
```

각 노드의 실행은 `ros2 run` 명령어를 통해 아래와 같이 실행하면 된다.

```
$ ros2 run my_first_ros_rclpy_pkg helloworld_subscriber
[INFO]: Received message: Hello World: 0
[INFO]: Received message: Hello World: 1
[INFO]: Received message: Hello World: 2
[INFO]: Received message: Hello World: 3
[INFO]: Received message: Hello World: 4
[INFO]: Received message: Hello World: 5
```

```
$ ros2 run my_first_ros_rclpy_pkg helloworld_publisher
[INFO]: Published message: Hello World: 0
[INFO]: Published message: Hello World: 1
[INFO]: Published message: Hello World: 2
[INFO]: Published message: Hello World: 3
[INFO]: Published message: Hello World: 4
[INFO]: Published message: Hello World: 5
```

6.5. ROS2 인터페이스 구현하기

6.5.1. ROS2 인터페이스 신규 작성

ROS 2 프로그래밍은 노드간의 메시지 통신을 위해 정수(integer), 부동 소수점(floating point), 불(boolean)같은 기본 인터페이스를 모아둔 std_msgs 이나 병진속도 x, y, z, 회전속도 x, y, z 를 표현할 수 있는 geometry_msgs 의 Twist.msg 인터페이스, 레이저 스캐닝 값을 담을 수 있는 sensor_msgs 의 LaserScan.msg 과 같은 이미 만들어 놓은 인터페이스를 사용하는 것이 일반적이다. 하지만 이러한 인터페이스들이 유저가 원하는 모든 정보를 담을 수는 없고 토픽 인터페이스 이외에 서비스나 액션 인터페이스는 매우 기본적인 인터페이스만 제공하기에 사용자가 원하는 정보의 형태가 아니라면 새로 만들어 써야 한다.

ROS 2 인터페이스는 이 인터페이스를 사용하려는 패키지에 포함시켜도 되지만 경험상 인터페이스으로만 구성된 패키지를 별도로 만들어 사용하는 것이 의존성면에서 관리하기 편하다. 예를 들어 A라는 패키지에서 a라는 인터페이스를 사용한다고 했을 때 a라는 인터페이스를 사용하는 B, C라는 패키지가 있다면 B 와 C 패키지는 a 인터페이스가 담긴 A라는 패키지를 통째로 의존하기 때문이다. 이러한 이유로 인터페이스는 별도의 독립된 패키지로 구성해야한다. (예: geometry_msgs, turtlebot3_msgs)

이번에 우리는 msg_srv_action_interface_example 이라는 이름의 패키지를 만들 것이고 이

인터페이스 전용 패키지에는 아래와 같이 msg 인터페이스 1개, srv 인터페이스 1개, action 인터페이스 1개를 포함시킬 것이다.

- ArithmeticArgument.msg
- ArithmeticOperator.srv
- ArithmeticChecker.action

6.5.2. 인터페이스 패키지 만들기

워크스페이스의 src 폴더로 이동 후 앞선 강좌에서 익힌 `ros2 pkg create` 명령어를 이용하여 msg_srv_action_interface_example이라는 패키지를 만들어주자. 그 뒤에 아래와 같이 인터페이스 파일을 담을 msg, srv, action 폴더를 생성해주자.

```
$ cd ~/robot_ws/src  
$ ros2 pkg create --build-type ament_cmake msg_srv_action_interface_example  
$ cd msg_srv_action_interface_example  
$ mkdir msg srv action
```

그 뒤 각 인터페이스 파일을 `ArithmeticArgument.msg`, `ArithmeticOperator.srv`, `ArithmeticChecker.action`이라는 이름으로 신규로 작성하여 각각 msg, srv, action 폴더에 파일을 만들어 넣어주자.

ROS 2 인터페이스 류의 파일명은 `CamelCased` 규칙을 따른다. 그 이유는 *.msg 및 *.srv 또는 *.action는 *.h(pp) 변환 후 인터페이스 타입으로 구조체 및 타입으로 사용되기 때문이다.

```
└── action  
    └── ArithmeticChecker.action  
└── msg  
    └── ArithmeticArgument.msg  
└── srv  
    └── ArithmeticOperator.srv
```

ArithmeticArgument.msg 생성

- 저장 위치: msg_srv_action_interface_example/msg

```
# Messages
builtin_interfaces/Time stamp
float32 argument_a
float32 argument_b
```

ArithmeticOperator.srv 생성

- 저장 위치: msg_srv_action_interface_example/srv

```
# Constants
int8 PLUS = 1
int8 MINUS = 2
int8 MULTIPLY = 3
int8 DIVISION = 4

# Request
int8 arithmetic_operator
---

# Response
float32 arithmetic_result
```

ArithmeticChecker.action 생성

- 저장 위치: msg_srv_action_interface_example/action

```
# Goal
float32 goal_sum
---

# Result
string[] all_formula
float32 total_sum
---

# Feedback
string[] formula
```

msg, srv, action 인터페이스 비교

	msg 인터페이스	srv 인터페이스	action 인터페이스
확장자	*.msg	*.srv	*.action
데이터	토픽 데이터 (data)	서비스 요청 (request) --- 서비스 응답 (response)	액션 목표 (goal) --- 액션 결과 (result) --- 액션 피드백 (feedback)
형식	fieldtype1 fieldname1 fieldtype2 fieldname2 fieldtype3 fieldname3	fieldtype1 fieldname1 fieldtype2 fieldname2 --- fieldtype3 fieldname3 fieldtype4 fieldname4	fieldtype1 fieldname1 fieldtype2 fieldname2 --- fieldtype3 fieldname3 fieldtype4 fieldname4 --- fieldtype5 fieldname5 fieldtype6 fieldname6
사용 예	[ArithmeticArgument.msg] builtin_interfaces/Time stamp float32 argument_a float32 argument_b	[ArithmeticOperator.srv] # Constants int8 PLUS = 1 int8 MINUS = 2 int8 MULTIPLY = 3 int8 DIVISION = 4 # Request int8 arithmetic_operator --- # Response float32 arithmetic_result	[ArithmeticChecker.action] # Goal float32 goal_sum --- # Result string[] all_formula float32 total_sum --- # Feedback string[] formula

6.5.3. Package.xml 파일 작성

msg_srv_action_interface_example 의 패키지 설정 파일(package.xml)은 아래와 같이 작성해주자. 물론 파일의 내용은 본인에게 맞게 원하는대로 변경해도 된다.

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd"
schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
<name>msg_srv_action_interface_example</name>
<version>0.2.0</version>
<description>
ROS 2 example for message, service and action interface
</description>
<maintainer email="passionvirus@gmail.com">Pyo</maintainer>
```

```

<license>Apache 2.0</license>
<author email="passionvirus@gmail.com">Pyo</author>
<author email="routiful@gmail.com">Darby Lim</author>
<buildtool_depend>ament_cmake</buildtool_depend>
<buildtool_depend>rosidl_default_generators</buildtool_depend>
<exec_depend>builtin_interfaces</exec_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
<member_of_group>rosidl_interface_packages</member_of_group>
<export>
  <build_type>ament_cmake</build_type>
</export>
</package>

```

일반적인 패키지와 다른 점은 아래와 같이 빌드 시에 DDS에서 사용되는 IDL(Interface Definition Language)생성과 관련한 `rosidl_default_generators`이 사용된다는 점과 실행 시에 `builtin_interfaces`와 `rosidl_default_runtime`이 사용된다는 점이다. 그 이외에는 일반적인 패키지의 설정과 동일하다. 인터페이스 전용 패키지를 만들 때에는 필수적인 의존성 패키지이니 기억해두자.

```

<buildtool_depend>rosidl_default_generators</buildtool_depend>
<exec_depend>builtin_interfaces</exec_depend>
<exec_depend>rosidl_default_runtime</exec_depend>

```

6.5.4. CMakeLists.txt

msg_srv_action_interface_example의 패키지의 빌드 설정 파일(CMakeLists.txt)은 아래와 같이 작성해주자.

```

#####
# Set minimum required version of cmake, project name and compile options
#####
cmake_minimum_required(VERSION 3.5)
project(msg_srv_action_interface_example)

if(NOT CMAKE_CXX_STANDARD)
  set(CMAKE_CXX_STANDARD 14)
endif()

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -Wextra -Wpedantic")
endif()

#####
# Find and load build settings from external packages
#####
find_package(ament_cmake REQUIRED)
find_package(builtin_interfaces REQUIRED)
find_package(rosidl_default_generators REQUIRED)

```

```

#####
# Declare ROS messages, services and actions
#####
set(msg_files
    "msg/ArithmetricArgument.msg"
)

set(srv_files
    "srv/ArithmetricOperator.srv"
)

set(action_files
    "action/ArithmetricChecker.action"
)

rosidl_generate_interfaces(${PROJECT_NAME}
    ${msg_files}
    ${srv_files}
    ${action_files}
    DEPENDENCIES builtin_interfaces
)

#####
# Macro for ament package
#####
ament_export_dependencies(rosidl_default_runtime)
ament_package()

```

일반적인 패키지와 다른 점은 아래와 같이 `set` 명령어로 msg, srv, action 파일을 지정해주고 `rosidl_generate_interfaces`에 해당 셋들을 기입해주면 끝이다.

```

set(msg_files
    "msg/ArithmetricArgument.msg"
)

set(srv_files
    "srv/ArithmetricOperator.srv"
)

set(action_files
    "action/ArithmetricChecker.action"
)

rosidl_generate_interfaces(${PROJECT_NAME}
    ${msg_files}
    ${srv_files}
    ${action_files}
    DEPENDENCIES builtin_interfaces
)

```

6.5.5. 빌드하기

```

$ cw
$ cbp msg_srv_action_interface_example

```

빌드한 후 빌드에 문제가 없다면 `~/robot_ws/install/msg_srv_action_interface_example` 폴더 안에 우리가 작성한 ROS 인터페이스를 사용하기 위한 파일들이 저장되게 된다. 예를 들어 C는 위한 h, C++을 위한 hpp, 파이썬 모듈, IDL 파일들이다. 빌드 후에 참고삼아 해당 폴더의 파일들을

확인해보도록 하자.

```
~/robot_ws/install/msg_srv_action_interface_example
└── include
    └── msg_srv_action_interface_example
        ├── action
        ├── msg
        └── srv
└── lib
    └── python3.6
        └── site-packages
            └── msg_srv_action_interface_example
                ├── action
                ├── msg
                └── srv
└── share
    └── msg_srv_action_interface_example
        ├── action
        ├── msg
        └── srv
```

6.6. ROS2 패키지 설계

6.6.1. ROS2 패키지 설계

ROS 와 연동되는 로봇 프로그램과 일반적인 로봇 프로그램과의 차이는 프로세스를 목적별로 나누어 노드 (node) 단위의 프로그램을 작성하고 노드와 노드간의 데이터 통신을 고려하여 설계해야 한다는 것이다. 이번에는는 ROS 2 의 토픽, 서비스, 액션 프로그래밍의 예제이면서 하나의 패키지로 토픽, 서비스, 액션이 따로 구동하는 것이 아닌 연동되어 구동하는 방식으로 설계하였다.

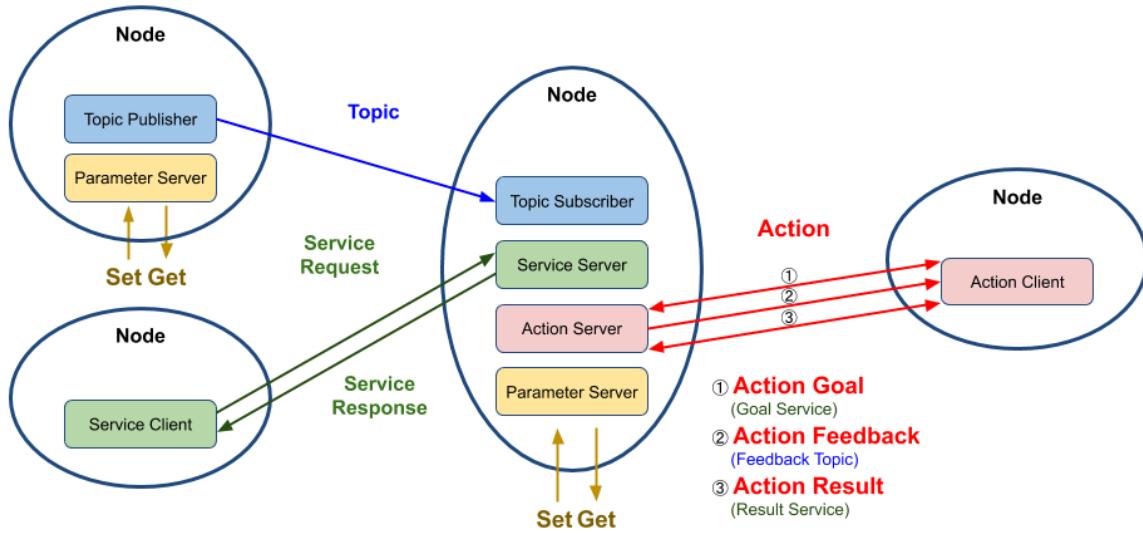


그림 73 패키지 구성

이렇게 설계한 패키지는 `topic_service_action_rclpy_example` 이라는 이름으로 패키지 명을 명명하였다. 이 패키지에서 각각의 노드와 토픽, 서비스, 액션도 고유의 이름을 가지고 있는데 이는 그림 74 와 같다. 그림 2 에서의 프로세스를 설명하자면 다음과 같다.

- argument node : `arithmetic_argument` 이라는 토픽 이름으로 현재 시간과 변수 a 와 b 를 퍼블리시한다.
- calculator node : 토픽이 생성된 시간과 변수 a 와 b 를 서브스크라이브한다.
- operator node : `arithmetic_operator` 이라는 서비스 이름으로 calculator 노드에게 연산자(+, -, *, /)를 서비스 요청값으로 보낸다.
- calculator node : 서브스크라이브하여 저장하고 있는 변수 a 와 b 를 operator 노드로부터 요청 값으로 받은 연산자를 이용하여 계산(a 연산자 b)하고 operator 노드에게 연산의 결괏값을 서비스 응답값으로 보낸다.
- checker node : 연산값의 합계의 한계치를 액션 목표값으로 전달한 후, calculator 노드는 이를 받은 후 부터의 연산 값을 합하고 액션 피드백으로 각 연산 계산식을 보낸다. 지정한 연산값의 목표 합계를 넘기면 액션 결괏값으로 최종 연산 합계를 보낸다.

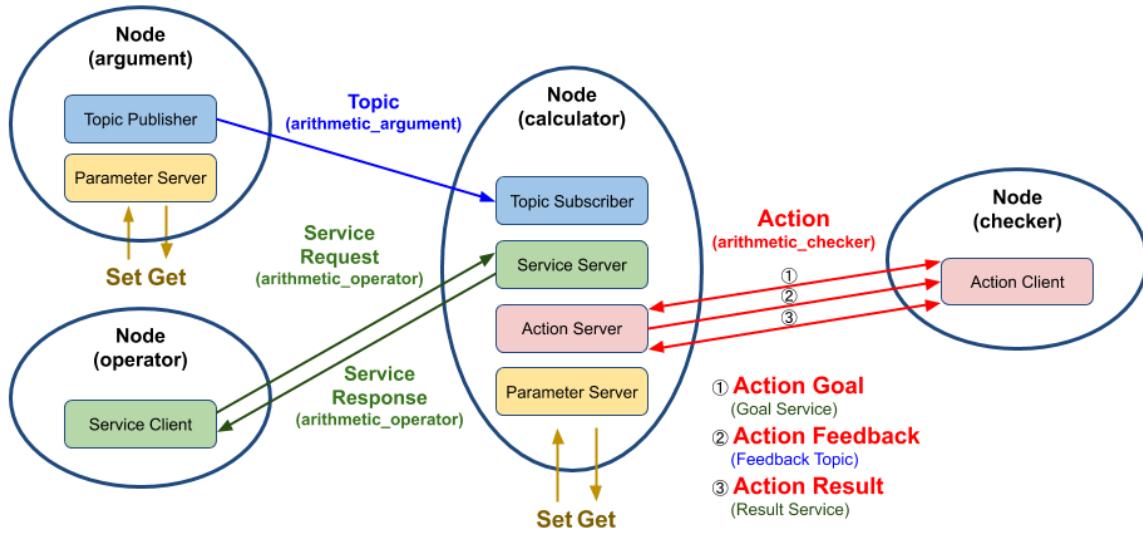


그림 74 패키지 구성(상세)

위에서 설명한 `topic_service_action_rclcpp_example` 패키지는 argument 노드, operator 노드, calculator 노드, checker 노드와 같이 4 개의 노드로 구성되어 있다. 해당 노드의 원본 코드는 참고자료로 올린 리포지토리에서 미리 확인하실 수 있으며 자세한 내용은 아래와 같이 이어지는 내용에서 자세히 설명할 예정이다.

6.6.2. Python 패키지 설계하기

package.xml

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd"
schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>topic_service_action_rclpy_example</name>
  <version>0.2.0</version>
  <description>ROS 2 rclpy example package for the topic, service, action</description>
  <maintainer email="passionvirus@gmail.com">Pyo</maintainer>
  <license>Apache License 2.0</license>
  <author email="passionvirus@gmail.com">Pyo</author>
  <author email="routiful@gmail.com">Darby Lim</author>
  <depend>rclpy</depend>
  <depend>std_msgs</depend>
  <depend>msg_srv_action_interface_example</depend>
  <test_depend>ament_copyright</test_depend>
  <test_depend>ament_flake8</test_depend>
  <test_depend>ament_pep257</test_depend>
  <test_depend>python3-pytest</test_depend>
  <export>
    <build_type>ament_python</build_type>
  </export>
```

```
</package>
```

대부분 이전 강좌에서 설명한 부분인데 이 패키지만의 특별한 설정은 6.5 에서 작성한 토픽, 서비스, 액션 인터페이스(패키지명: msg_srv_action_interface_example)를 사용할 예정이기에 의존성을 걸어두는 것이다. 그 이외에는 특별한 설정은 없다.

```
<depend>msg_srv_action_interface_example</depend>
```

setup.py

topic_service_action_rclpy_example 패키지의 파일 패키지 설정 파일 (setup.py)의 전문은 다음과 같다.

```
#!/usr/bin/env python3

import glob
import os

from setuptools import find_packages
from setuptools import setup

package_name = 'topic_service_action_rclpy_example'
share_dir = 'share/' + package_name

setup(
    name=package_name,
    version='0.2.0',
    packages=find_packages(exclude=['test']),
    data_files=[
        ('share/ament_index/resource_index/packages', ['resource/' + package_name]),
        (share_dir, ['package.xml']),
        (share_dir + '/launch', glob.glob(os.path.join('launch', '*.launch.py'))),
        (share_dir + '/param', glob.glob(os.path.join('param', '*.yaml'))),
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    author='Pyo, Darby Lim',
    author_email='passionvirus@gmail.com, routiful@gmail.com',
    maintainer='Pyo',
    maintainer_email='passionvirus@gmail.com',
    keywords=['ROS'],
    classifiers=[
        'Intended Audience :: Developers',
        'License :: OSI Approved :: Apache Software License',
        'Programming Language :: Python',
        'Topic :: Software Development',
    ],
    description='ROS 2 rclpy example package for the topic, service, action',
    license='Apache License, Version 2.0',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'argument = topic_service_action_rclpy_example.argument:main',
            'operator = topic_service_action_rclpy_example.operator:main',
            'calculator = topic_service_action_rclpy_example.calculator.main:main',
            'checker = topic_service_action_rclpy_example.checker.main:main',
        ],
    },
)
```

이 패키지는 여느 다른 파이썬 ROS 패키지와 다를 바 없는데 헷갈일 수 있는 두 가지만 집고 넘어가도록 하자.

data_files

첫번째는 `data_files` 설정이다. 이 패키지에서 사용되는 파일들을 기입하여 함께 배포하는데 `ROS`에서는 주로 `resource` 폴더 내에 있는 `ament_index`를 위한 패키지의 이름의 빈 파일이나 `package.xml`, `*.launch.py`, `*.yaml` 등을 기입한다. 이를 통해 빌드 후에 해당 파일들이 설치 폴더에 추가되게 된다. 흔히 "파이썬 ROS 패키지는 빌드하지 않아도 되지 않나?" 생각하기 쉬운데 파이썬 코드 이외에도 해당 패키지의 다양한 파일들을 설치 폴더에 담아야 하기에 빌드 후 심볼링크나 원본이 복사되어 사용된다. 즉, 파이썬 ROS 패키지가 파이썬 코드로 구성되어 있기는 하지만 빌드는 해야한다는 점을 잊지 않도록 하자.

이 패키지에서도 `arithmetic.launch.py` 이라는 런치 파일과 `arithmetic_config.yaml` 파라미터 파일이 사용된다. 이에 하기와 같이 해당 파일들의 설정도 필요한 것이다.

```
data_files=[  
    ('share/ament_index/resource_index/packages', ['resource/' + package_name]),  
    (share_dir, ['package.xml']),  
    (share_dir + '/launch', glob.glob(os.path.join('launch', '*launch.py'))),  
    (share_dir + '/param', glob.glob(os.path.join('param', '*yaml'))),  
],
```

entry_points

`entry_points`는 설치하여 사용할 실행 가능한 콘솔 스크립트의 이름과 호출 함수를 기입하도록 되어 있다. 우리는 4 개의 노드를 작성하고 `ros2 run` 과 같은 노드 실행 명령어를 통하여 각각의 노드를 실행할 예정이기에 하기와 같이 `entry_points`를 추가하도록 하자.

```
entry_points={  
    'console_scripts': [  
        'argument = topic_service_action_rclpy_example.arithmetic.argument:main',  
        'operator = topic_service_action_rclpy_example.arithmetic.operator:main',  
        'calculator = topic_service_action_rclpy_example.calculator.main:main',  
        'checker = topic_service_action_rclpy_example.checker.main:main',  
    ],  
},
```

소스코드 다운로드 및 빌드

소스 코드 다운로드 및 빌드는 하기와 같은 명령어로 진행하면 된다.

```
$ cd ~/robot_ws/src  
$ git clone https://github.com/robotpilot/ros2-seminar-examples.git  
$ cd ~/robot_ws && colcon build --symlink-install
```

빌드한 후 빌드에 문제가 없다면 `~/robot_ws/install/topic_service_action_rclpy_example` 폴더 안에 우리가 작성한 ROS 인터페이스를 사용하기 위한 파일들이 저장되게 된다. 그리고 하기 폴더에는 launch 폴더와 param 폴더가 생성되고 각각 런치파일 `arithmetic.launch.py` 파일과 파라미터 파일인 `arithmetic_config.yaml` 파일이 위치하게 된다.

```
~/robot_ws/install/topic_service_action_rclpy_example/share/  
topic_service_action_rclpy_example
```

6.6.3. Python 패키지 실행하기

토픽 서브스크라이버, 서비스 서버, 액션 서버 실행

제일 먼저 calculator 노드를 실행해보자. 이 노드는 이 패키지에서 토픽 서브스크라이버, 서비스 서버, 액션 서버 역할을 수행하는 노드로 실행 후 아무런 동작은 하고 있지 않은 대기 상태가 된다.

```
$ ros2 run topic_service_action_rclpy_example calculator
```

토픽 퍼블리셔 실행

다음으로 토픽 퍼블리셔 역할을 하는 argument 노드를 실행해보자. 그러면 터미널 창에 퍼블리시하고 있는 argument a, argument b 가 표시된다. 더불어 calculator 노드를 실행 시킨 터미널 창에서는 수신 받은 시간 정보와 argument a, argument b 가 표시된다.

```
$ ros2 run topic_service_action_rclpy_example argument  
[INFO]: Published argument a: 5.0  
[INFO]: Published argument b: 0.0  
[INFO]: Published argument a: 1.0  
[INFO]: Published argument b: 5.0  
$ ros2 run topic_service_action_rclpy_example calculator  
[INFO]: Subscribed at: builtin_interfaces.msg.Time(sec=1607301572, nanosec=687669256)  
[INFO]: Subscribed argument a: 5.0  
[INFO]: Subscribed argument b: 0.0  
[INFO]: Subscribed at: builtin_interfaces.msg.Time(sec=1607301573, nanosec=687576780)  
[INFO]: Subscribed argument a: 1.0  
[INFO]: Subscribed argument b: 5.0
```

서비스 클라이언트 실행

다음으로 서비스 클라이언트 역할을 하는 operator 노드를 실행해보자. 그러면 calculator 노드에게 랜덤으로 선택한 연산자(+, -, *, /)를 서비스 요청값으로 보내고 연산된 결과값을 받아 터미널 창에 표시한다. 계산식은 calculator 노드가 실행 중인 창에서 확인할 수 있다. 예를 들어 calculator는 변수 a, b 로 9 와 6 을 받았고 연산자로 곱셈(*)을 받았을 때 $9.0 * 6.0 = 54.0$ 이므로 operator는 54.0 를 표시하게 된다.

```
$ ros2 run topic_service_action_rclpy_example operator  
[INFO]: Result: 54.0  
[INFO]: Result: 0.625  
[INFO]: Subscribed at: builtin_interfaces.msg.Time(sec=1607379440, nanosec=275268255)  
[INFO]: Subscribed argument a: 9.0  
[INFO]: Subscribed argument b: 6.0  
[INFO]: 9.0 * 6.0 = 54.0  
[INFO]: Subscribed at: builtin_interfaces.msg.Time(sec=1607379441, nanosec=275234062)  
[INFO]: Subscribed argument a: 5.0  
[INFO]: Subscribed argument b: 8.0  
[INFO]: 5.0 / 8.0 = 0.625
```

액션 클라이언트 실행

마지막으로 checker 노드는 연산값의 합계의 한계치를 액션 목표값으로 전달한 후, calculator 노드는 이를 받은 후 부터의 연산 값을 합하고 액션 피드백으로 각 연산 계산식을 보낸다. 지정한 연산값의 목표 합계를 넘기면 액션 결괏값으로 최종 연산 합계를 보낸다.

```
$ ros2 run topic_service_action_rclpy_example checker
[INFO]: Action goal accepted.
[INFO]: Action feedback: ['7.0 + 3.0 = 10.0']
[INFO]: Action feedback: ['7.0 + 3.0 = 10.0', '8.0 + 1.0 = 9.0']
[INFO]: Action feedback: ['7.0 + 3.0 = 10.0', '8.0 + 1.0 = 9.0', '6.0 / 1.0 = 6.0']
[INFO]: Action feedback: ['7.0 + 3.0 = 10.0', '8.0 + 1.0 = 9.0', '6.0 / 1.0 = 6.0', '8.0 - 6.0 = 2.0']
[INFO]: Action feedback: ['7.0 + 3.0 = 10.0', '8.0 + 1.0 = 9.0', '6.0 / 1.0 = 6.0', '8.0 - 6.0 = 2.0', '6.0 * 6.0 = 36.0']
[INFO]: Action succeeded!
[INFO]: Action result(all formula): ['7.0 + 3.0 = 10.0', '8.0 + 1.0 = 9.0', '6.0 / 1.0 = 6.0', '8.0 - 6.0 = 2.0', '6.0 * 6.0 = 36.0']
[INFO]: Action result(total sum): 63.0
```

합계의 한계치는 기본으로 50으로 설정되어 있다. 이를 수정하여 실행하고 싶으면 아래와 같이 checker 노드를 실행시키면서 실행 인자로 `-g 100` 이라고 입력하면 GOAL_TOTAL_SUM이라는 인자로 100을 할당할 수 있다. 여기서 실행인자에 대한 이해가 실행 인자 프로그래밍에서 다를 예정이다.

```
$ ros2 run topic_service_action_rclpy_example checker -g 100
```

런치 파일 실행

참고로 argument 노드와 calculator 노드를 한번에 실행시키고자 한다면 아래와 같이 launch 파일인 arithmetic.launch.py를 실행하는 방법으로 두개의 노드를 동시에 실행 시킬 수 있다. launch 파일에 대한 자세한 설명은 추후 런치프로그래밍에서 자세히 알아볼 예정이기에 여기서는 실행 방법만 알아보았다.

```
$ ros2 launch topic_service_action_rclpy_example arithmetic.launch.py
```

6.6.4. C++ 패키지 설계하기

Package.xml

topic_service_action_rclcpp_example 패키지의 설정 파일(package.xml)은 아래와 같이 작성해주자.

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd"
schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>topic_service_action_rclcpp_example</name>
  <version>0.2.0</version>
  <description>ROS 2 rclcpp example package for the topic, service, action</description>
  <maintainer email="passionvirus@gmail.com">Pyo</maintainer>
  <license>Apache License 2.0</license>
```

```
<author email="passionvirus@gmail.com">Pyo</author>
<author email="routiful@gmail.com">Darby Lim</author>
<buildtool_depend>ament_cmake</buildtool_depend>
<depend>rclcpp</depend>
<depend>rclcpp_action</depend>
<depend>msg_srv_action_interface_example</depend>
<test_depend>ament_lint_auto</test_depend>
<test_depend>ament_lint_common</test_depend>
<export>
  <build_type>ament_cmake</build_type>
</export>
</package>
```

ROS 2의 패키지 설정 파일의 package format은 세번째 버전이다. (ROS 에서는 두번째 버전을 사용한다.)

```
<package format="3">
빌드툴은 ament_cmake 를 사용하도록 하였고,
```

```
<buildtool_depend>ament_cmake</buildtool_depend>
dependency로 rclcpp 와 rclcpp_action 그리고 `027 토픽, 서비스, 액션 인터페이스` [8]
강좌에서 작성한 토픽, 서비스, 액션 인터페이스(패키지명: msg_srv_action_interface_example)를
적어준다.
```

```
<depend>rclcpp</depend>
<depend>rclcpp_action</depend>
<depend>msg_srv_action_interface_example</depend>
```

test_depend 태그에는 사용하고자 하는 Lint 패키지를 명시한다. 해당 태그에 테스트 코드를 위한
의존성 패키지도 명시해줄 수 있다.

```
<test_depend>ament_lint_auto</test_depend>
<test_depend>ament_lint_common</test_depend>
```

export 태그 아래의 build_type에는 ament_cmake 를 적어준다.(Python 패키지는
ament_python 이다.)

```
<export>
  <build_type>ament_cmake</build_type>
</export>
```

CMakelists.txt

topic_service_action_rclcpp_example 패키지의 빌드 설정 파일 (CMakeLists.txt)의 전문은 다음과 같다.

```
# Set minimum required version of cmake, project name and compile options
cmake_minimum_required(VERSION 3.5)
project(topic_service_action_rclcpp_example)

if(NOT CMAKE_C_STANDARD)
  set(CMAKE_C_STANDARD 99)
endif()

if(NOT CMAKE_CXX_STANDARD)
  set(CMAKE_CXX_STANDARD 14)
endif()

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
```

```

add_compile_options(-Wall -Wextra -Wpedantic)
endif()

# Find dependencies
find_package(ament_cmake REQUIRED)
find_package(msg_srv_action_interface_example REQUIRED)
find_package(rclcpp REQUIRED)
find_package(rclcpp_action REQUIRED)

include_directories(include)

# Build
add_executable(argument src/arithmetic/argument.cpp)
ament_target_dependencies(argument
    msg_srv_action_interface_example
    rclcpp
)

add_executable(calculator src/calculator/main.cpp src/calculator/calculator.cpp)
ament_target_dependencies(calculator
    msg_srv_action_interface_example
    rclcpp
    rclcpp_action
)

add_executable(checker src/checker/main.cpp src/checker/checker.cpp)
ament_target_dependencies(checker
    msg_srv_action_interface_example
    rclcpp
    rclcpp_action
)

add_executable(operator src/arithmetic/operator.cpp)
ament_target_dependencies(operator
    msg_srv_action_interface_example
    rclcpp
)

# Install
install(TARGETS
    argument
    calculator
    checker
    operator
    DESTINATION lib/${PROJECT_NAME}
)

install(DIRECTORY launch param
    DESTINATION share/${PROJECT_NAME}
)

# Test
if(BUILD_TESTING)
    find_package(ament_lint_auto REQUIRED)
    ament_lint_auto_find_test_dependencies()
endif()

# Macro for ament package
ament_package()

```

ROS 2 빌드를 위한 CMakeList 파일은 크게 cmake 설정, 의존성 명시, 빌드, 설치, 테스트, ament package 매크로 설정으로 나눌 수 있다.

cmake는 최소 3.5 버전 이상을 사용해야 하며, 별다른 명시가 없다면 C99와 C++14를 기본으로 사용하게 된다. GNU 컴파일러를 기본으로 사용하지만 Clang 컴파일러를 사용할 수도 있다.

```
cmake_minimum_required(VERSION 3.5)
project(topic_service_action_rclcpp_example)

if(NOT CMAKE_C_STANDARD)
  set(CMAKE_C_STANDARD 99)
endif()

if(NOT CMAKE_CXX_STANDARD)
  set(CMAKE_CXX_STANDARD 14)
endif()

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()
```

ament_target_dependencies에는 프로그램 실행을 위해 필요한 의존성 패키지들을 모두 적어준다.

```
ament_target_dependencies(argument
  msg_srv_action_interface_example
  rclcpp
)
```

launch 와 param 폴더는 share 폴더 아래에 저장해야만 하는것 기억하자.

```
install(DIRECTORY launch param
  DESTINATION share/${PROJECT_NAME}
)
```

Lint와 Test 코드를 위한 의존성 패키지는 아래와 같이 적어준다. 이는 colcon test 명령어를 통해 실행할 수 있다.

```
if(BUILD_TESTING)
  find_package(ament_lint_auto REQUIRED)
  ament_lint_auto_find_test_dependencies()
endif()
```

가장 아래에는 ament 패키지를 위한 매크로 함수를 적어주게 되는데 ament_package는 꼭 적어주어야 한다.

```
ament_package()
언급하지 않은 명령어들이 궁금하다면 CMake 3.5를 확인해보기 바란다.
```

소스코드 다운로드 및 빌드

소스 코드 다운로드 및 빌드는 하기와 같은 명령어로 진행하면 된다.

```
$ cd ~/robot_ws/src
$ git clone https://github.com/robotpilot/ros2-seminar-examples.git
$ cd ~/robot_ws && colcon build --symlink-install
$ echo 'source ~/robot_ws/install/local_setup.bash' >> ~/.bashrc
$ source ~/.bashrc
```

빌드한 후 빌드에 문제가 없다면 `~/robot_ws/install/topic_service_action_rclcpp_example` 폴더 안에 우리가 작성한 ROS 인터페이스를 사용하기 위한 파일들이 저장되게 된다.

예를 들어 하기 폴더에는 argument, operator, calculator, checker와 같은 실행 스크립트가 위치하게 된다.

```
~/robot_ws/install/topic_service_action_rclcpp_example/lib/  
topic_service_action_rclcpp_example
```

그리고 하기 폴더에는 launch 폴더와 param 폴더가 생성되고 각각 런치파일 arithmetic.launch.py 파일과 파라미터 파일인 arithmetic_config.yaml 파일이 위치하게 된다.

```
~/robot_ws/install/topic_service_action_rclcpp_example/share/  
topic_service_action_rclcpp_example
```

[출처] 034 ROS 2 패키지 설계 (C++) (오픈소스 소프트웨어 & 하드웨어: 로봇 기술 공유 카페 (오로카)) | 작성자 Routiful

6.6.5. C++ 패키지 실행하기

토픽 서브스크라이버, 서비스 서버, 액션 서버 실행

제일 먼저 calculator 노드를 실행해보자. 이 노드는 이 패키지에서 토픽 서브스크라이버, 서비스 서버, 액션 서버 역할을 수행하는 노드로 실행이 되었다는 로그 한줄을 띄워주고 난 이후로는 아무런 동작은 하고 있지 않은 대기 상태가 된다.

```
$ ros2 run topic_service_action_rclcpp_example calculator  
[INFO]: Run calculator
```

토픽 퍼블리셔 실행

다음으로 새로운 터미널을 열어서 토픽 퍼블리셔 역할을 하는 argument 노드를 실행해보자. 그러면 터미널 창에 퍼블리시하고 있는 argument a, argument b 가 표시된다. 더불어 calculator 노드를 실행 시킨 터미널 창에서는 수신 받은 시간 정보와 argument a, argument b 가 표시된다.

```
$ ros2 run topic_service_action_rclcpp_example argument  
[INFO]: Published argument_a 0.25  
[INFO]: Published argument_b 5.93  
[INFO]: Published argument_a 5.45  
[INFO]: Published argument_b 1.37  
[INFO]: Run calculator  
[INFO]: Subscribed at: sec 1610237910 nanosec 33666586  
[INFO]: Subscribed argument a : 0.25  
[INFO]: Subscribed argument b : 5.93  
[INFO]: Subscribed at: sec 1610237911 nanosec 33606825  
[INFO]: Subscribed argument a : 5.45  
[INFO]: Subscribed argument b : 1.37
```

서비스 클라이언트 실행

다음으로 새로운 터미널을 열어서 서비스 클라이언트 역할을 하는 operator 노드를 실행해보자. 그러면 calculator 노드에게 랜덤으로 선택한 연산자(+,-,*,/)를 서비스 요청값으로 보내고 연산된 결과값을

받아 터미널 창에 표시한다. 실제 계산식은 calculator 노드가 실행 중인 창에서 확인할 수 있다. 예를 들어 calculator는 변수 a, b로 9.0 와 6.0 을 받았고 연산자로 곱셈(*)을 받았을 때 $9.0 * 6.0 = 54.0$ 이므로 operator는 54.0 를 표시하게 된다. Enter 키를 눌러 서비스를 한번 더 보낼 수 있다.

```
$ ros2 run topic_service_action_rclcpp_example operator
Press Enter for next service call.
[INFO]: Result 4.69
Press Enter for next service call.
[INFO]: Result 6.07
[INFO]: Subscribed at: sec 1610238085 nanosec 806984947
[INFO]: Subscribed argument a : 5.17
[INFO]: Subscribed argument b : 0.47
[INFO]: 5.167154 - 0.473616 = 4.69354

[INFO]: Subscribed at: sec 1610238086 nanosec 807010534
[INFO]: Subscribed argument a : 1.36
[INFO]: Subscribed argument b : 4.47
[INFO]: 1.358604 * 4.469692 = 6.07254
```

액션 클라이언트 실행

마지막으로 터미널을 새로 열어서 checker 노드를 실행해보자. checker 노드는 먼저 연산값의 합계 한계치를 액션 목표값으로 calculator 노드에 전달하게 된다. 이후 checker 노드는 calculator 노드에게 액션 피드백을 받게 되는데, 그 피드백은 각 연산과 그 결과의 string 타입이다. 지정한 연산값의 합계가 목표 합계를 넘기면 checker 노드는 액션 결과값으로 calculator 노드로 부터 최종 연산 합계 받게 된다.

```
$ ros2 run topic_service_action_rclcpp_example checker
goal_total_sum : 50
[INFO]: Action goal accepted.
[INFO]: Action feedback:
[INFO]:     4.983427 * 5.585670 = 27.8358

[INFO]: Action feedback:
[INFO]:     4.983427 * 5.585670 = 27.8358

[INFO]:     4.983427 * 5.585670 = 27.8358

[INFO]: Action succeeded!
[INFO]: Action result(all formula):
[INFO]:     4.983427 * 5.585670 = 27.8358

[INFO]:     4.983427 * 5.585670 = 27.8358

[INFO]: Action result(total sum):
[INFO]:     55.67
```

합계의 한계치는 기본으로 50 으로 설정되어 있다. 이를 수정하여 실행하고 싶으면 아래와 같이 checker 노드를 실행시키면서 실행 인자로 `-g 100` 이라고 입력하면 GOAL_TOTAL_SUM 이라는 인자로 100 을 할당할 수 있다.

```
$ ros2 run topic_service_action_rclcpp_example checker -g 100
```

런치 파일 실행

참고로 argument 노드와 calculator 노드를 한번에 실행시키고자 한다면 아래와 같이 launch 파일인 arithmetic.launch.py 를 실행하는 방법으로 두개의 노드를 동시에 실행 시킬 수 있다. launch 파일에 대한 자세한 설명은 추후 강좌에서 더 자세히 알아볼 예정이기에 여기서는 실행 방법만 알아보았다.

```
$ ros2 launch topic_service_action_rclcpp_example arithmetic.launch.py
```

6.7. ROS2 Python 프로그래밍

6.7.1. 토픽 퍼블리셔(Argument.py)

```
import random

from msg_srv_action_interface_example.msg import ArithmeticArgument
from rcl_interfaces.msg import SetParametersResult
import rclpy
from rclpy.node import Node
from rclpy.parameter import Parameter
from rclpy.qos import QoSDurabilityPolicy
from rclpy.qos import QoSHistoryPolicy
from rclpy.qos import QoSProfile
from rclpy.qos import QoSReliabilityPolicy

class Argument(Node):

    def __init__(self):
        super().__init__('argument')
        self.declare_parameter('qos_depth', 10)
        qos_depth = self.get_parameter('qos_depth').value
        self.declare_parameter('min_random_num', 0)
        self.min_random_num = self.get_parameter('min_random_num').value
        self.declare_parameter('max_random_num', 9)
        self.max_random_num = self.get_parameter('max_random_num').value
        self.add_on_set_parameters_callback(self.update_parameter)

        QOS_RKL10V = QoSProfile(
            reliability=QoSReliabilityPolicy.RELIABLE,
            history=QoSHistoryPolicy.KEEP_LAST,
            depth=qos_depth,
            durability=QoSDurabilityPolicy.VOLATILE)

        self.arithmetic_argument_publisher = self.create_publisher(
            ArithmeticArgument,
            'arithmetic_argument',
            QOS_RKL10V)

        self.timer = self.create_timer(1.0, self.publish_random_arithmetic_arguments)

    def publish_random_arithmetic_arguments(self):
        msg = ArithmeticArgument()
        msg.stamp = self.get_clock().now().to_msg()
        msg.argument_a = float(random.randint(self.min_random_num, self.max_random_num))
```

```

msg.argument_b = float(random.randint(self.min_random_num, self.max_random_num))
self.arithmetic_argument_publisher.publish(msg)
self.get_logger().info('Published argument a: {}'.format(msg.argument_a))
self.get_logger().info('Published argument b: {}'.format(msg.argument_b))

def update_parameter(self, params):
    for param in params:
        if param.name == 'min_random_num' and param.type_ == Parameter.Type.INTEGER:
            self.min_random_num = param.value
        elif param.name == 'max_random_num' and param.type_ ==
Parameter.Type.INTEGER:
            self.max_random_num = param.value
    return SetParametersResult(successful=True)

def main(args=None):
    rclpy.init(args=args)
    try:
        argument = Argument()
        try:
            rclpy.spin(argument)
        except KeyboardInterrupt:
            argument.get_logger().info('Keyboard Interrupt (SIGINT)')
        finally:
            argument.destroy_node()
    finally:
        rclpy.shutdown()

if __name__ == '__main__':
    main()

```

이 중 토픽 퍼블리셔와 관련한 코드는 아래에 간추려 놓았다. 우선 Argument 클래스인데 rclpy.node 모듈의 Node 클래스를 상속하고 있으며 생성자에서 'argument'이라는 노드 이름으로 초기화되었다. 그 뒤 rclpy.qos 모듈의 QoSProfile 클래스를 이용하여 토픽 퍼블리셔에서 사용할 QoS 설정을 해주었다. 여기서 QoS 는 RELIABLE, KEEP_LAST, DEPTH 10, VOLATILE로 설정하였다.

```

class Argument(Node):

    def __init__(self):
        super().__init__('argument')

    (일부 코드 생략)

```

```

QOS_RKL10V = QoSProfile(
    reliability=QoSReliabilityPolicy.RELIABLE,
    history=QoSHistoryPolicy.KEEP_LAST,
    depth=qos_depth,
    durability=QoSDurabilityPolicy.VOLATILE)

```

제일 중요한 설정은 arithmetic_argument_publisher 선언이다. 이는 Node 클래스의 create_publisher 함수를 이용하여 퍼블리셔를 선언하는 부분으로 토픽의 타입으로 ArithmeticArgument 으로 선언하였고, 토픽 이름으로는 'arithmetic_argument', QoS 는 위에서 설정한 QOS_RKL10V 를 사용하였다.

```

self.arithmetic_argument_publisher = self.create_publisher(
    ArithmeticArgument,
    'arithmetic_argument',
    QOS_RKL10V)

```

그리고 마지막 줄에서 create_timer 함수를 이용하여 1 초마다 publish_random_arithmetic_arguments 이라는 함수를 실행시키도록 설정하였다. 이전 설정들은 퍼블리시를 위한 기본 설정이고 실제 토픽 발행이

이루어지는 부분은 publish_random_arithmetic_arguments 함수임을 알아두자.

```
self.timer = self.create_timer(1.0, self.publish_random_arithmetic_arguments)
```

다음으로 publish_random_arithmetic_arguments 함수의 내용을 보도록 하자. 이 함수는 지정 타이머로 동작하는 콜백함수이며 1초마다 실행되며 매번 실행될 때마다 msg 인터페이스의 ArithmeticArgument() 클래스로 생성해주었고 get_clock().now().to_msg() 함수를 통하여 토픽이 생성된 시간을 msg.stamp에 기록해두었다. 그 뒤 랜덤 함수를 통해 0~9 까지의 숫자를 float로 변환하여 msg.argument_a 와 msg.argument_b에 저장하였다.

그 뒤 이어지는 arithmetic_argument_publisher.publish(msg) 함수가 실제로 토픽 발행이 이루어지는 함수로 우리가 발행 시간 및 변수 a, b를 저장한 msg 메시지를 퍼블리쉬한다는 의미이다.

마지막으로 get_logger().info()는 주로 디버깅에 사용되는 함수로 노드를 실행한 터미널 창에 특정 값을 표시하게 된다. 여기서는 토픽으로 보낸 변수 a, b 값을 화면에 표시하고 있다.

```
def publish_random_arithmetic_arguments(self):
    msg = ArithmeticArgument()
    msg.stamp = self.get_clock().now().to_msg()
    msg.argument_a = float(random.randint(self.min_random_num, self.max_random_num))
    msg.argument_b = float(random.randint(self.min_random_num, self.max_random_num))
    self.arithmetic_argument_publisher.publish(msg)
    self.get_logger().info('Published argument a: {}'.format(msg.argument_a))
    self.get_logger().info('Published argument b: {}'.format(msg.argument_b))
```

6.7.2. 토픽 서브스크라이버(calculator.py)

우선 Calculator 클래스인데 토픽 퍼블리셔 노드와 마찬가지로 rclpy.node 모듈의 Node 클래스를 상속하고 있으며 생성자에서 'calculator'이라는 노드 이름으로 초기화되었다. 그 뒤 위에서와 마찬가지로 rclpy.qos 모듈의 QoSProfile 클래스[18]를 이용하여 토픽 서브스크라이버에서 사용할 QoS 설정을 해주었다. 여기서 QoS는 앞서 설명한 토픽 퍼블리셔 노드와 동일하게 RELIABLE, KEEP_LAST, DEPTH 10, VOLATILE로 설정하였다.

```
class Calculator(Node):

    def __init__(self):
        super().__init__('calculator')
        self.argument_a = 0.0
        self.argument_b = 0.0
        self.callback_group = ReentrantCallbackGroup()

    # 일부 코드 생략

    QOS_RKL10V = QoSProfile(
        reliability=QoSReliabilityPolicy.RELIABLE,
        history=QoSHistoryPolicy.KEEP_LAST,
        depth=qos_depth,
        durability=QoSDurabilityPolicy.VOLATILE)
```

제일 중요한 설정은 arithmetic_argument_subscriber 선언이다. 이는 Node 클래스의 create_subscription 함수를 이용하여 서브스크라이버로 선언하는 부분으로 토픽의 타입으로 퍼블리셔와 동일하게 ArithmeticArgument으로 선언하였고, 토픽 이름으로는 'arithmetic_argument', QoS는 위에서 설정한 QOS_RKL10V를 사용하였다. 여기까지는

퍼블리셔와 비슷한데 퍼블리셔와 다른점은 `get_arithmetic_argument`라는 콜백함수를 두어 퍼블리셔로부터 메시지를 서브스크라이브할 때 마다 실행되는 함수를 지정한 것이다. 그리고 `callback_group`을 `ReentrantCallbackGroup`으로 지정했는데 콜백함수를 병렬로 실행할 수 있게 해주며 뒤에서 설명할 `MultiThreadedExecutor`와 함께 사용되곤 한다.

좀 더 자세히 `callback_group`을 설명하자면 원래 `callback_group`은 지정하지 않아도 되는데 지정하지 않게 되면 `MutuallyExclusiveCallbackGroup`이 기본 설정으로 사용된다. `MutuallyExclusive CallbackGroup`은 한 번에 하나의 콜백함수만 실행하도록 허용하는 것이고, 다른 하나의 설정은 `ReentrantCallbackGroup`으로써 제한없이 콜백함수를 병렬로 실행할 수 있게 해준다. 이러한 `callback_group` 설정은 `create_subscription()`, `create_service()`, `ActionServer()`, `create_timer()`에서 사용됨을 알아두도록 하자.

```
self.arithmetic_argument_subscriber = self.create_subscription(  
    ArithmeticArgument,  
    'arithmetic_argument',  
    self.get_arithmetic_argument,  
    QOS_RKL10V,  
    callback_group=self.callback_group)
```

하기의 `get_arithmetic_argument` 함수는 앞서서 콜백함수라 설명하였다. 이 함수는 `arithmetic_argument`이라는 토픽 이름에 `ArithmeticArgument` 타입의 메시지를 서브스크라이브하게 되면 실행되는 함수이다. 서브스크라이브한 `msg`의 `argument_a`와 `argument_b`를 멤버 변수에 저장하고 `get_logger().info()` 함수를 이용하여 토픽으로 받은 시간, 변수 `a`, `b` 값을 화면에 표시하고 있다.

```
def get_arithmetic_argument(self, msg):  
    self.argument_a = msg.argument_a  
    self.argument_b = msg.argument_b  
    self.get_logger().info('Subscribed at: {}'.format(msg.stamp))  
    self.get_logger().info('Subscribed argument a: {}'.format(self.argument_a))  
    self.get_logger().info('Subscribed argument b: {}'.format(self.argument_b))
```

6.7.3. 서비스 서버(calculator.py)

`calculator` 노드는 서브스크라이브하여 저장하고 있는 변수 `a`와 `b`와 `operator` 노드로부터 요청 값으로 받은 연산자를 이용하여 계산(`a` 연산자 `b`)하고 `operator` 노드에게 연산의 결괏값을 서비스 응답값으로 보낸다.

이 중 서비스 서버와 관련한 코드는 아래와 같다. 서버 관련 코드는 서비스 서버로 선언하는 부분과 콜백함수를 지정하는 것이다. `arithmetic_service_server`이 서비스 서버로 이는 `Node` 클래스의 `create_service` 함수를 이용하여 서비스 서버로 선언되었으며 서비스의 타입으로 `ArithmeticOperator`으로 선언하였고, 서비스 이름으로는 '`arithmetic_operator`', 서비스 클라이언트로부터 서비스 요청이 있으면 실행되는 콜백함수는 `get_arithmetic_operator`으로 지정했으며 멀티 스레드 병렬 콜백함수 실행을 위해 지난번 강좌에서 설명한 `callback_group` 설정을 하였다.

이러한 설정들은 서비스 서버를 위한 기본 설정이고 실제 서비스 요청에 해당되는 특정 수행 코드가

수행되는 부분은 `get_arithmetic_operator` 이라는 콜백함수임을 알아두자.

```
self.arithmetic_service_server = self.create_service(  
    ArithmeticOperator,  
    'arithmetic_operator',  
    self.get_arithmetic_operator,  
    callback_group=self.callback_group)
```

자 그러면 `get_arithmetic_operator` 함수의 내용을 보도록 하자. 우선 제일 먼저 `request` 와 `response` 이라는 매개 변수가 보이는데 이는 `ArithmeticOperator()` 클래스로 생성된 인터페이스로 서비스 요청에 해당되는 `request` 부분과 응답에 해당되는 `response` 으로 구분된다. `get_arithmetic_operator` 함수는 서비스 요청이 있을 때 실행되는 콜백함수인데 여기서는 서비스 요청시 요청값으로 받은 연산자와 이전에 토픽 서브스크라이버가 토픽 값으로 전달받아 저장해둔 변수 `a`, `b` 를 전달받은 연산자로 연산 후에 결괏값을 서비스 응답값으로 반환한다. 이 함수의 첫 줄에서 `request.arithmetic_operator`를 받아와서 `calculate_given_formula` 함수에서 서비스 요청값으로 받은 연산자에 따라 연산하는 코드를 볼 수 있을 것이다. `calculate_given_formula` 함수로부터 받은 연산 결괏값은 `response.arithmetic_result`에 저장하고 끝으로 관련 수식을 문자열로 표현하여 `get_logger().info()` 함수를 통해 화면에 표시하고 있다.

```
def get_arithmetic_operator(self, request, response):  
    self.argument_operator = request.arithmetic_operator  
    self.argument_result = self.calculate_given_formula(  
        self.argument_a,  
        self.argument_b,  
        self.argument_operator)  
    response.arithmetic_result = self.argument_result  
    self.argument_formula = '{0} {1} {2} = {3}'.format(  
        self.argument_a,  
        self.operator[self.argument_operator-1],  
        self.argument_b,  
        self.argument_result)  
    self.get_logger().info(self.argument_formula)  
    return response
```

하기 함수는 `calculate_given_formula` 함수로 앞서 설명한 것과 같이 인수 `a`, `b` 를 가지고 주어진 연산자 `operator`에 따라 사칙연산을 하여 결괏값을 반한다.

```
def calculate_given_formula(self, a, b, operator):  
    if operator == ArithmeticOperator.Request.PLUS:  
        self.argument_result = a + b  
    elif operator == ArithmeticOperator.Request_MINUS:  
        self.argument_result = a - b  
    elif operator == ArithmeticOperator.Request.MULTIPLY:  
        self.argument_result = a * b  
    elif operator == ArithmeticOperator.Request.DIVISION:  
        try:  
            self.argument_result = a / b  
        except ZeroDivisionError:  
            self.get_logger().error('ZeroDivisionError!')  
            self.argument_result = 0.0  
        return self.argument_result  
    else:  
        self.get_logger().error(  
            'Please make sure arithmetic operator(plus, minus, multiply, division).')  
        self.argument_result = 0.0  
    return self.argument_result
```

6.7.4. 서비스 클라이언트(operator.py)

서비스 클라이언트의 전체 코드는 아래와 같다.

```
import random

from msg_srv_action_interface_example.srv import ArithmeticOperator
import rclpy
from rclpy.node import Node


class Operator(Node):

    def __init__(self):
        super().__init__('operator')

        self.arithmetic_service_client = self.create_client(
            ArithmeticOperator,
            'arithmetic_operator')

        while not self.arithmetic_service_client.wait_for_service(timeout_sec=0.1):
            self.get_logger().warning('The arithmetic_operator service not available.')

    def send_request(self):
        service_request = ArithmeticOperator.Request()
        service_request.arithmetic_operator = random.randint(1, 4)
        futures = self.arithmetic_service_client.call_async(service_request)
        return futures

    def main(args=None):
        rclpy.init(args=args)
        operator = Operator()
        future = operator.send_request()
        user_trigger = True
        try:
            while rclpy.ok():
                if user_trigger is True:
                    rclpy.spin_once(operator)
                    if future.done():
                        try:
                            service_response = future.result()
                        except Exception as e: # noqa: B902
                            operator.get_logger().warn('Service call failed: {}'.format(str(e)))
                        else:
                            operator.get_logger().info(
                                'Result: {}'.format(service_response.arithmetic_result))
                            user_trigger = False
                else:
                    input('Press Enter for next service call.')
                    future = operator.send_request()
                    user_trigger = True
        except KeyboardInterrupt:
            operator.get_logger().info('Keyboard Interrupt (SIGINT)')

        operator.destroy_node()
        rclpy.shutdown()
```

```
if __name__ == '__main__':
    main()
```

우선 Operator 클래스인데 rclpy.node 모듈의 Node 클래스를 상속하고 있으며 생성자에서 'operator'이라는 노드 이름으로 초기화되었다. 그 뒤 arithmetic_service_client이라는 이름으로 서비스 클라이언트를 선언해주는데 이는 Node 클래스의 create_client 함수를 이용하여 서비스 클라이언트로 선언하는 부분으로 서비스의 타입으로 서비스 서버와 동일하게 ArithmeticOperator으로 선언하였고, 서비스 이름으로는 'arithmetic_operator'으로 선언하였다. arithmetic_service_client 의 wait_for_service 함수는 서비스 요청을 할 수 있는 상태인지 알아보기 위해 서비스 서버가 실행되어 있는지 확인하는 함수로 0.1 초 간격으로 서비스 서버가 실행되어 있는지 확인하게 된다.

```
class Operator(Node):
    def __init__(self):
        super().__init__('operator')

        self.arithmetic_service_client = self.create_client(
            ArithmeticOperator,
            'arithmetic_operator')

    while not self.arithmetic_service_client.wait_for_service(timeout_sec=0.1):
        self.get_logger().warning('The arithmetic_operator service not available.')
```

우리가 작성하고 있는 서비스 클라이언트의 목적은 서비스 서버에게 연산에 필요한 연산자를 보내는 것이라고 했다. 이 send_request 함수가 실질적인 서비스 클라이언트의 실행 코드로 서비스 서버에게 서비스 요청값을 보내고 응답값을 받게 된다. 서비스 요청값을 보내기 위하여 제일 먼저 우리가 미리 작성해둔 서비스 인터페이스 ArithmeticOperator.Request() 클래스로 service_request를 선언하였고 서비스 요청값으로 random.randint() 함수를 이용하여 특정 연산자를 self.request의 arithmetic_operator 변수에 저장하였다. 그 뒤 'call_async(self.request)' 함수로 서비스 요청을 수행하게 설정하였다. 끝으로 서비스 상태 및 응답값을 담은 futures를 반환하게 된다.

```
def send_request(self):
    service_request = ArithmeticOperator.Request()
    service_request.arithmetic_operator = random.randint(1, 4)
    futures = self.arithmetic_service_client.call_async(service_request)
    return futures
```

6.7.5. 액션 서버

액션 서버 역할을 하는 calculator 노드의 소스 코드는 깃허브 리포지토리에 위치해놓았다. 해당 리포지토리에서 하기의 장소에 관련 코드가 있기에 참고하도록 하자.

```
topic_service_action_rclpy_example/topic_service_action_rclpy_example/calculator/
calculator.py
```

이 코드는 이전 강좌에서 토픽 서브스크라이버, 서비스 서버를 설명하였을 때 소개하였던 노드로써 토픽 서브스크라이버, 서비스 서버, 액션 서버를 모두 포함하고 있어서 매우 길기 때문에 전체 코드를 강좌 글에 담는 것은 생략하도록 하고 전체 소스 코드 중 액션 서버와 관련한 코드에 대한 설명한 하기로 하자.

calculator 노드는 서브스크라이브하여 저장하고 있는 변수 a 와 b 와 operator 노드로부터 요청

값으로 받은 연산자를 이용하여 계산(a 연산자 b)하고 operator 노드에게 연산의 결괏값을 서비스 응답값으로 보낸다는 것은 이전 강좌에서 설명하였다. 여기에 각 연산을 기록하고 액션 서버로 동작하는 부분에 대해 추가 설명을 하도록 하겠다.

액션 서버와 관련한 코드는 아래와 같다. 서버 관련 코드는 액션 서버로 선언하는 부분과 콜백함수를 지정하는 것이다. `arithmetic_action_server` 이 액션 서버로 이는 `rclpy.action` 모듈의 `ActionServer` 클래스를 이용하여 액션 서버로 선언되었으며 액션의 타입으로 `ArithmeticChecker` 으로 선언하였고, 액션 이름으로는 '`arithmetic_checker`', 액션 클라이언트로부터 액션 목표를 받으면 실행되는 콜백함수는 `execute_checker` 으로 지정했으며 멀티 스레드 병렬 콜백함수 실행을 위해 지난 번 강좌에서 설명한 `callback_group` 설정을 하였다. 이러한 설정들은 액션 서버를 위한 기본 설정이고 실제 액션 목표를 받은 후에 실행되는 콜백함수는 `execute_checker` 함수임을 알아두자.

```
self.arithmetic_action_server = ActionServer(  
    self,  
    ArithmeticChecker,  
    'arithmetic_checker',  
    self.execute_checker,  
    callback_group=self.callback_group)
```

자 그러면 `execute_checker` 함수의 내용을 보도록 하자. 우선 제일 먼저 `goal_handle` 이라는 매개 변수가 보이는데 이는 `rclpy.action` 모듈의 `ServerGoalHandle` 클래스로 생성된 액션 상태 처리용으로 `execute`, `succeed`, `abort`, `canceled` 등과 같이 액션 상태에 따른 관련 함수 호출이 가능하며 `publish_feedback`와 같이 피드백을 퍼블리시할 수도 있다.

이어서 함수 내용을 더 자세히 알아보도록 하자. 첫 줄에서는 `get_logger().info()` 함수를 이용해 터미널창에 액션 서버가 시작됨을 표시하고, 다음 줄에서 `ArithmeticChecker.Feedback()`으로 액션 피드백을 보낼 `feedback_msg` 변수를 선언하였다. 이어서 실제 피드백에 해당되는 `feedback_msg.formula` 와 연산 합계값을 담을 `total_sum` 변수를 초기화하였다. 그 뒤 앞서 설명한 `goal_handle` 를 이용하여 `goal_handle.request.goal_sum` 에서 액션 목표값을 불러왔다.

```
def execute_checker(self, goal_handle):  
    self.get_logger().info('Execute arithmetic_checker action!')  
    feedback_msg = ArithmeticChecker.Feedback()  
    feedback_msg.formula = []  
    total_sum = 0.0  
    goal_sum = goal_handle.request.goal_sum  
    while total_sum < goal_sum:  
        total_sum += self.argument_result  
        feedback_msg.formula.append(self.argument_formula)  
        self.get_logger().info('Feedback: {}'.format(feedback_msg.formula))  
        goal_handle.publish_feedback(feedback_msg)  
        time.sleep(1)  
    goal_handle.succeed()  
    result = ArithmeticChecker.Result()  
    result.all_formula = feedback_msg.formula  
    result.total_sum = total_sum  
    return result
```

다음 `while` 반복 구문은 액션 목표값(`goal_sum`)과 `get_arithmetic_operator` 를 통해 매해 계산되는 연산 결괏값인 `argument_result` 를 누적한 합계(`total_sum`)값을 비교하여 액션 목표값을 넘을 때까지의 연산식(`argument_formula`)을 액션 피드백(`feedback_msg.formula`)에 저장하는 구문을 볼 수 있다. 이 피드백 값은 디버깅을 위해 `get_logger().info()` 를 통해 터미널창에 출력하고

`goal_handle.publish_feedback()` 함수를 통해 액션 클라이언트에서 전송하게 된다.

```
while total_sum < goal_sum:
    total_sum += self.argument_result
    feedback_msg.formula.append(self.argument_formula)
    self.get_logger().info('Feedback: {}'.format(feedback_msg.formula))
    goal_handle.publish_feedback(feedback_msg)
    time.sleep(1)
```

마지막으로 아래 구문들을 통해 액션 목표를 달성했다는 상태 전환 함수인 `goal_handle.succeed()`를 실행시켜 액션 클라이언트에게 현재의 액션 상태를 알리고 액션 결괏값인 `all_formula`에 계산식 전체를 저장하고 `total_sum`에 연산 합계를 저장하여 액션 결괏값인 `result`은 리턴하는 것으로 액션 서버의 역할을 마치게 된다.

```
goal_handle.succeed()
result = ArithmeticChecker.Result()
result.all_formula = feedback_msg.formula
result.total_sum = total_sum
return result
```

6.7.6. 액션 클라이언트

전체 코드는 아래와 같다.

```
from action_msgs.msg import GoalStatus
from msg_srv_action_interface_example.action import ArithmeticChecker
from rclpy.action import ActionClient
from rclpy.node import Node

class Checker(Node):

    def __init__(self):
        super().__init__('checker')
        self.arithmetic_action_client = ActionClient(
            self,
            ArithmeticChecker,
            'arithmetic_checker')

    def send_goal_total_sum(self, goal_sum):
        wait_count = 1
        while not self.arithmetic_action_client.wait_for_server(timeout_sec=0.1):
            if wait_count > 3:
                self.get_logger().warning('Arithmetic action server is not available.')
                return False
            wait_count += 1
        goal_msg = ArithmeticChecker.Goal()
        goal_msg.goal_sum = (float)(goal_sum)
        self.send_goal_future = self.arithmetic_action_client.send_goal_async(
            goal_msg,
            feedback_callback=self.get_arithmetic_action_feedback)
        self.send_goal_future.add_done_callback(self.get_arithmetic_action_goal)
        return True

    def get_arithmetic_action_goal(self, future):
        goal_handle = future.result()
        if not goal_handle.accepted:
            self.get_logger().warning('Action goal rejected.')

if __name__ == '__main__':
    main()
```

```

        return
    self.get_logger().info('Action goal accepted.')
    self.action_result_future = goal_handle.get_result_async()
    self.action_result_future.add_done_callback(self.get_arithmetic_action_result)

def get_arithmetic_action_feedback(self, feedback_msg):
    action_feedback = feedback_msg.feedback.formula
    self.get_logger().info('Action feedback: {}'.format(action_feedback))

def get_arithmetic_action_result(self, future):
    action_status = future.result().status
    action_result = future.result().result
    if action_status == GoalStatus.STATUS_SUCCEEDED:
        self.get_logger().info('Action succeeded!')
        self.get_logger().info(
            'Action result(all formula): {}'.format(action_result.all_formula))
        self.get_logger().info(
            'Action result(total sum): {}'.format(action_result.total_sum))
    else:
        self.get_logger().warning(
            'Action failed with status: {}'.format(action_status))

```

우선 Checker 클래스인데 rclpy.node 모듈의 Node 클래스를 상속하고 있으며 생성자에서 'checker' 이라는 노드 이름으로 초기화되었다. 그 뒤 arithmetic_action_client이라는 이름으로 액션 클라이언트를 선언해주는데 이는 rclpy.action 모듈의 ActionClient 클래스 이용하여 액션 클라이언트로 선언하는 부분으로 액션의 타입으로 액션 서버와 동일하게 ArithmeticChecker으로 선언하였고, 서비스 이름으로는 'arithmetic_checker'으로 선언하였다.

액션 클라이언트에서 수행하는 액션 목표는 항시 퍼블리시하고 서브스크라이브하는 토픽과는 달리 필요시 비정기적으로 수행하는 것이다. 여기서는 예시를 위해 액션 목표를 main 함수에서 한번 실행하게 된다. 이 부분은 이어지는 설명에서 다루겠다. 우선 액션 목표 전송, 액션 상태 파악, 피드백 및 결괏값 받기를 위한 함수를 우선 알아보자.

```

class Checker(Node):

    def __init__(self):
        super().__init__('checker')
        self.arithmetic_action_client = ActionClient(
            self,
            ArithmeticChecker,
            'arithmetic_checker')

```

send_goal_total_sum 함수는 액션 목표를 액션 서버에게 전송하고 액션 피드백 및 결괏값을 받기 위한 콜백함수를 지정하는 함수로 사용하고 있다.

```

def send_goal_total_sum(self, goal_sum):
    wait_count = 1
    while not self.arithmetic_action_client.wait_for_server(timeout_sec=0.1):
        if wait_count > 3:
            self.get_logger().warning('Arithmetic action server is not available.')
            return False
        wait_count += 1
    goal_msg = ArithmeticChecker.Goal()
    goal_msg.goal_sum = (float)(goal_sum)
    self.send_goal_future = self.arithmetic_action_client.send_goal_async(
        goal_msg,
        feedback_callback=self.get_arithmetic_action_feedback)
    self.send_goal_future.add_done_callback(self.get_arithmetic_action_goal)

```

```
return True
```

하기 내용은 액션 클라이언트가 액션 서버에 연결시도를 하는 것으로 연결에 문제가 있을 때에 하기의 while 문을 반복하게 되고 문제 없이 연결되었을 때에는 다음 구문으로 넘어가게 된다.

```
wait_count = 1
while not self.arithmetic_action_client.wait_for_server(timeout_sec=0.1):
    if wait_count > 3:
        self.get_logger().warning('Arithmetic action server is not available.')
        return False
    wait_count += 1
```

그 뒤 ArithmeticChecker.Goal() 클래스로 액션 메시지 goal_msg 을 선언하고, goal_msg.goal_sum 과 같이 액션 목표값을 설정하게 된다. 그리고 ActionClient 클래스의 send_goal_async 함수를 이용하여 미리 설정해둔 액션 메시지를 매개변수로 넣고 피드백을 전달 받기 위한 콜백함수로 get_arithmetic_action_feedback 함수를 지정하였다. 마지막으로 send_goal_async 를 통해 선언된 비동기 future task[27]로 선언된 send_goal_future 의 add_done_callback 함수를 통해 액션 결괏값을 받을 때 사용할 콜백함수로 get_arithmetic_action_goal 를 선언하였다.

```
goal_msg = ArithmeticChecker.Goal()
goal_msg.goal_sum = (float)(goal_sum)
self.send_goal_future = self.arithmetic_action_client.send_goal_async(
    goal_msg,
    feedback_callback=self.get_arithmetic_action_feedback)
self.send_goal_future.add_done_callback(self.get_arithmetic_action_goal)
```

액션 처리 과정이 좀 복잡하기에 다시 복기를 하자면 다음과 같다. 지금까지 앞의 2 단계를 했기에 남은 3 가지 함수를 더 알아보도록 하자.

- (1) 액션 클라이언트 선언
- (2) 액션 목표값 전달 함수 선언
- (3) 액션 피드값 콜백 함수 선언
- (4) 액션 상태 콜백 함수 선언
- (5) 액션 결괏값 콜백 함수 선언

"액션 피드값 콜백 함수"는 다음과 같다. 액션 피드백을 액션 서버로부터 전달 받으면 아래 콜백함수가 실행되는데 피드백인 feedback_msg.feedback.formula 값을 받아 get_logger().info() 함수를 이용해 터미널창에 출력하는 것이다.

```
def get_arithmetic_action_feedback(self, feedback_msg):
    action_feedback = feedback_msg.feedback.formula
    self.get_logger().info('Action feedback: {}'.format(action_feedback))
```

다음은 "액션 상태 콜백 함수"이다. 이 함수는 비동기 future task 로 선언된 send_goal_future 의 add_done_callback 함수를 통해 콜백함수로 선언된 함수로 액션 서버가 액션 목표값을 전달 받고 Goal State Machine 의 accepted 상태일 때를 확인하여 처리하는 구문이다. 액션 목표값을 문제없이 전달하였다면 액션 결괏값 콜백 함수를 선언하게 된다. 여기서 콜백 함수는 후술할

`get_arithmetic_action_result` 함수이다.

```
def get_arithmetic_action_goal(self, future):
    goal_handle = future.result()
    if not goal_handle.accepted:
        self.get_logger().warning('Action goal rejected.')
        return
    self.get_logger().info('Action goal accepted.')
    self.action_result_future = goal_handle.get_result_async()
    self.action_result_future.add_done_callback(self.get_arithmetic_action_result)
```

앞서 지정한 "액션 결괏값 콜백 함수"는 아래와 같다. 비동기 future task로 현재의 상태값(status)과 결괏값(result)을 받고 상태 값이 STATUS_SUCCEEDED 일 때 액션 서버로부터 전달 받은 액션 결괏값인 계산식(action_result.all_formula)과 연산 합계(action_result.total_sum)를 터미널 창에 출력하게 된다.

```
def get_arithmetic_action_result(self, future):
    action_status = future.result().status
    action_result = future.result().result
    if action_status == GoalStatus.STATUS_SUCCEEDED:
        self.get_logger().info('Action succeeded!')
        self.get_logger().info(
            'Action result(all formula): {0}'.format(action_result.all_formula))
        self.get_logger().info(
            'Action result(total sum): {0}'.format(action_result.total_sum))
    else:
        self.get_logger().warning(
            'Action failed with status: {0}'.format(action_status))
```

6.7.7. 실행인자(Arguments) 프로그래밍

프로그램의 실행 명령어에 실행 시에 사용될 인수를 추가하여 프로그램을 실행하는 경우가 있는데 이 때에 사용되는 인자를 실행 인자라고 하고 main 함수에서 매개변수로 사용된다.

예를 하기와 같은 명령어를 통해 GOAL_TOTAL_SUM 값을 100으로 설정할 수 있었다. 여기서 `ros2 run` 가 명령어이고 `topic_service_action_rclpy_example` 패키지의 `checker` 노드를 실행하라는 의미이다. 여기에 추가로 `-g 100` 이 실행 인자로 사용되었다. 이 강좌에서는 이렇게 실행시에 사용되는 실행 인자를 이용한 프로그래밍에 대해 알아보자.

```
$ ros2 run topic_service_action_rclpy_example checker -g 100
```

* 참고로 파라미터(parameter)는 매개 변수로 풀이하고 아규먼트(argument)는 실행 인자라 풀이한다. C++ 언어에서는 이들의 분류를 더 확실히 하는 편인데 Parameter는 함수 선언시 사용되고 Argument는 함수 호출 시의 인수라고 생각하면 된다.

- Parameter: 매개 변수
- Argument: 실행 인자

ROS 2 에서의 실행 인자 처리는 하기의 예제들과 같이 처리된다. C++ 언어의 경우 main 문에서 argc 라고 argument 의 수를 받고, argv 로 인자들을 배열로 받은 후 rclcpp 의 init 함수의 argument 로 넘겨주게 된다.

```
int main(int argc, char * argv[])
{
    rclcpp::init(argc, argv);
    (이하 생략)
```

Python 의 경우도 비슷한데 인수들을 무시할 때에는 다음 예제와 같이 args 를 None 으로 설정 후에 rclpy 모듈의 init 함수에 바로 넘기게 된다.

```
def main(args=None):
    rclpy.init(args=args)
    (이하 생략)
```

만약 실행 인자를 사용하고자 한다면 첫 argv 인 실행명 및 실행 경로인 첫번째 인자를 삭제한 것은 argv 에 저장하고 이를 rclpy 모듈의 init 함수에 넘기게 된다. 이때에 C++과는 달리 argparse 모듈을 이용하여 실행 인자를 위한 구문 해석 프로그램을 작성해야한다.

```
def main(argv=sys.argv[1:]):
    (argparse 구문 추가)
    rclpy.init(args=argv)
    (이하 생략)
```

실행 인자의 구문 해석

실행 인자의 구문 해석과 관련한 부분은 Checker 노드의 main 함수 부분에서 구현되어 있기에 관련 코드를 살펴보며 실행 인자를 다루어 볼 것이다. 우선 전체 소스 코드는 다음과 같다.

```
import argparse
import sys

import rclpy

from topic_service_action_rclpy_example.checker.checker import Checker

def main(argv=sys.argv[1:]):
    parser =
    argparse.ArgumentParser(formatter_class=argparse.ArgumentDefaultsHelpFormatter)
    parser.add_argument(
        '-g',
        '--goal_total_sum',
        type=int,
        default=50,
        help='Target goal value of total sum')
    parser.add_argument(
        'argv', nargs=argparse.REMAINDER,
        help='Pass arbitrary arguments to the executable')
    args = parser.parse_args()

    rclpy.init(args=args.argv)
    try:
        checker = Checker()
        checker.send_goal_total_sum(args.goal_total_sum)
        try:
            rclpy.spin(checker)
        except KeyboardInterrupt:
            checker.get_logger().info('Keyboard Interrupt (SIGINT)')
        finally:
            checker.arithmetic_action_client.destroy()
```

```
    checker.destroy_node()
finally:
    rclpy.shutdown()
```

```
if __name__ == '__main__':
    main()
```

실행 인자의 구문해석 프로그램은 파이썬의 argparse 모듈을 이용하여 파서를 선언 후 사용할 실행 인자 값을 지정하는 것이 주를 이루게 된다. 이를 순서대로 나열하면 다음과 같다. 몇 줄 되지 않는 구문이니 순서대로 알아보자.

파서 만들기

우선 argparse 모듈의 ArgumentParser 객체를 parser라는 이름으로 선언하자. 여기서 formatter_class으로 argparse 모듈의 가장 기본적인 형식을 사용하도록 설정하였다.

```
def main(argv=sys.argv[1:]):
    parser =
argparse.ArgumentParser(formatter_class=argparse.ArgumentDefaultsHelpFormatter)
```

인자 추가하기

다음으로 인자 추가하기에 대해 알아보도록 하겠다. 실행 인자로 사용할 인자를 추가하려면 하기와 같이 add_argument() 메서드를 호출하고 인자의 내용을 채운다. 하기 예제에서는 인자 이름으로 줄여서는 '-g', 풀네이밍으로는 '--goal_total_sum'을 사용한다고 선언하였고, 형태로는 int형, 기본 값은 50을 지정하였으며 간단히 지정 인자의 설명을 넣어주었다. 이 설명은 프로그램을 실행할 때 `-h`와 같이 실행 인자에 대한 도움말을 실행하면 볼 수 있는 문구이다.

```
parser.add_argument(
    '-g',
    '--goal_total_sum',
    type=int,
    default=50,
    help='Target goal value of total sum')
```

인자 파싱하기

사용할 인자를 추가했으면 다음으로 parse_args() 메서드를 통해 인자를 파싱하면 된다. 사용 방법은 다음과 같이 매우 간단하다. 이것으로 간단한 사용을 위한 기본 설정은 모두 끝났다.

```
args = parser.parse_args()
```

인자 사용하기

이자를 사용하려면 하기와 같이 인자를 파싱하여 대입한 args의 변수처럼 인자를 사용하면 된다. 예를 들어 add_argument를 통해 인자로 추가했던 '--goal_total_sum'은 `args.goal_total_sum`처럼 사용할 수 있게 된다.

```
checker.send_goal_total_sum(args.goal_total_sum)
```

6.8. ROS2 C++ 프로그래밍

6.8.1. 토픽 퍼블리셔(argument)

토픽 퍼블리셔 역할을 하는 argument 노드의 전체 소스 코드는 아래와 같다.

```
#ifndef ARITHMETIC_ARGUMENT_HPP
#define ARITHMETIC_ARGUMENT_HPP_

#include <chrono>
#include <memory>
#include <string>
#include <utility>

#include "rclcpp/rclcpp.hpp"

#include "msg_srv_action_interface_example/msg/arithmetic_argument.hpp"

class Argument : public rclcpp::Node
{
public:
    using ArithmeticArgument = msg_srv_action_interface_example::msg::ArithmeticArgument;

    explicit Argument(const rclcpp::NodeOptions & node_options = rclcpp::NodeOptions());
    virtual ~Argument();

private:
    void publish_random_arithmetic_arguments();
    void update_parameter();

    float min_random_num_;
    float max_random_num_;

    rclcpp::Publisher<ArithmeticArgument>::SharedPtr arithmetic_argument_publisher_;
    rclcpp::TimerBase::SharedPtr timer_;
    rclcpp::Subscription<rcl_interfaces::msg::ParameterEvent>::SharedPtr
    parameter_event_sub_;
    rclcpp::AsyncParametersClient::SharedPtr parameters_client_;
};

#endif // ARITHMETIC_ARGUMENT_HPP
topic_service_action_rclcpp_example/src/arithmetic/argument.cpp
#include <cstdio>
#include <memory>
#include <string>
#include <utility>
#include <random>

#include "rclcpp/rclcpp.hpp"
#include "rcutils/cmdline_parser.h"

#include "arithmetic/argument.hpp"
```

```

using namespace std::chrono_literals;

Argument::Argument(const rclcpp::NodeOptions & node_options)
: Node("argument", node_options),
min_random_num_(0.0),
max_random_num_(0.0)
{
    this->declare_parameter("qos_depth", 10);
    int8_t qos_depth = this->get_parameter("qos_depth").get_value<int8_t>();
    this->declare_parameter("min_random_num", 0.0);
    min_random_num_ = this->get_parameter("min_random_num").get_value<float>();
    this->declare_parameter("max_random_num", 9.0);
    max_random_num_ = this->get_parameter("max_random_num").get_value<float>();
    this->update_parameter();

    const auto QOS_RKL10V =
        rclcpp::QoS(rclcpp::KeepLast(qos_depth)).reliable().durability_volatile();

    arithmetic_argument_publisher_ =
        this->create_publisher<ArithmetricArgument>("arithmetic_argument", QOS_RKL10V);

    timer_ =
        this->create_wall_timer(1s, std::bind(&Argument::publish_random_arithmetic_arguments,
this));
}

Argument::~Argument()
{
}

void Argument::publish_random_arithmetic_arguments()
{
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<float> distribution(min_random_num_, max_random_num_);

    msg_srv_action_interface_example::msg::ArithmetricArgument msg;
    msg.stamp = this->now();
    msg.argument_a = distribution(gen);
    msg.argument_b = distribution(gen);
    arithmetic_argument_publisher_->publish(msg);

    RCLCPP_INFO(this->get_logger(), "Published argument_a %.2f", msg.argument_a);
    RCLCPP_INFO(this->get_logger(), "Published argument_b %.2f", msg.argument_b);
}

void Argument::update_parameter()
{
    parameters_client_ = std::make_shared<rclcpp::AsyncParametersClient>(this);
    while (!parameters_client_->wait_for_service(1s)) {
        if (!rclcpp::ok()) {
            RCLCPP_ERROR(this->get_logger(), "Interrupted while waiting for the service. Exiting.");
            return;
        }
        RCLCPP_INFO(this->get_logger(), "service not available, waiting again...");
    }

    auto param_event_callback =
        [this](const rcl_interfaces::msg::ParameterEvent::SharedPtr event) -> void
    {
        for (auto & changed_parameter : event->changed_parameters) {

```

```

if (changed_parameter.name == "min_random_num") {
    auto value =
rclcpp::Parameter::from_parameter_msg(changed_parameter).as_double();
    min_random_num_ = value;
} else if (changed_parameter.name == "max_random_num") {
    auto value =
rclcpp::Parameter::from_parameter_msg(changed_parameter).as_double();
    max_random_num_ = value;
}
};

parameter_event_sub_ = parameters_client_->on_parameter_event(param_event_callback);
}

void print_help()
{
    printf("For argument node:\n");
    printf("node_name [-h]\n");
    printf("Options:\n");
    printf("\t-h Help      : Print this help function.\n");
}

int main(int argc, char * argv[])
{
    if (rcutils_cli_option_exist(argv, argv + argc, "-h")) {
        print_help();
        return 0;
    }

    rclcpp::init(argc, argv);

    auto argument = std::make_shared<Argument>();

    rclcpp::spin(argument);

    rclcpp::shutdown();

    return 0;
}

```

먼저 hpp 파일을 보면 ROS 2 를 다루는데 필수적인 라이브러리들이 먼저 선언되어 있다. 그 아래에는 rclcpp API를 담고있는 rclcpp 헤더파일과 인터페이스를 담고있는 헤더파일이 선언되어 있다.

```

#include <chrono>
#include <memory>
#include <string>
#include <utility>

#include "rclcpp/rclcpp.hpp"

#include "msg_srv_action_interface_example/msg/arithmetic_argument.hpp"

```

rclcpp 의 Node 클래스를 상속받는 Arument 클래스에 대해 알아보자. Argument 클래스의 생성자는 rclcpp 의 NodeOptions 를 인자로 받는다. NodeOptions 에는 context, arguments, intra process communication, parameter, allocator 와 같은 Node 생성을 위한 다양한 옵션을 정할 수 있다.

Arguments 클래스에는 rclcpp::Publisher 와 rclcpp::TimerBase 멤버변수가 선언되어 있으며 토픽 메시지에 담을 랜덤 변수의 범위를 정해줄 두 멤버변수도 함께 확인할 수 있다.

```

class Argument : public rclcpp::Node
{
public:
    using ArithmeticArgument = msg_srv_action_interface_example::msg::ArithmeticArgument;
    explicit Argument(const rclcpp::NodeOptions & node_options = rclcpp::NodeOptions());
    virtual ~Argument();

private:
    void publish_random_arithmetic_arguments();
    void update_parameter();

    float min_random_num_;
    float max_random_num_;

    rclcpp::Publisher<ArithmeticArgument>::SharedPtr arithmetic_argument_publisher_;
    rclcpp::TimerBase::SharedPtr timer_;
    rclcpp::Subscription<rcl_interfaces::msg::ParameterEvent>::SharedPtr
    parameter_event_sub_;
    rclcpp::AsyncParametersClient::SharedPtr parameters_client_;
};

그 아래 cmdline_parser 헤더파일은 프로그램 실행시 넘겨받은 인자를 다루는 ROS 2 라이브러리이다.

```

```

#include <cstdio>
#include <memory>
#include <string>
#include <utility>
#include <random>

#include "rclcpp/rclcpp.hpp"
#include "rcutils/cmdline_parser.h"

#include "arithmetic/argument.hpp"

```

Argument 클래스의 생성자를 보자. 부모클래스인 rclcpp::Node를 먼저 선언해야만 한다. 첫번째 인자에는 노드의 이름을 적어주고, 두번째 인자에는 노드 옵션 변수를 명시해주자. 퍼블리쉬를 위한 QoS는 rclcpp::QoS 라이브러리를 이용하여 History 옵션은 KeepLast(depth는 10), Reliability 옵션은 reliable, Durability 옵션은 volatile로 설정하였다. QoS는 아래 publisher를 초기화 할때 두번째 인자로 들어가게 되고, 첫번째 인자에는 메시지 통신에 사용될 토픽명을 적어준다. timer의 경우 1초당 한번씩 publisher_random_arithmetic_arguments 멤버함수를 호출하도록 설정하였다.

```

Argument::Argument(const rclcpp::NodeOptions & node_options)
: Node("argument", node_options),
  min_random_num_(0.0),
  max_random_num_(0.0)
{
    this->declare_parameter("qos_depth", 10);
    int8_t qos_depth = this->get_parameter("qos_depth").get_value<int8_t>();
    this->declare_parameter("min_random_num", 0.0);
    min_random_num_ = this->get_parameter("min_random_num").get_value<float>();
    this->declare_parameter("max_random_num", 9.0);
    max_random_num_ = this->get_parameter("max_random_num").get_value<float>();
    this->update_parameter();

    const auto QOS_RKL10V =
        rclcpp::QoS(rclcpp::KeepLast(qos_depth)).reliable().durability_volatile();

    arithmetic_argument_publisher_ =

```

```

    this->create_publisher<ArithmetricArgument>("arithmetic_argument", QOS_RKL10V);

    timer_ =
        this->create_wall_timer(1s, std::bind(&Argument::publish_random_arithmetric_arguments,
this));
}

publisher_random_arithmetric_arguments 멤버함수를 보자. 해당 함수는 timer에 의해 1초당
한번씩 호출된다. random 라이브러리를 사용하여 ROS 2 파라미터를 통해 얻은 숫자가 저장된
min_random_num, max_random_num 사이의 랜덤한 숫자를 생성하도록 하자.

```

그리고 토픽 메시지 통신에서 사용할 인터페이스(msg)를 선언하자. 해당 인터페이스의 멤버 변수들을 올바르게 채워주고 난뒤, Argument 클래스에서 초기화한 publisher를 통해 해당 인터페이스를 publish 함수를 통해 송신할 수 있다.

그 아래에는 인터페이스를 통해 송신한 랜덤한 숫자를 로그로 표시하는 코드도 확인할 수 있다.

```

void Argument::publish_random_arithmetric_arguments()
{
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<float> distribution(min_random_num_, max_random_num_);

    msg_srv_action_interface_example::msg::ArithmetricArgument msg;
    msg.stamp = this->now();
    msg.argument_a = distribution(gen);
    msg.argument_b = distribution(gen);
    arithmetic_argument_publisher_->publish(msg);

    RCLCPP_INFO(this->get_logger(), "Published argument_a %.2f", msg.argument_a);
    RCLCPP_INFO(this->get_logger(), "Published argument_b %.2f", msg.argument_b);
}

```

6.8.2. 토픽 서브스크라이버(calculator)

Calculator 클래스는 토픽 퍼블리셔 노드와 마찬가지로 rclcpp::node 를 상속하고 있으며 생성자에서 'calculator' 이라는 노드 이름으로 초기화되었다. 그 뒤 위에서와 마찬가지로 QoS 는 rclcpp::QoS 라이브러리를 이용하여 History 옵션은 KeepLast(depth는 10), Reliability 옵션은 reliable, Durability 옵션은 volatile 로 설정하였다.

이제 subscriber 에 대해 보자. subscriber 는 create_subscription 을 통해 초기화되며 해당 함수는 토픽명과 QoS 그리고 콜백함수인자로 구성되어 있다. 토픽명과 QoS 는 Argument 클래스에서 적은 것과 동일하게 적어 주었고, 콜백함수는 std::bind 를 사용한 것이 아닌 람다 표현식[19]를 이용하였다. 람다 표현식은 C++ 14 부터 적용된 문법으로 잘 활용하면 가독성을 높여준다.

콜백함수를 보면 인자를 통해 수신받은 메시지에 접근하여 멤버 변수에 저장하고 있다. 그리고 수신받은 시간과 전달받은 데이터를 로그로 나타내는 코드도 확인할 수 있다.

```

const auto QOS_RKL10V =
    rclcpp::QoS(rclcpp::KeepLast(qos_depth)).reliable().durability_volatile();

arithmetic_argument_subscriber_ = this->create_subscription<ArithmetricArgument>(
    "arithmetic_argument",

```

```

QOS_RKL10V,
[this](const ArithmeticArgument::SharedPtr msg) -> void
{
    argument_a_ = msg->argument_a;
    argument_b_ = msg->argument_b;

    RCLCPP_INFO(
        this->get_logger(),
        "Subscribed at: sec %ld nanosec %ld",
        msg->stamp.sec,
        msg->stamp.nanosec);

    RCLCPP_INFO(this->get_logger(), "Subscribed argument a : %.2f", argument_a_);
    RCLCPP_INFO(this->get_logger(), "Subscribed argument b : %.2f", argument_b_);
}
);

```

6.8.3. 서비스 서버(calculator)

Calculator 클래스는 rclcpp::node 를 상속하고 있으며 생성자에서 'calculator' 라는 노드 이름으로 초기화되었다. arithmetic_argument_server 멤버변수는 rclcpp::Service 타입의 스마트포인터변수로 서비스명과 콜백함수를 인자로 받는 create_service 함수를 통해 실체화 된다. 해당 코드에서는 그림 1 과 같이 arithmetic_operator 서비스명을 사용했고, 콜백함수는 람다 표현식을 이용하여 get_arithmetic_operator를 지정하였다. 람다 표현식이 먼저 정의되어야지만 이를 사용할 수 있기에 코드의 순서가 아래와 같다. 만약 ROS 와 같은 스타일로 코드를 만들고 싶다면 std::bind 를 사용해도 문제없다.

서비스 통신도 DDS를 통해 동작하기에 QoS를 지원한다. create_service 의 세번째 인자에 rmw_qos_profile_t 의 형태로 원하는 설정이 가능하며, 기본적으로는 rmw_qos_profile_services_default 가 설정되어 있다.

get_arithmetic_operator 를 살펴보자. 해당 람다 표현식은 Request 와 Response 인자를 가지고 있어서 함수 내부에서 이를 사용할 수 있다. 우선 argument_operator 멤버 변수에 요청받은 연산자를 저장한다. 그다음 argument_a, argument_b 와 함께 argument_operator 변수를 calculate_given_formula 함수에 넘겨주어 그 결괏값을 리턴받아 response 변수에 저장하여 서비스를 요청한 클라이언트가 이를 받아볼 수 있도록 한다. 그리고 해당 연산식을 ostringstream 라이브러리를 이용해 string 변수에 저장하였다. 저장된 string 변수는 로그로 확인할 수 있다.

```

auto get_arithmetic_operator =
[this]{
    const std::shared_ptr<ArithmeticOperator::Request> request,
    std::shared_ptr<ArithmeticOperator::Response> response) -> void
{
    argument_operator_ = request->arithmetic_operator;
    argument_result_ =
        this->calculate_given_formula(argument_a_, argument_b_, argument_operator_);
    response->arithmetic_result = argument_result_;

    std::ostringstream oss;
    oss << std::to_string(argument_a_) << ' ' <<
        operator_[argument_operator_-1] << ' ' <<

```

```

    std::to_string(argument_b_) << " = " <<
    argument_result_ << std::endl;
    argument_formula_ = oss.str();

    RCLCPP_INFO(this->get_logger(), "%s", argument_formula_.c_str());
};

arithmetic_argument_server_ =
    create_service<ArithmeticOperator>("arithmetic_operator", get_arithmetic_operator);
calculate_given_formula 함수는 상수 a, b 와 연산자 정보를 포함하는 상수 operator를 인자로
받아 그 계산 결괏값을 리턴해주는 역할을 한다. 아래 코드에서 Request 구조체를 통해 미리 저장된
상수와 operator 인자를 비교하여 해당 연산을 진행하는 로직을 확인할 수 있다.

```

```

# Constants
int8 PLUS = 1
int8 MINUS = 2
int8 MULTIPLY = 3
int8 DIVISION = 4

```

```

float Calculator::calculate_given_formula(
    const float & a,
    const float & b,
    const int8_t & operators)
{
    float argument_result = 0.0;
    ArithmeticOperator::Request arithmetic_operator;

    if (operators == arithmetic_operator.PLUS) {
        argument_result = a + b;
    } else if (operators == arithmetic_operator_MINUS) {
        argument_result = a - b;
    } else if (operators == arithmetic_operator_MULTIPLY) {
        argument_result = a * b;
    } else if (operators == arithmetic_operator_DIVISION) {
        argument_result = a / b;
        if (b == 0.0) {
            RCLCPP_ERROR(this->get_logger(), "ZeroDivisionError!");
            argument_result = 0.0;
            return argument_result;
        }
    } else {
        RCLCPP_ERROR(
            this->get_logger(),
            "Please make sure arithmetic operator(plus, minus, multiply, division).");
        argument_result = 0.0;
    }

    return argument_result;
}

```

6.8.4. 서비스 클라이언트(operator)

서비스 서버 역할을 하는 operator 노드의 전체 소스 코드는 아래와 같다.

```

#ifndef ARITHMETIC_OPERATOR_HPP_
#define ARITHMETIC_OPERATOR_HPP_

```

```

#include <chrono>
#include <memory>
#include <string>
#include <utility>
#include <random>

#include "rclcpp/rclcpp.hpp"
#include "msg_srv_action_interface_example/srv/arithmetic_operator.hpp"

class Operator : public rclcpp::Node
{
public:
    using ArithmeticOperator = msg_srv_action_interface_example::srv::ArithmeticOperator;
    explicit Operator(const rclcpp::NodeOptions & node_options = rclcpp::NodeOptions());
    virtual ~Operator();

    void send_request();

private:
    rclcpp::Client<ArithmeticOperator>::SharedPtr arithmetic_service_client_;
};

#endif // ARITHMETIC_OPERATOR_HPP_
topic_service_action_rclcpp_example/src/arithmetic/operator.cpp
#include <fcntl.h>
#include <getopt.h>
#include <termios.h>
#include <unistd.h>
#include <cstdio>
#include <iostream>
#include <memory>
#include <string>
#include <utility>

#include "rclcpp/rclcpp.hpp"
#include "rcutils/cmdline_parser.h"

#include "arithmetic/operator.hpp"

using namespace std::chrono_literals;

Operator::Operator(const rclcpp::NodeOptions & node_options)
: Node("operator", node_options)
{
    arithmetic_service_client_ = this->create_client<ArithmeticOperator>("arithmetic_operator");
    while (!arithmetic_service_client_->wait_for_service(1s)) {
        if (!rclcpp::ok()) {
            RCLCPP_ERROR(this->get_logger(), "Interrupted while waiting for the service.");
            return;
        }
        RCLCPP_INFO(this->get_logger(), "Service not available, waiting again...");
    }
}

Operator::~Operator()
{
}

void Operator::send_request()

```

```

{
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<int> distribution(1, 4);

    auto request = std::make_shared<ArithmeticOperator::Request>();
    request->arithmetic_operator = distribution(gen);

    using ServiceResponseFuture = rclcpp::Client<ArithmeticOperator>::SharedPtr;
    auto response_received_callback = [this](ServiceResponseFuture future) {
        auto response = future.get();
        RCLCPP_INFO(this->get_logger(), "Result %.2f", response->arithmetic_result);
        return;
    };

    auto future_result =
        arithmetic_service_client_->async_send_request(request, response_received_callback);
}

```

hpp 파일에 선언된 Operator 클래스를 보자. Operator 클래스는 rclcpp::Node 클래스를 상속받는 자식 클래스이고, 생성자에서 rclcpp::NodeOptions 를 인자로 받는다. 그리고 서비스 요청을 위한 send_request 함수와 rclcpp::Client 스마트포인터 타입의 멤버변수를 가지고 있다.

```

class Operator : public rclcpp::Node
{
public:
    using ArithmeticOperator = msg_srv_action_interface_example::srv::ArithmeticOperator;

    explicit Operator(const rclcpp::NodeOptions & node_options = rclcpp::NodeOptions());
    virtual ~Operator();

    void send_request();

private:
    rclcpp::Client<ArithmeticOperator>::SharedPtr arithmetic_service_client_;
};

```

Operator 클래스의 생성자 내부를 보면 부모 클래스인 rclcpp::Node 를 노드명과 node_options 인자로 먼저 초기화해 준다. 그리고 create_client 함수를 통해 서비스명을 인자로 받아 rclcpp::Client 를 실체화 시켜준다. 서비스 통신도 DDS 를 통해 동작하기에 QoS 를 지원한다. create_client 함수의 두번째 인자에 rmw_qos_profile_t 의 형태로 원하는 설정이 가능하며, 기본적으로는 rmw_qos_profile_services_default 가 설정되어 있다. 서비스 클라이언트는 서비스 서버가 없다면 요청을 할수 없을 뿐더러 원하는 응답도 얻지 못하기에 같은 서비스 명을 가진 서비스 서버를 기다리는 코드가 함께 따라다닌다.

```

Operator::Operator(const rclcpp::NodeOptions & node_options)
: Node("operator", node_options)
{
    arithmetic_service_client_ = this-
        >create_client<ArithmeticOperator>("arithmetic_operator");
    while (!arithmetic_service_client_->wait_for_service(1s)) {
        if (!rclcpp::ok()) {
            RCLCPP_ERROR(this->get_logger(), "Interrupted while waiting for the service.");
            return;
        }
        RCLCPP_INFO(this->get_logger(), "Service not available, waiting again...");
    }
}

```

실제로 요청을 수행하는 `send_request` 함수를 보자. 랜덤한 연산자를 고르기 위해 1 ~ 4 사이의 랜덤한 숫자를 생성하여 `request` 변수에 저장하게 된다. 그리고 요청에 의한 응답이 왔을 때 불려질 `response_received_callback` 콜백함수를 람다 표현식을 이용하여 정의해보자. 해당 람다 표현식은 인자로 `future`를 가지고 있는데, 이는 C++11의 비동기식 프로그래밍의 `future`, `promise` 개념과 동일하다. 콜백함수가 불려졌다면 `future` 변수를 통해 `response` 값을 저장할 수 있고, 이를 로그로 확인할 수 있다. 코드 마지막줄에서 정의된 `request` 변수와 람다 표현식을 가지고 `async_send_request` 함수를 통해 비동기식으로 서비스 요청을 보내는 것을 확인할 수 있다.

```
void Operator::send_request()
{
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<int> distribution(1, 4);

    auto request = std::make_shared<ArithmeticOperator::Request>();
    request->arithmetic_operator = distribution(gen);

    using ServiceResponseFuture = rclcpp::Client<ArithmeticOperator>::SharedFuture;
    auto response_received_callback = [this](ServiceResponseFuture future) {
        auto response = future.get();
        RCLCPP_INFO(this->get_logger(), "Result %.2f", response->arithmetic_result);
        return;
    };

    auto future_result =
        arithmetic_service_client_->async_send_request(request, response_received_callback);
}
```

6.8.5. 액션 서버

`arithmetic_action_server` 멤버변수는 `rclcpp_action::Server` 타입의 스마트포인터변수이다. 토픽과 서비스 통신을 위한 멤버변수들은 `rclcpp::Publisher`, `rclcpp::Service` 와 같이 `rclcpp` 네임스페이스를 가지는데 반해 액션 통신을 위한 변수들은 `rclcpp_action` 네임스페이스를 가진다. `arithmetic_action_server` 멤버변수는 `node` 정보들과 액션명 그리고 콜백함수들을 인자로 가지는 `create_server` 함수를 통해 실체화된다.

```
arithmetic_action_server_ = rclcpp_action::create_server<ArithmeticChecker>(
    this->get_node_base_interface(),
    this->get_node_clock_interface(),
    this->get_node_logging_interface(),
    this->get_node_waitables_interface(),
    "arithmetic_checker",
    std::bind(&Calculator::handle_goal, this, _1, _2),
    std::bind(&Calculator::handle_cancel, this, _1),
    std::bind(&Calculator::execute_checker, this, _1)
);
```

`handle_goal` 함수는 액션 클라이언트에서 액션 목표(goal)을 요청했을 때 콜백되는 함수이다. 해당 함수 인자로 액션 클라이언트 메시지의 `uuid`와 액션 목표 값을 확인할 수 있으며, 액션 목표 값에 대한 처리를 통해 테스크를 진행할지 그렇지 아니할지를 리턴값[19]으로 보낼 수 있다. 이는 서비스 통신처럼 동작한다.

```
rclcpp_action::GoalResponse Calculator::handle_goal(
    const rclcpp_action::GoalUUID & uuid,
    std::shared_ptr<const ArithmeticChecker::Goal> goal)
{
    (void)uuid;
    (void)goal;
    return rclcpp_action::GoalResponse::ACCEPT_AND_EXECUTE;
}
```

`handle_cancel` 함수는 액션 클라이언트에서 액션 목표 취소를 요청을 했을 때 콜백되는 함수이다. 해당 함수 인자로 `ServerGoalHandle` 타입을 사용하여 진행 중이었던 테스크의 결괏값, 진행상황, 상태 등을 넘겨주는 동작을 수행할 수 있다. 이는 서비스 통신처럼 동작한다.

```
rclcpp_action::CancelResponse Calculator::handle_cancel(
    const std::shared_ptr<GoalHandleArithmeticChecker> goal_handle)
{
    RCLCPP_INFO(this->get_logger(), "Received request to cancel goal");
    (void)goal_handle;
    return rclcpp_action::CancelResponse::ACCEPT;
}
```

`execute_checker` 함수는 액션 클라이언트에서 요청한 액션 목표를 가지고 실제 테스크가 진행되는 함수이다. 해당 함수 내부에서는 `ServerGoalHandle` 타입의 인자를 가지고 액션 목표과 액션 피드백 그리고 액션 결과값에 접근할 수 있다.

함수 내부 로직에서는 `ServerGoalHandle` 클래스의 `get_goal` 멤버 함수를 통해 액션 클라이언트에서 보낸 액션 목표를 가져와서 `Calculator` 클래스의 `argument_result` 멤버 변수값의 합과 비교한 후 만약 그 합이 액션 목표값과 비교하는 모습을 확인할 수 있다. 그리고 반복문이 동작할 때마다 `Calculator` 클래스의 `argument_formula` 멤버 변수를 액션 클라이언트에 `publish`(이는 토픽처럼 동작한다)해주는 모습도 확인할 수 있다.

```
void Calculator::execute_checker(const std::shared_ptr<GoalHandleArithmeticChecker>
goal_handle)
{
    RCLCPP_INFO(this->get_logger(), "Execute arithmetic_checker action!");
    rclcpp::Rate loop_rate(1);

    auto feedback_msg = std::make_shared<ArithmeticChecker::Feedback>();
    float total_sum = 0.0;
    float goal_sum = goal_handle->get_goal()->goal_sum;

    while ((total_sum < goal_sum) && rclcpp::ok()) {
        total_sum += argument_result_;
        feedback_msg->formula.push_back(argument_formula_);
        if (argument_formula_.empty()) {
            RCLCPP_WARN(this->get_logger(), "Please check your formula");
            break;
        }
        RCLCPP_INFO(this->get_logger(), "Feedback: ");
        for (const auto & formula : feedback_msg->formula) {
            RCLCPP_INFO(this->get_logger(), "\t%s", formula.c_str());
        }
        goal_handle->publish_feedback(feedback_msg);
        loop_rate.sleep();
    }

    if (rclcpp::ok()) {
        auto result = std::make_shared<ArithmeticChecker::Result>();
```

```

    result->all_formula = feedback_msg->formula;
    result->total_sum = total_sum;
    goal_handle->succeed(result);
}
}

```

6.8.6. 액션 클라이언트

전체 코드는 아래와 같다.

```

#ifndef CHECKER__CHECKER_HPP_
#define CHECKER__CHECKER_HPP_

#include <memory>
#include <string>
#include <utility>

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

class Checker : public rclcpp::Node
{
public:
    using ArithmeticChecker = msg_srv_action_interface_example::action::ArithmeticChecker;
    using GoalHandleArithmeticChecker =
        rclcpp_action::ClientGoalHandle<ArithmeticChecker>;

    explicit Checker(
        float goal_sum,
        const rclcpp::NodeOptions & node_options = rclcpp::NodeOptions());
    virtual ~Checker();

private:
    void send_goal_total_sum(float goal_sum);

    void get_arithmetic_action_goal(
        std::shared_future<rclcpp_action::ClientGoalHandle<ArithmeticChecker>::SharedPtr>
        future);

    void get_arithmetic_action_feedback(
        GoalHandleArithmeticChecker::SharedPtr,
        const std::shared_ptr<const ArithmeticChecker::Feedback> feedback);

    void get_arithmetic_action_result(
        const GoalHandleArithmeticChecker::WrappedResult & result);

    rclcpp_action::Client<ArithmeticChecker>::SharedPtr arithmetic_action_client_;
};

#endif // CHECKER__CHECKER_HPP_


#include <memory>
#include "checker/checker.hpp"

Checker::Checker(float goal_sum, const rclcpp::NodeOptions & node_options)
: Node("checker", node_options)
{

```

```

arithmetic_action_client_ = rclcpp_action::create_client<ArithmeticChecker>(
    this->get_node_base_interface(),
    this->get_node_graph_interface(),
    this->get_node_logging_interface(),
    this->get_node_waitables_interface(),
    "arithmetic_checker");

send_goal_total_sum(goal_sum);
}

Checker::~Checker()
{
}

void Checker::send_goal_total_sum(float goal_sum)
{
    using namespace std::placeholders;

    if (!this->arithmetic_action_client_) {
        RCLCPP_WARN(this->get_logger(), "Action client not initialized");
    }

    if (!this->arithmetic_action_client_->wait_for_action_server(std::chrono::seconds(10))) {
        RCLCPP_WARN(this->get_logger(), "Arithmetic action server is not available.");
        return;
    }

    auto goal_msg = ArithmeticChecker::Goal();
    goal_msg.goal_sum = goal_sum;

    auto send_goal_options = rclcpp_action::Client<ArithmeticChecker>::SendGoalOptions();
    send_goal_options.goal_response_callback =
        std::bind(&Checker::get_arithmetic_action_goal, this, _1);
    send_goal_options.feedback_callback =
        std::bind(&Checker::get_arithmetic_action_feedback, this, _1, _2);
    send_goal_options.result_callback =
        std::bind(&Checker::get_arithmetic_action_result, this, _1);
    this->arithmetic_action_client_->async_send_goal(goal_msg, send_goal_options);
}

void Checker::get_arithmetic_action_goal(
    std::shared_future<GoalHandleArithmeticChecker::SharedPtr> future)
{
    auto goal_handle = future.get();
    if (!goal_handle) {
        RCLCPP_WARN(this->get_logger(), "Action goal rejected.");
    } else {
        RCLCPP_INFO(this->get_logger(), "Action goal accepted.");
    }
}

void Checker::get_arithmetic_action_feedback(
    GoalHandleArithmeticChecker::SharedPtr,
    const std::shared_ptr<const ArithmeticChecker::Feedback> feedback)
{
    RCLCPP_INFO(this->get_logger(), "Action feedback: ");
    for (const auto & formula : feedback->formula) {
        RCLCPP_INFO(this->get_logger(), "\t%s ", formula.c_str());
    }
}

```

```

void Checker::get_arithmetic_action_result(
    const GoalHandleArithmetricChecker::WrappedResult & result)
{
    switch (result.code) {
        case rclcpp_action::ResultCode::SUCCEEDED:
            RCLCPP_INFO(this->get_logger(), "Action succeeded!");
            RCLCPP_INFO(this->get_logger(), "Action result(all formula): ");
            for (const auto & formula : result.result->all_formula) {
                RCLCPP_INFO(this->get_logger(), "\t%s ", formula.c_str());
            }
            RCLCPP_INFO(this->get_logger(), "Action result(total sum): ");
            RCLCPP_INFO(this->get_logger(), "\t%.2f ", result.result->total_sum);
            break;
        case rclcpp_action::ResultCode::ABORTED:
            RCLCPP_WARN(this->get_logger(), "The action was aborted");
            return;
        case rclcpp_action::ResultCode::CANCELED:
            RCLCPP_WARN(this->get_logger(), "The action was canceled");
            return;
        default:
            RCLCPP_ERROR(this->get_logger(), "Unknown result code");
            return;
    }
    rclcpp::shutdown();
}

```

hpp 파일에 선언된 Checker 클래스를 보자. Checker 클래스는 rclcpp::Node 클래스를 상속받는 자식 클래스이고, 생성자에서 goal_sum 변수와 rclcpp::NodeOptions 를 인자로 받는다. goal_sum 변수의 경우 추후 실행 인자 프로그래밍 강의에서 다루도록 하겠다. 멤버 함수로는 액션 요청을 위한 send_goal_total_sum 함수와 액션 동작 시 호출되는 콜백 함수들 그리고 멤버 변수로 rclcpp_action:: Client 타입의 스마트 포인터를 확인할 수 있다.

```

class Checker : public rclcpp::Node
{
public:
    using ArithmeticChecker = msg_srv_action_interface_example::action::ArithmetricChecker;
    using GoalHandleArithmetricChecker =
rclcpp_action::ClientGoalHandle<ArithmetricChecker>;

    explicit Checker(
        float goal_sum,
        const rclcpp::NodeOptions & node_options = rclcpp::NodeOptions());
    virtual ~Checker();

private:
    void send_goal_total_sum(float goal_sum);

    void get_arithmetic_action_goal(
        std::shared_future<rclcpp_action::ClientGoalHandle<ArithmetricChecker>::SharedPtr>
future);

    void get_arithmetic_action_feedback(
        GoalHandleArithmetricChecker::SharedPtr,
        const std::shared_ptr<const ArithmetricChecker::Feedback> feedback);

    void get_arithmetic_action_result(
        const GoalHandleArithmetricChecker::WrappedResult & result);

    rclcpp_action::Client<ArithmetricChecker>::SharedPtr arithmetic_action_client_;
};

```

Checker 노드의 생성자를 보자. 먼저 부모 클래스인 rclcpp::Node 를 노드명과 node_options 인자로 먼저 초기화해 준다. 그리고 rclcpp_action::create_client 함수를 통해 해당 노드의 인터페이스들과 액션명을 인자로 받아 rclcpp_action::Client 를 실체화 시켜준다. 그리고 액션 목표 값을 보내주는 send_goal_total_sum 함수를 호출한다.

```
Checker::Checker(float goal_sum, const rclcpp::NodeOptions & node_options)
: Node("checker", node_options)
{
    arithmetic_action_client_ = rclcpp_action::create_client<ArithmeticChecker>(
        this->get_node_base_interface(),
        this->get_node_graph_interface(),
        this->get_node_logging_interface(),
        this->get_node_waitables_interface(),
        "arithmetic_checker");

    send_goal_total_sum(goal_sum);
}
```

send_goal_total_sum 함수는 goal_sum 변수를 인자로 받아 액션 목표 값을 보내주는 역할을 한다. 먼저 액션 클라이언트가 실체화 되었는지 확인하고, 액션 서버와 통신할 수 있는지 확인한다. 액션 목표 값을 설정하기 위해서 ArithmeticChecker 액션 인터페이스의 Goal 메시지를 불러와 인자로 받은 goal_sum 변수를 넘겨준다. 그리고 rclcpp_action::Client 의 SendGoalOptions 구조체[21]를 통해 goal_response_callback, feedback_callback, result_callback 함수를 초기화 시켜주고 이를 Goal 메시지와 함께 async_send_goal 함수를 통해 액션 서버에 보낼 수 있다.

```
void Checker::send_goal_total_sum(float goal_sum)
{
    using namespace std::placeholders;

    if (!this->arithmetic_action_client_) {
        RCLCPP_WARN(this->get_logger(), "Action client not initialized");
    }

    if (!this->arithmetic_action_client_->wait_for_action_server(std::chrono::seconds(10))) {
        RCLCPP_WARN(this->get_logger(), "Arithmetic action server is not available.");
        return;
    }

    auto goal_msg = ArithmeticChecker::Goal();
    goal_msg.goal_sum = goal_sum;

    auto send_goal_options = rclcpp_action::Client<ArithmeticChecker>::SendGoalOptions();
    send_goal_options.goal_response_callback =
        std::bind(&Checker::get_arithmetic_action_goal, this, _1);
    send_goal_options.feedback_callback =
        std::bind(&Checker::get_arithmetic_action_feedback, this, _1, _2);
    send_goal_options.result_callback =
        std::bind(&Checker::get_arithmetic_action_result, this, _1);
    this->arithmetic_action_client_->async_send_goal(goal_msg, send_goal_options);
}
```

get_arithmetic_action_goal 함수는 앞서 설정한 SendGoalOptions 의 goal_response_callback 으로 앞서 설명했던 액션 서버의 handle_goal 함수와 연결되어 있다. 이는 서비스 통신처럼 동작하며, 액션 목표 값을 보내고 난 뒤 이에 대한 첫번째 응답을 확인할 수 있다.

```
void Checker::get_arithmetic_action_goal(
    std::shared_future<GoalHandleArithmeticChecker::SharedPtr> future)
```

```

{
    auto goal_handle = future.get();
    if (!goal_handle) {
        RCLCPP_WARN(this->get_logger(), "Action goal rejected.");
    } else {
        RCLCPP_INFO(this->get_logger(), "Action goal accepted.");
    }
}

```

`get_arithmetic_action_feedback` 함수는 앞서 설정한 `SendGoalOptions` 의 `feedback_callback` 으로 앞서 설명했던 액션 서버의 `execute_checker` 함수와 연결되어 있다. 액션 인터페이스의 `feedback` 메시지를 인자로 가지고 있어 액션 피드백을 확인할 수 있다.

```

void Checker::get_arithmetic_action_feedback(
    GoalHandleArithmetiChecker::SharedPtr,
    const std::shared_ptr<const ArithmeticChecker::Feedback> feedback)
{
    RCLCPP_INFO(this->get_logger(), "Action feedback: ");
    for (const auto & formula : feedback->formula) {
        RCLCPP_INFO(this->get_logger(), "\t%s ", formula.c_str());
    }
}

```

`get_arithmetic_action_result` 함수는 앞서 설정한 `SendGoalOptions` 의 `result_callback` 으로 앞서 설명했던 액션 서버의 `execute_checker` 함수와 연결되어 있다. `WrappedResult` 타입[22]의 변수를 인자로 가지고 있어서 액션 결괏값을 확인할 수 있을 뿐만 아니라, 액션 결과 상태에 따른 액션 클라이언트의 대응 로직을 쉽게 개발할 수 있다.

```

void Checker::get_arithmetic_action_result(
    const GoalHandleArithmetiChecker::WrappedResult & result)
{
    switch (result.code) {
        case rclcpp_action::ResultCode::SUCCEEDED:
            RCLCPP_INFO(this->get_logger(), "Action succeeded!");
            RCLCPP_INFO(this->get_logger(), "Action result(all formula): ");
            for (const auto & formula : result.result->all_formula) {
                RCLCPP_INFO(this->get_logger(), "\t%s ", formula.c_str());
            }
            RCLCPP_INFO(this->get_logger(), "Action result(total sum): ");
            RCLCPP_INFO(this->get_logger(), "\t%.2f ", result.result->total_sum);
            break;
        case rclcpp_action::ResultCode::ABORTED:
            RCLCPP_WARN(this->get_logger(), "The action was aborted");
            return;
        case rclcpp_action::ResultCode::CANCELED:
            RCLCPP_WARN(this->get_logger(), "The action was canceled");
            return;
        default:
            RCLCPP_ERROR(this->get_logger(), "Unknown result code");
            return;
    }
    rclcpp::shutdown();
}

```

6.8.7. 실행인자(Arguments) 프로그래밍

checker 노드를 실행시키는 main 함수의 전체 코드를 살펴보자.

```

#include <cstdio>
#include <memory>
#include <string>
#include <utility>

#include "rclcpp/rclcpp.hpp"
#include "rcutils/cmdline_parser.h"

#include "checker/checker.hpp"

void print_help()
{
    printf("For Node node:\n");
    printf("node_name [-h]\n");
    printf("Options:\n");
    printf("\t-h Help      : Print this help function.\n");
}

int main(int argc, char * argv[])
{
    if (rcutils_cli_option_exist(argv, argv + argc, "-h")) {
        print_help();
        return 0;
    }

    rclcpp::init(argc, argv);

    float goal_total_sum = 50.0;
    char * cli_option = rcutils_cli_get_option(argv, argv + argc, "-g");
    if (nullptr != cli_option) {
        goal_total_sum = std::stof(cli_option);
    }
    printf("goal_total_sum : %2.f\n", goal_total_sum);

    auto checker = std::make_shared<Checker>(goal_total_sum);

    rclcpp::spin(checker);

    rclcpp::shutdown();

    return 0;
}

```

먼저 헤더파일을 살펴보면 `rcutils`의 `cmdline_parser` 가 포함되어 있는 것을 확인할 수 있다. 해당 라이브러리를 통해 `main` 문의 인자를 쉽게 확인할 수 있다.

```

#include "rcutils/cmdline_parser.h"

```

`main` 함수 안을 보면 가장 먼저 `rcutils_cli_option_exist` 함수를 이용하여 `-h` 인자가 있는지 확인한다. 만약 `-h` 인자가 있다면 `print_help` 함수를 출력하고 `main` 함수를 빠져나가게 된다. 이를 통해 사용자가 해당 노드를 처음 사용하게 될 때 필요한 정보를 제공해 줄 수 있다.

```

if (rcutils_cli_option_exist(argv, argv + argc, "-h")) {
    print_help();
    return 0;
}

```

다음 줄의 `rclcpp::init` 함수를 통해 `main` 함수로 넘겨받은 `argc`, `argv` 인자를 다시 넘겨주어 `--ros-args` 인자를 `rclcpp` 가 확인할 수 있도록 해주는 모습을 확인할 수 있다.

```
rclcpp::init(argc, argv);
```

`rcutils_cli_get_option` 함수는 실행 인자를 확인하고 그 값을 문자열 포인터로 반환해주는 역할을 한다. 해당 함수를 이용하여 사용자는 쉽게 여러개의 실행 인자를 파싱할 수 있고, 문자열 포인터를 원하는 변수 타입으로 변경하여 노드의 생성 인자로 넘겨줄 수 있다.

```
float goal_total_sum = 50.0;
char * cli_option = rcutils_cli_get_option(argv, argv + argc, "-g");
if (nullptr != cli_option) {
    goal_total_sum = std::stof(cli_option);
}
printf("goal_total_sum : %2.f\n", goal_total_sum);

auto checker = std::make_shared<Checker>(goal_total_sum);
```

7. ROS2 기반 주행로봇 구동하기

7.1. Turtlebot3 개발환경 구축

7.1.1. Turtlebot3 소개

Turtlebot은 ROS 표준 플랫폼 환경에 맞춘 Ground Robot으로 1967년 교육용 컴퓨터 프로그래밍 언어인 Turtle에서 파생되었다. 이 Turtlebot을 통해 ROS 환경에 익숙하지 않은 사람들을 쉽게 가르치고 프로그래밍 언어를 배울 수 있도록 설계되어 있으며 그 이후 ROS의 표준 플랫폼으로 도약하게 되었다.

Trutelbot 시리즈에는 3 가지 버전이 존재하는데 Trutelbot1은 iRobot 사의 Roomba 기반 연구로봇인 Create의 ROS 배포를 위해 Willow Garage에서 개발되었다. 2012년 출시된 Turtlebot2는 iClebo Kobuki를 기반으로 한국의 유진로봇이 개발하였고 2017년 기존의 Turtlebot 시리즈의 부족한 기능 및 사용자의 요구를 추가하여 지금의 Turtlebot3가 출시되었다.

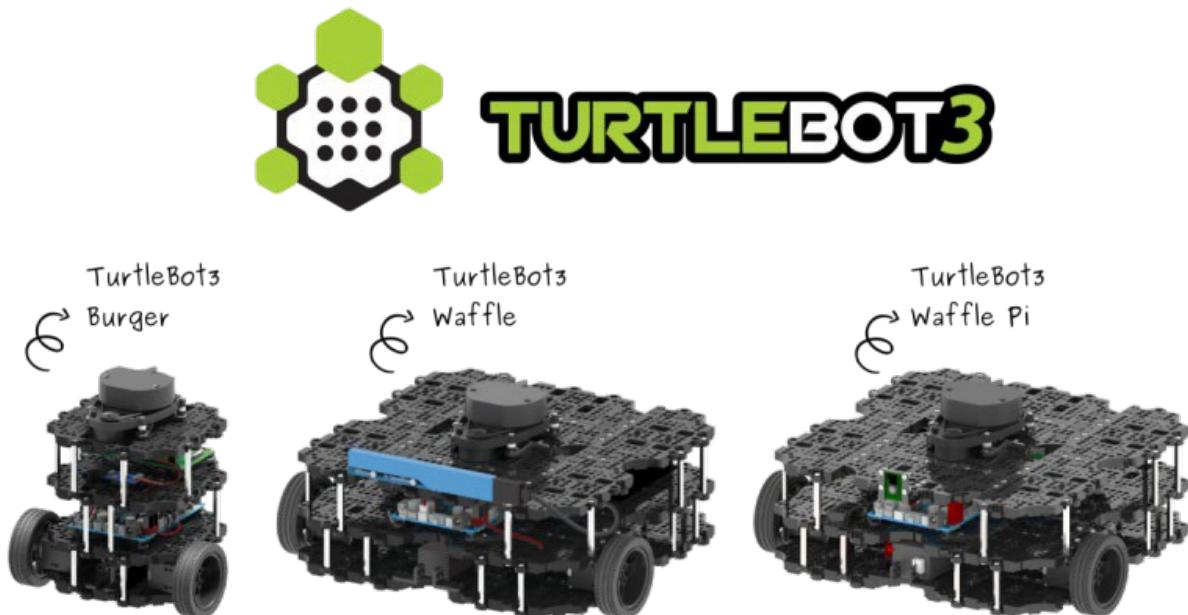


그림 75. Turtlebot3 로고 및 Turtlebot3 종류

Trutelbot3 는

교육, 연구, 취미 및 제품 프로토타입 제작에 사용하기 위해 획기적으로 크기를 줄이고 높은 확장성을 제공하는 것을 목표로 각 구성품을 재구성하고 컴퓨터 및 센서와 같은 옵션 부품을 사용하는 방법에 따라 다양하게 구성할 수 있다는 장점이 있다. 또한 Turtlebot3의 핵심 기술은 SLAM, Navigation, Manipulation 등 가정용 로봇을 위한 시제품 제작에 적합하다는 것이 가장 큰 장점이다.

 WORLD'S MOST POPULAR ROS PLATFORM TurtleBot is the world's most popular open source robot for education and research.	 MODULAR ACTUATOR Easy to assemble, maintain, replace and reconfigure.
 AFFORDABLE COST TurtleBot is the most affordable platform for educations and prototype research & developments.	 OPEN SOURCE SOFTWARE Variety of open source software for the user. You can modify downloaded source code and share it with your friends.
 SMALL SIZE Imagine the TurtleBot in your backpack and bring it anywhere.	 OPEN SOURCE HARDWARE Schematics, PCB Gerber, BOM and 3D CAD data are fully opened to the user.
 EXTENSIBILITY Extend ideas beyond imagination with various SBC, sensor, motor and flexible structure.	 STRONG SENSOR LINEUPS 8MP Camera, Enhanced 360° LiDAR, 9-Axis Inertial Measurement Unit and precise encoder for your robot.

그림 76. Turtlebot3의 특장점

터틀봇 3의 특장점은 다음과 같다.

세계에서 가장 인기있는 ROS 플랫폼

TurtleBot 교육 및 연구 분야에서 가장 널리 사용되는 오픈 소스 로봇입니다. 새로운 세대는 TurtleBot3A 는 small, low cost, fully programmable, ROS based mobile robot. 교육, 연구, 취미 및 제품 프로토 타이핑에 사용하기 위한 것입니다.

저렴한 비용

TurtleBot은 학교, 실험실 및 회사의 비용에 민감한 요구를 충족시키기 위해 개발되었습니다. TurtleBot3은 360 ° 레이저 거리 센서 LDS-01 이 장착 된 SLAM 가능 모바일 로봇 중에서 가장 저렴한 로봇입니다.

작은 크기

TurtleBot3 Burger 의 치수는 138mm x 178mm x 192mm (L x W x H)입니다. 그것의 크기는 전임자의 크기의 대략 1/4 이다. TurtleBot3 을 배낭에 보관하고 프로그램을 개 발하여 어디에서나 테스트 해보는 것을 상상해보십시오.

ROS 표준

TurtleBot 브랜드는 ROS 를 개발하고 유지하는 Open Robotics 에서 관리합니다 . 요즘 ROS는 전 세계의 모든 로봇 공학자들이 사용할 수 있는 플랫폼이 되었습니다. TurtleBot은 기존의 ROS 기반 로봇 구성 요소와 통합 될 수 있지만 TurtleBot3 은 ROS 학습을 시작하려는 저렴한 플랫폼이 될 수 있습니다.

확장성

TurtleBot3 은 사용자가 오픈 소스 임베디드 보드 (제어 보드), 컴퓨터 및 센서와 같은 몇 가지 대체 옵션을 사용하여 기계 구조를 사용자 정의할 것을 권장합니다. TurtleBot3 버거는 2륜 차동 드라이브 유형 플랫폼이지만 자동차, 자전거, 트레일러 등 다양한 방식으로 구조적 및 기계적으로 맞춤 설정할 수 있습니다. 확장 가능한 구조의 다양한 SBC, 센서 및 모터를 통해 상상 이상으로 아이디어를 확장하십시오.

ROS 용 개방 제어 보드

제어 보드는 ROS 통신을 위해 하드웨어 현명하고 소프트웨어 적으로 오픈 소스입니다. 오픈 소스 제어 보드 OpenCR1.0 은 DYNAMIXEL 뿐만 아니라 비용 효율적인 방식으로 기본 인식 작업에 자주 사용되는 ROBOTIS 센서를 제어 할만큼 강력 합니다. 터치 센서, 적외선 센서, 컬러 센서 및 기타 다양한 센서를 사용할 수 있습니다. OpenCR1.0 에는 보드 내부에 IMU 센서가 있어 무수한 응용 프로그램을 정밀하게 제어 할 수 있습니다. 이 보드는 사용 가능한 컴퓨터 장치 라인업을 보강하기 위해 3.3V, 5V, 12V 전원 공급 장치를 갖추고 있습니다.

강력한 센서 라인업

TurtleBot3 버거는 향상된 360 ° LiDAR, 9 축 관성 측정 장치 및 정확한 엔코더를 사용하여 연구 및 개발합니다. TurtleBot3 Waffle 은 동일한 360 ° LiDAR 가 장착되어 있지 만 인식 SDK 가 있는 강력한 Intel® RealSense™ 도 추가로 제안합니다. TurtleBot3 Waffle Pi 는 많이 사용되는 Raspberry Pi Camera 를 사용합니다. 이것은 모바일 로봇을 만들기 위한 최고의 하드웨어 솔루션이 될 것입니다.

오픈 소스

TurtleBot3의 하드웨어, 펌웨어 및 소프트웨어는 사용자가 소스 코드를 다운로드, 수정 및 공유하도록 환영받는 오픈 소스입니다. TurtleBot3의 모든 구성 요소는 저비 용을 위해 사출 성형 플라스틱으로 제조되었지만 3D CAD 데이터는 3D 인쇄에도 사용할 수 있습니다. 3D CAD 데이터는 Full-Cloud 3D CAD 편집기인 Onshape를 통해 릴리스됩니다. 사용자는 데스크톱 PC, 노트북 및 휴대용 장치의 웹 브라우저를 사용하여 액세스 할 수 있습니다. 또한 OpenCR1.0 보드를 자체적으로 만들고자하는 사용자를 위해 회로도, PCB 거버 파일, BOM 및 펌웨어 소스 코드와 같은 OpenCR1.0 보드의 모든 세부 사항이 사용자 및 ROS용 오픈 소스 라이센스하에 완전히 공개되어 있다.

7.1.2. Turtlebot3를 위한 PC Setup

ROS2 관련 의존성 패키지 설치

가장 먼저 colcon package를 설치해주도록 한다.

```
$ sudo apt install python3-colcon-common-extensions
```

설치가 완료되었다면 turtlebot3의 urdf 등을 보고 시뮬레이션을 하기 위해 Gazebo9을 설치해주도록 한다.

```
$ curl -sSL http://get.gazebosim.org | sh  
$ sudo apt remove gazebo11 libgazebo11-dev  
$ sudo apt install gazebo9 libgazebo9-dev  
$ sudo apt install ros-dashing-gazebo-ros-pkgs
```

다음으로 SLAM/Navigation을 위한 의존성 패키지들을 설치하도록 한다.

```
$ sudo apt install ros-dashing-cartographer  
$ sudo apt install ros-dashing-cartographer-ros  
$ sudo apt install ros-dashing-navigation2  
$ sudo apt install ros-dashing-nav2-bringup
```

마지막으로 버전관리를 위한 vcstool을 설치해준다.

```
sudo apt install python3-vcstool
```

Turtlebot3 관련 패키지 설치

다음으로 Turtlebot3를 제어하기 위한 패키지를 다운받을 것이다. Turtlebot3에는 Dynamixel이라는 스텝모터를 사용하여 구동체를 제어하고, turtlebot3_msgs라는 메시지 타입을 통해 데이터를 전송하게 된다. 이와 관련한 의존성 패키지를 다운받고 빌드를 실행한다.

```
$ sudo apt remove ros-dashing-turtlebot3-msgs  
$ sudo apt remove ros-dashing-turtlebot3  
$ mkdir -p ~/turtlebot3_ws/src  
$ cd ~/turtlebot3_ws/src/  
$ git clone -b dashing-devel https://github.com/ROBOTIS-GIT/DynamixelSDK.git  
$ git clone -b dashing-devel https://github.com/ROBOTIS-GIT/turtlebot3_msgs.git  
$ git clone -b dashing-devel https://github.com/ROBOTIS-GIT/turtlebot3.git  
$ colcon build --symlink-install  
$ source ~/.bashrc
```

빌드가 완료되었다면 turtlebot3_ws를 bashrc에 추가하여 노드 실행을 하도록 한다. 또한 다수의 turtlebot이 연결되는 부분을 방지하기 위해 domain id를 추가해주도록 한다.

```
$ echo 'source ~/turtlebot3_ws/install/setup.bash' >> ~/.bashrc
$ echo 'export ROS_DOMAIN_ID=30 #TURTLEBOT3' >> ~/.bashrc
$ source ~/.bashrc
```

7.1.3. Turtlebot3 를 위한 SBC Setup

Ubuntu 18.04 Server 설치

기본적으로 Turtlebot3 에는 Raspberrypi3 b+모델을 제공하고 있다. ROS2 Dashing 설치를 위해서는 Ubuntu server 18.04 를 사용해야 한다. ‘releases.ubuntu.com’ 사이트에 접속하여 ‘ubuntu-18.04.4-preinstalled-server-armhf+raspi3.img.xz’ 파일을 다운받는다.

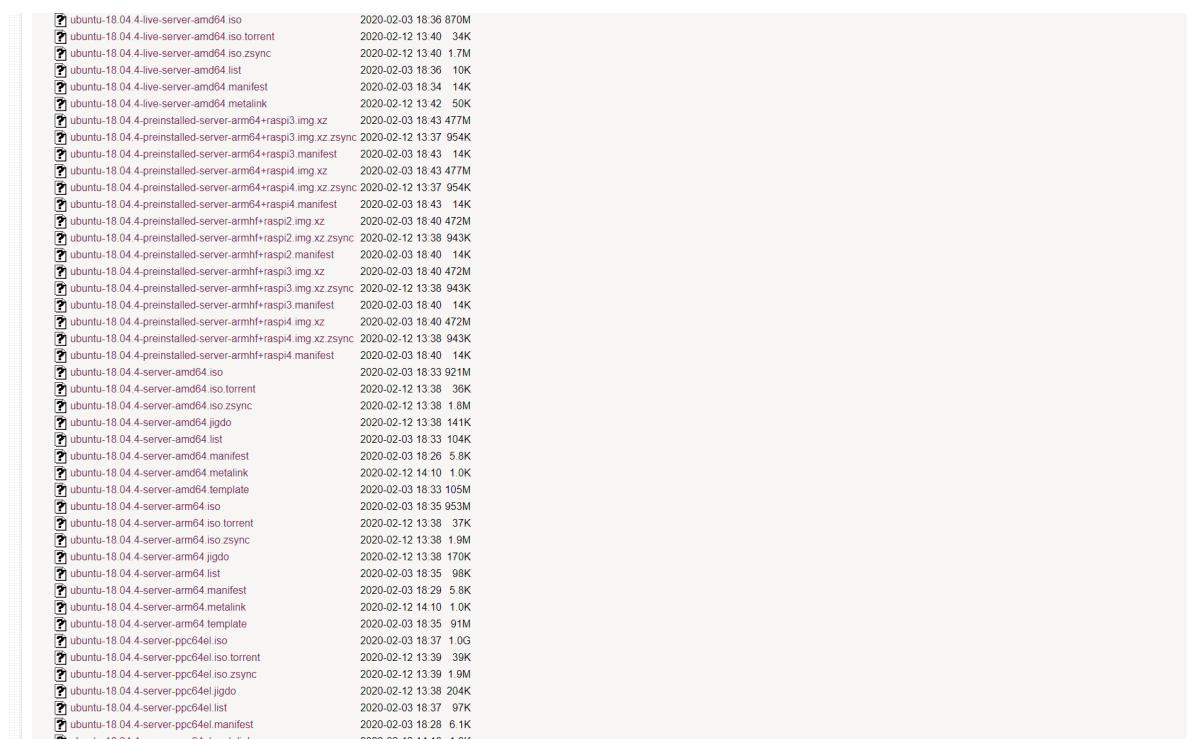


그림 77. releases.ubuntu.com 사이트 내 다운로드 경로

해당 압축파일을 압축해제 하고 기존 ubuntu 설치와 동일하게 부팅디스크를 생성해주도록 한다. 설치가 완료되었다면 라즈베리파이에 micro SD 카드를 삽입하고 hdmi 케이블을 이용해 모니터에 연결한 뒤 부팅을 실행한다. 해당 파일은 라즈베리파이와 관련된 기본적인 라이브러리가 설치되어 있는 이미지로 따로 설치작업 필요없이 바로 부팅이 가능할 것이다. 정상적으로 초기 부팅이 완료되었다면, 로그인을 하는 창이 나오게 되는데 기본으로 id 는 ubuntu, password 도 역시 ubuntu 로 되어있을 것이다.

라즈베리파이 환경설정

가장 먼저 자동 업데이트가 실행되는 부분을 막아주도록 하자. 자동 업데이트 설정파일을 열고 다음과 같이 자동 업데이트 설정을 비활성화 해주도록 한다.

```
$ sudo nano /etc/apt/apt.conf.d/20auto-upgrades
APT::Periodic::Update-Package-Lists "0";
APT::Periodic::Unattended-Upgrade "0";
```

해당 파일을 작성하였다면 Ctrl+x 후 y 를 눌러 저장후 파일을 나온다. 다음으로 wifi 네트워크 설정을 해보도록 하자. 네트워크 설정을 위해 다음과 같은 명령어를 입력해주도록 한다.

```
$ sudo nano /etc/netplan/50-cloud-init.yaml
```

입력이 완료되어서 설정창이 나오게 되었다면 아래와 같이 입력해주도록 한다. 아래의 WIFI_SSID 부분에는 내가 접속하고자 하는 네트워크명을, WIFI_PASSWORD 부분에는 해당 wifi 의 비밀번호를 입력해준다.

```
Network:  
  version: 2  
  renderer: networkd  
  ethernets:  
    eth0:  
      dhcp4: yes  
      dhcp6: yes  
      optional: true  
  wifis:  
    dhcp4: yes  
    dhcp6: yes  
    access-points:  
      WIFI_SSID:  
        Password: WIFI_PASSWORD
```

작성을 완료하였다면 ctrl+x 키를 눌러 종료를 하고 y 키를 눌러 저장을 실행한다. 저장이 완료되었다면 다음 네트워크 설정이 이루어지도록 구성을 적용한 다음 라즈베리파이를 재부팅 해주도록 한다.

```
$ sudo netplan apply  
$ reboot
```

재부팅이 완료되었다면 네트워크가 없는 환경에서 부팅지연이 발생하는 것을 막기 위해 마스크를 설정해주도록 한다. 또한 잠자기 모드와 최대절전모드에 진입하지 않도록 설정해준다.

```
$ systemctl mask systemd-networkd-wait-online.service  
$ sudo systemctl mask sleep.target suspend.target hibernate.target hybrid-sleep.target  
마지막으로 라즈베리파이에 원격으로 접속하기 위해 SSH 설치 및 활성화를 해주도록 한다.
```

```
$ sudo apt install ssh  
$ sudo systemctl enable --now ssh  
$ reboot
```

정상적으로 재부팅이 이루어졌다면 ssh 를 사용하여 원격 PC 에서 접속을 실행하여 정상적으로 접속이 이루어지는지 확인해보도록 한다.

마지막으로 스왑 공간을 추가해주도록 하자. Ubuntu server 의 preinstalled 의 경우 기본적으로 스왑공간이 할당되어있지 않다. 또한 라즈베리파이 3b+의 경우 1Gb 의 메모리용량만 보유하고 있기 때문에 turtlebot3 의 빌드를 수행하기에는 부족한 상황이다. 이에 스왑공간을 2GB 정도로 할당하여 안정적인 빌드가 이루어질 수 있도록 하자.

```
$ sudo swapoff /swapfile  
$ sudo fallocate -l 2G /swapfile  
$ sudo chmod 600 /swapfile  
$ sudo mkswap /swapfile  
$ sudo swapon /swapfile  
$ sudo nano /etc/fstab
```

/etc/fstab 가 정상적으로 열리게 되었다면 파일 가장 하단에 다음과 같이 입력해주도록 한다.

```
/swapfile swap swap defaults 0 0
```

작성 후 **ctrl+x** 를 눌러 종료 명령을 보낸 후 **y** 키를 눌러 저장을 실시한다. 정상적으로 저장이 되었다면 다음 명령어를 통해 스왑 공간이 할당되었는지 확인해보도록 한다.

```
$ sudo free -h
      total    used     free   shared  buff/cache available
Mem:      912M     97M    263M     4.4M    550M     795M
Swap:      2.0G      0B     2.0G
```

ROS Dashing Diademeta 설치

스왑공간이 할당되었다면 이제 ROS2 를 설치해보도록 할 것이다. ROS2 를 설치하는 방법은 크게 두가지가 있다. 첫번째로 Debian package 를 사용하여 설치하는 방법, 두번째로는 소스코드를 통해 직접 빌드를 하는 방법이 있다. 첫번째 방법을 수행하기 위해 우리는 ubuntu server 를 설치한 것이다. 만약 ubuntu 가 아닌 Rasbian OS 또는 WebOS 등 타 OS 를 사용해야 하는 경우 소스코드를 통해 빌드를 진행해야만 한다. 본 내용에서는 첫번째 방법인 Debian Package 를 통해 설치를 하는 방법을 사용할 것이다.

가장 먼저 소프트웨어 업데이트 및 업그레이드를 실시한다.

```
$ sudo apt update && sudo apt upgrade
```

업데이트 및 업그레이드가 완료되었다면 ROS Dashing Diademeta 의 리포지토리를 추가해줄 것이다. 이를 위해 public key 값을 라즈베리파이에 다운로드 해주도록 한다.

```
$ curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc | sudo apt-key add -
다운로드가 완료되었다면 리포지토리 주소를 추가한다.
```

```
$ sudo sh -c 'echo "deb [arch=amd64,arm64] http://packages.ros.org/ros2/ubuntu
`lsb_release -cs` main" > /etc/apt/sources.list.d/ros2-latest.list'
```

리포지토리를 추가해주었다면 **apt-get update** 를 통해 리포지토리를 최신화 해주고 ROS Dashing 을 설치한다. 라즈베리파이에서는 Rviz2, RQT, Gazebo 등 GUI tool 을 사용하지 않을 것이기 때문에 base 버전을 설치해주도록 한다.

```
$ sudo apt update
$ sudo apt install ros-dashing-ros-base
```

설치가 완료되었다면 ROS 및 turtlebot3 용 의존성 패키지를 설치 및 빌드를 수행한다.

```
$ sudo apt install python3-argcomplete python3-colcon-common-extensions libboost-system-dev build-essential
$ sudo apt install ros-dashing-hls-lfcd-lds-driver
$ sudo apt install ros-dashing-turtlebot3-msgs
$ sudo apt install ros-dashing-dynamixel-sdk
$ mkdir -p ~/turtlebot3_ws/src && cd ~/turtlebot3_ws/src
$ git clone -b dashing-devel https://github.com/ROBOTIS-GIT/turtlebot3.git
$ cd ~/turtlebot3_ws/src/turtlebot3
$ rm -r turtlebot3_cartographer turtlebot3_navigation2
$ cd ~/turtlebot3_ws/
$ echo 'source /opt/ros/dashing/setup.bash' >> ~/.bashrc
$ source ~/.bashrc
$ colcon build --symlink-install --parallel-workers 1
$ echo 'source ~/turtlebot3_ws/install/setup.bash' >> ~/.bashrc
$ echo 'export ROS_DOMAIN_ID=30 #TURTLEBOT3' >> ~/.bashrc
$ source ~/.bashrc
```

7.1.4. Turtlebot3를 위한 OpenCR 설정

Turtlebot3의 2 층 layer를 보면 파란색의 OpenCR 보드가 설치되어 있을 것이다. OpenCR은 ROS 임베디드 시스템을 위해 개발된 오픈소스 기반의 하드웨어로, PCB 거버, BOM, turtlebot3, Manipulator 등을 위한 소스코드를 공개하고 오픈소스 개발자들을 통해 자유롭게 커스터마이징이 이루어질 수 있도록 만들어져 있다.

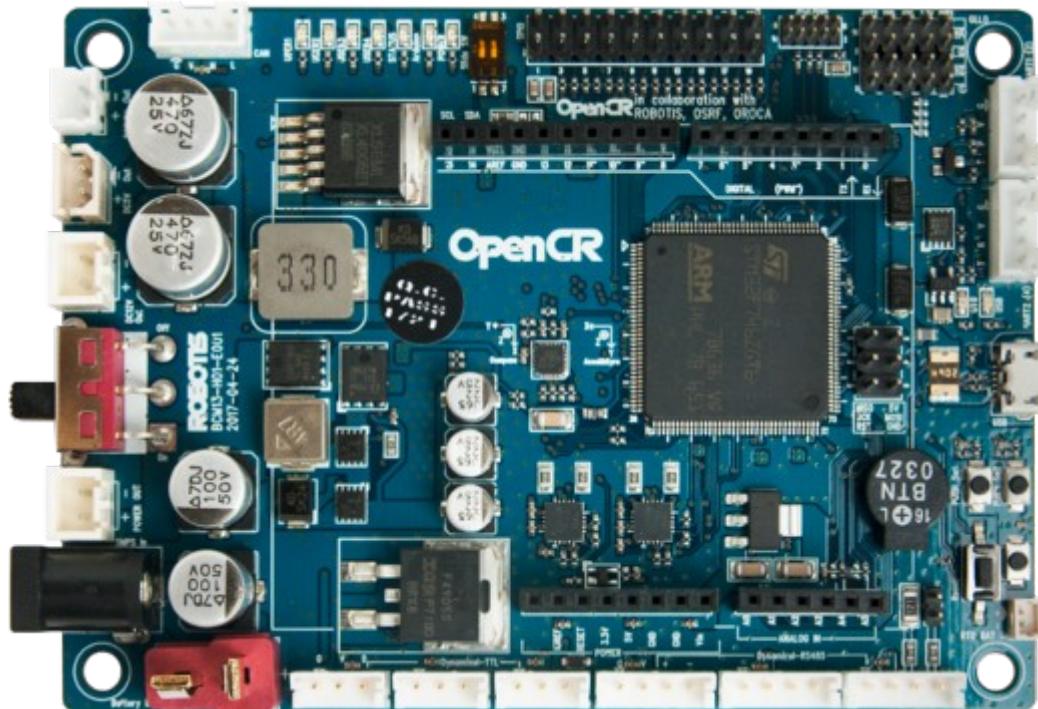


그림 78. OpenCR 구조도

OpenCR1.0 보드 내부의 STM32F7 시리즈 칩은 플로팅 포인트 유닛이 있는 매우 강력한 ARM Cortex-M7을 기반으로 구성되어 있으며 OpenCR1.0의 개발 환경은 아두이노 IDE와 어린 학생들을 위한 Scratch에서 전문가들을 위한 전통적인 펌웨어 개발까지 폭넓게 개방되어 있다.

해당 OpenCR을 통해 turtlebot3의 구동체인 Dynamixel을 제어할 것이며, 이를 위해서는 OpenCR 보드에 관련 펌웨어를 업로드 해주어야 한다. 이러한 펌웨어를 업로드해보도록 하자.

가장 먼저 라즈베리파이에 펌웨어 업로드를 위한 의존성 패키지들을 설치해주도록 한다.

```
$ sudo dpkg --add-architecture armhf  
$ sudo apt-get update  
$ sudo apt-get install libc6:armhf
```

다음으로 robot platform과 관련된 명령어를 입력해주도록 한다. 만약 구동하고자 하는 모델이 turtlebot3 burger라면 다음과 같이 bash에 export를 해주도록 한다.

```
$ export OPENCR_PORT=/dev/ttyACM0  
$ export OPENCR_MODEL=burger
```

만약 본인이 사용하는 turtlebot 구동체가 waffle 또는 waffle pi라면 다음과 같이 입력해주도록 한다.

```
$ export OPENCR_PORT=/dev/ttymcu0
```

```
$ export OPENCR_MODEL=waffle
```

다음으로 펌웨어 및 펌웨어 업로더 파일을 다운로드 한 다음 압축을 해제한다.

```
$ wget https://github.com/ROBOTIS-GIT/OpenCR-Binaries/raw/master/turtlebot3/ROS2/latest/  
opencr_u pdate.tar.bz2  
$ tar -xjf ./opencr_update.tar.bz2
```

압축해제가 완료되었다면 펌웨어를 OpenCR 보드에 업로드를 해준다.

```
$ cd ~/opencr_update  
$ ./update.sh $OPENCR_PORT $OPENCR_MODEL.opencr
```

만약 정상적으로 업로드가 되었다면 다음과 같은 화면을 볼 수 있을 것이다.

```
turtlebot@turtlebot:~  
turtlebot@turtlebot: ~ 98x55  
[ ] turtlebot@turtlebot:~$ export OPENCR_PORT=/dev/ttyACM0  
[ ] turtlebot@turtlebot:~$ export OPENCR_MODEL=burger  
[ ] turtlebot@turtlebot:~$ rm -rf ./opencr_update.tar.bz2  
[ ] turtlebot@turtlebot:~$ wget https://github.com/ROBOTIS-GIT/OpenCR/raw/develop/arduino/opencr_release/shell_update/opencr_update.tar.bz2 && tar -xvf opencr_update.tar.bz2 && cd ./opencr_update && ./update.sh $OPENCR_PORT $OPENCR_MODEL.opencr && cd ..  
- 2018-03-05 11:42:25-- https://github.com/ROBOTIS-GIT/OpenCR/raw/develop/arduino/opencr_release/shell_update/opencr_update.tar.bz2  
Resolving github.com (github.com)... 192.30.255.112, 192.30.255.113  
Connecting to github.com (github.com)|192.30.255.112|:443... connected.  
HTTP request sent, awaiting response... 302 Found  
Location: https://raw.githubusercontent.com/ROBOTIS-GIT/OpenCR/develop/arduino/opencr_release/shell_update/opencr_update.tar.bz2 [following]  
- 2018-03-05 11:42:25-- https://raw.githubusercontent.com/ROBOTIS-GIT/OpenCR/develop/arduino/opencr_release/shell_update/opencr_update.tar.bz2  
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 151.101.228.133  
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|151.101.228.133|:443... connected.  
HTTP request sent, awaiting response... 200 OK  
Length: 293362 (286K) [application/octet-stream]  
Saving to: 'opencr_update.tar.bz2'  
  
opencr_update.tar.bz2 100%[=====] 286.49K 468KB/s in 0.6s  
2018-03-05 11:42:27 (468 KB/s) - 'opencr_update.tar.bz2' saved [293362/293362]  
  
opencr_update/burger_turtlebot3_core.ino.bin  
opencr_update/waffle_turtlebot3_core.ino.bin  
opencr_update/opencr_ld_shell_arm  
opencr_update/update.sh  
opencr_update/  
opencr_update/opencr_ld_shell_x86  
opencr_update/waffle.opencr  
opencr_update/burger.opencr  
opencr_update/released_1.0.17.txt  
armv7l  
arm  
OpenCR Update Start..  
opencr_ld_shell ver 1.0.0  
opencr_ld_main  
[ ] file name : burger.opencr  
[ ] file size : 172 KB  
[ ] fw_name : burger  
[ ] fw_ver : 1.0.17  
[OK] Open port : /dev/ttyACM0  
[ ]  
[ ] Board Name : OpenCR R1.0  
[ ] Board Ver : 0x17020800  
[ ] Board Rev : 0x00000000  
[OK] flash_erase : 0.92s  
[OK] flash_write : 1.84s  
[OK] CRC Check : 11A1E12 11A1E12 , 0.005000 sec  
[OK] Download  
[OK] jump_to_fw  
turtlebot@turtlebot:~$
```

그림 79. OpenCR 펌웨어 업로드 결과 화면

7.2. Turtlebot3 제어 Node 실행 및 기본 조작하기

위의 7.1.4.까지를 통해 우리는 turtlebot3를 제어할 수 있도록 기본적인 환경설정이 끝나게 되었다. 이제 본격적으로 turtlebot 구동 Node를 실행하고 관련 topic 및 센서 데이터를 시각화해보도록 할 것이다.

먼저 turtlebot3의 ssh로 접속을 하도록 한다.

```
$ ssh ubuntu@{IP_ADDRESS_OF_RASPBERRY_PI}
```

다음으로 Node를 실행하기 전 해당 구동체가 어떤모델인지를 bash에 입력해주도록 한다. 하단의 \${TB3_MODEL} 부분에 본인의 turtlebot이 burger인지 waffle인지 waffle_pi인지 확인 후 맞는 값을 입력한다. 그 뒤 turtlebot3_bringup 패키지 안에 있는 robot.launch.py 파일을 실행해주도록 한다.

```
$ export TURTLEBOT3_MODEL=${TB3_MODEL}  
$ ros2 launch turtlebot3_bringup robot.launch.py
```

만약 정상적으로 실행이 되었다면 다음과 같은 메시지를 확인할 수 있을 것이다.

```
[INFO] [launch]: All log files can be found below /home/ubuntu/.ros/log/2019-08-19-01-24-19-  
009803-ubuntu-15310  
[INFO] [launch]: Default logging verbosity is set to INFO  
urdf_file_name : turtlebot3_burger.urdf  
[INFO] [robot_state_publisher-1]: process started with pid [15320]  
[INFO] [hlds_laser_publisher-2]: process started with pid [15321]  
[INFO] [turtlebot3_ros-3]: process started with pid [15322]  
[robot_state_publisher-1] Initialize urdf model from file:  
/home/ubuntu/turtlebot_ws/install/turtle  
bot3_description/share/turtlebot3_description/urdf/turtlebot3_burger.urdf  
[robot_state_publisher-1] Parsing robot urdf xml string.  
[robot_state_publisher-1] Link base_link had 5 children  
[robot_state_publisher-1] Link caster_back_link had 0 children  
[robot_state_publisher-1] Link imu_link had 0 children  
[robot_state_publisher-1] Link base_scan had 0 children  
[robot_state_publisher-1] Link wheel_left_link had 0 children  
[robot_state_publisher-1] Link wheel_right_link had 0 children  
[robot_state_publisher-1] got segment base_footprint  
[robot_state_publisher-1] got segment base_link  
[robot_state_publisher-1] got segment base_scan  
[robot_state_publisher-1] got segment caster_back_link  
[robot_state_publisher-1] got segment imu_link  
[robot_state_publisher-1] got segment wheel_left_link  
[robot_state_publisher-1] got segment wheel_right_link  
[turtlebot3_ros-3] [INFO] [turtlebot3_node]: Init TurtleBot3 Node Main  
[turtlebot3_ros-3] [INFO] [turtlebot3_node]: Init DynamixelSDKWrapper  
[turtlebot3_ros-3] [INFO] [DynamixelSDKWrapper]: Succeeded to open the  
port(/dev/ttyACM0)!  
[turtlebot3_ros-3] [INFO] [DynamixelSDKWrapper]: Succeeded to change the baudrate!  
[robot_state_publisher-1] Adding fixed segment from base_footprint to base_link  
[robot_state_publisher-1] Adding fixed segment from base_link to caster_back_link  
[robot_state_publisher-1] Adding fixed segment from base_link to imu_link  
[robot_state_publisher-1] Adding fixed segment from base_link to base_scan  
[robot_state_publisher-1] Adding moving segment from base_link to wheel_left_link  
[robot_state_publisher-1] Adding moving segment from base_link to wheel_right_link  
[turtlebot3_ros-3] [INFO] [turtlebot3_node]: Start Calibration of Gyro  
[turtlebot3_ros-3] [INFO] [turtlebot3_node]: Calibration End  
[turtlebot3_ros-3] [INFO] [turtlebot3_node]: Add Motors  
[turtlebot3_ros-3] [INFO] [turtlebot3_node]: Add Wheels  
[turtlebot3_ros-3] [INFO] [turtlebot3_node]: Add Sensors  
[turtlebot3_ros-3] [INFO] [turtlebot3_node]: Succeeded to create battery state publisher  
[turtlebot3_ros-3] [INFO] [turtlebot3_node]: Succeeded to create imu publisher  
[turtlebot3_ros-3] [INFO] [turtlebot3_node]: Succeeded to create sensor state publisher  
[turtlebot3_ros-3] [INFO] [turtlebot3_node]: Succeeded to create joint state publisher  
[turtlebot3_ros-3] [INFO] [turtlebot3_node]: Add Devices  
[turtlebot3_ros-3] [INFO] [turtlebot3_node]: Succeeded to create motor power server  
[turtlebot3_ros-3] [INFO] [turtlebot3_node]: Succeeded to create reset server  
[turtlebot3_ros-3] [INFO] [turtlebot3_node]: Succeeded to create sound server  
[turtlebot3_ros-3] [INFO] [turtlebot3_node]: Run!
```

```
[turtlebot3_ros-3] [INFO] [diff_drive_controller]: Init Odometry  
[turtlebot3_ros-3] [INFO] [diff_drive_controller]: Run!
```

Turtlebot3를 통해 발행되고 있는 topic 들은 다음과 같은 것들이 있다.

```
$ ros2 topic list  
/battery_state  
/cmd_vel  
/imu  
/joint_states  
/magnetic_field  
/odom  
/parameter_events  
/robot_description  
/rosout  
/scan  
/sensor_state  
/tf  
/tf_static
```

Turtlebot3를 통해 사용할 수 있는 service 목록은 다음과 같다.

```
$ ros2 service list  
/diff_drive_controller/describe_parameters  
/diff_drive_controller/get_parameter_types  
/diff_drive_controller/get_parameters  
/diff_drive_controller/list_parameters  
/diff_drive_controller/set_parameters  
/diff_drive_controller/set_parameters_atomically  
/hlds_laser_publisher/describe_parameters  
/hlds_laser_publisher/get_parameter_types  
/hlds_laser_publisher/get_parameters  
/hlds_laser_publisher/list_parameters  
/hlds_laser_publisher/set_parameters  
/hlds_laser_publisher/set_parameters_atomically  
/launch_ros/describe_parameters  
/launch_ros/get_parameter_types  
/launch_ros/get_parameters  
/launch_ros/list_parameters  
/launch_ros/set_parameters  
/launch_ros/set_parameters_atomically  
/motor_power  
/reset  
/sound  
/turtlebot3_node/describe_parameters  
/turtlebot3_node/get_parameter_types  
/turtlebot3_node/get_parameters  
/turtlebot3_node/list_parameters  
/turtlebot3_node/set_parameters  
/turtlebot3_node/set_parameters_atomically
```

이제 turtlebot3를 시각화 해보도록 하자. 센서 데이터 정보들은 rviz를 통해 시각화가 가능하며 다음과 같은 명령어를 입력해 turtlebot3 모델과, tf, LiDAR를 통해 수집되는 정보드를 시각화하여 볼 수 있을 것이다.

```
$ ros2 launch turtlebot3_bringup rviz2.launch.py
```

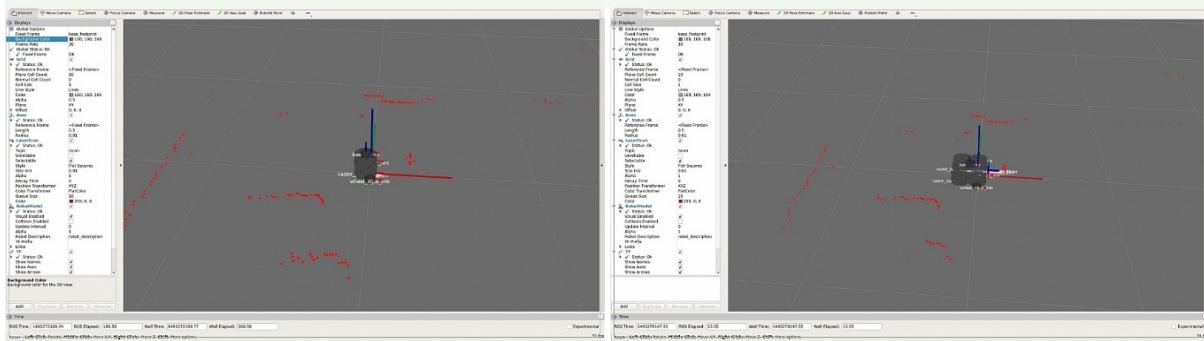


그림 79. Rviz 실행화면

정상적으로 Rviz 가 수행되었다면 이제 간단하게 turtlebot3 를 이동시켜보도록 하자. Turtlebot3 에서는 기본적으로 teleoperation Node 를 제공하고 있으며, 이를 통해 turtlebot 을 원하는 위치로 이동시킬 수 있다.

```
$ export TURTLEBOT3_MODEL=${TB3_MODEL}
$ ros2 run turtlebot3_teleop teleop_keyboard
```

노드가 정상적으로 실행되면 터미널에서 다음과 같은 화면이 나타나게 되며 터미널에 값을 입력하여 정상적으로 이동하는지 확인해보도록 하자.

Control Your Turtlebot3

Moving around

```
w  
a s d  
x
```

w/x : increase/decrease linear velocity (Burger : ~ 0.22, Waffle and Waffle Pi : ~ 0.26)

a/d : increase/decrease angular velocity (Burger : ~ 2.84, Waffle and Waffle Pi : ~ 1.82)

space key, s : force stop

CTRL-C to quit

7.3. Turtlebot3 를 이용한 SLAM & Navigation

7.3.1. SLAM 이란?

로봇이 미지의 공간 상에서 주행을 시작하는 경우 주변환경에 대한 아무런 정보가 없다. 따라서 로봇이 위치정보에 기반하여 작업을 수행하기 위해서는 센서정보를 이용하여 환경에 대한 지도를 작성하고, 동시에 작성된 지도로부터 로봇의 현재 위치를 추정하는 SLAM (Simultaneous Localization And Mapping) 과정이 필요하다. 최근 성공적으로 상용화된 위치인식 기능이 탑재된 청소로봇의 경우를 예로 들면, 전원을 켰을 때의 위치를 기준점으로 설정하여 장착된 센서(예, 비전센서, 거리센서)로부터 수집한 정보로 지도를 작성하는 동시에, 자신의 위치를 실시간으로 추정하여 청소한 구역과 청소해야 할 구역을 구분한다.

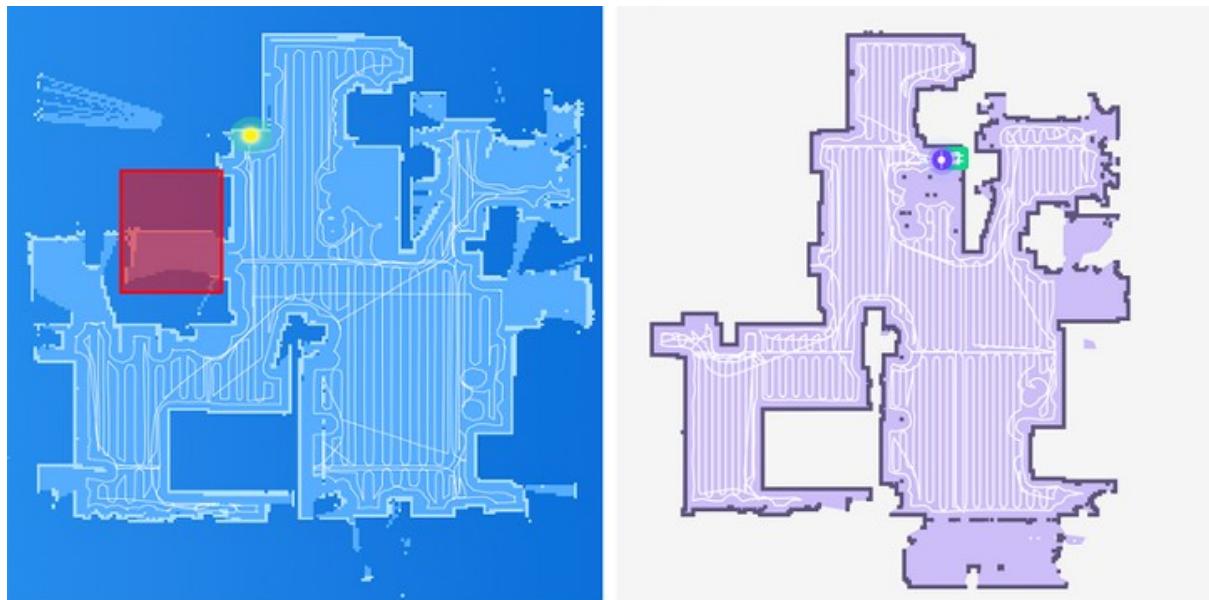


그림 80. 청소로봇의 위치 추정을 위한 지도 및 경로 정보

SLAM에는 거리센서, 비전센서 등 다양한 종류의 센서가 사용된다. 거리센서는 대표적으로 레이저스캐너, 적외선스캐너, 초음파센서, 라이다(LIDAR), 레이더(RADAR) 등이 있으며, 비전센서에는 스테레오카메라, 모노카메라, 전방향카메라, Kinect 등이 포함된다. 거리센서는 쉽게 물체까지의 거리정보를 얻을 수는 있지만, 활용할 수 있는 정보의 종류가 제한적인 단점이 있다.

Velodine	SLAMTEC	Quster	Quanergy	Ibeo
2D SLAM			3D SLAM	
Gmapping	HectorSLAM	KartoSLAM	Loam	Lego-Loam
로봇에서 가장 많이 활용	2D SLAM 및 3D 네비게이션 결합	그래프 기반 SLAM	3D LiDAR를 사용한 실시간 SLAM 시스템	3D LiDAR와 IMU 데이터를 포인트클라우드로 입력 실시간을 6D 위치 추정
LagoSLAM	CoreSLAM	Cartographer	Cartographer	IMLS-SLAM
비선형 함수 활용 그래프 기반 SLAM	최소 성능 손실 알고리즘	Google의 SLAM 시스템 하위 맵과 loop closure 채택	Google의 SLAM 시스템 하위 맵과 loop closure 채택	Scan-to 모델 매칭 프레임워크 기반 SLAM

그림 81. LiDAR를 활용한 SLAM 방법들

반면에, 비전센서는 영상정보의 가공을 통해 거리 및 인식된 물체 등 다양한 종류의 정보를 획득할 수 있어 근래에 각광을 받고 있으나, 대부분 많은 연산량을 필요로 한다. 보통 이동로봇 분야에서 이러한 센서정보는 주로 오도메트리(예, 엔코더, 관성센서) 정보와 융합하여 사용하지만, 비전센서만으로 SLAM을 수행하는 연구도 활발히 진행되고 있다.

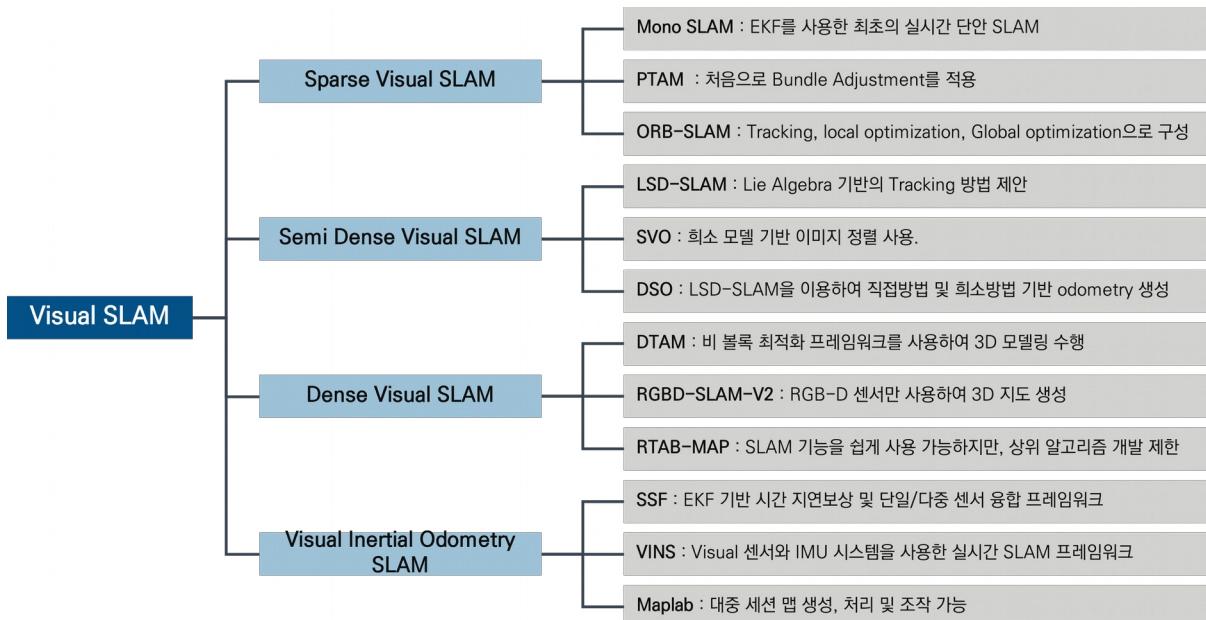


그림 82. Visual SLAM의 다양한 종류들

이러한 다양한 SLAM 방법들은 실제 산업분야에서 활용되고 있지만 몇가지 기술적 Issue들을 해결해야 하는 상황이다. 다음은 SLAM을 개발 및 활용시 발생되는 몇가지 ISSUE 사항들이다.

실제값과 상당한 편차를 초래하는 위치추정 오차 누적 문제

SLAM은 일부 오차 범위가 포함된 순차적 이동 정보를 추정한다. 이 오차는 시간이 지나면서 누적되고, 따라서 실제값과는 상당한 편차가 발생하게 되는데, 이렇게 되면 지도 데이터의 붕괴 또는 왜곡이 발생하여 후속 검색이 어려워지는 문제가 발생한다. 사각형 통로 주변을 주행하는 경우를 예로 들어 보면

오류가 누적되면서 로봇의 출발점과 끝점이 더는 일치하지 않게 되는 문제가 발생한다. 이를 우리는 Loop Closure 문제라고 한다. 이와 같은 자세 추정 오류는 실제 Ground Truth를 알지 못하는 주행로봇에선 불가피한 문제이다. 루프 폐쇄를 감지하고 누적된 오류를 정정하거나 상쇄하는 방법을 파악하는 것이 중요합니다.

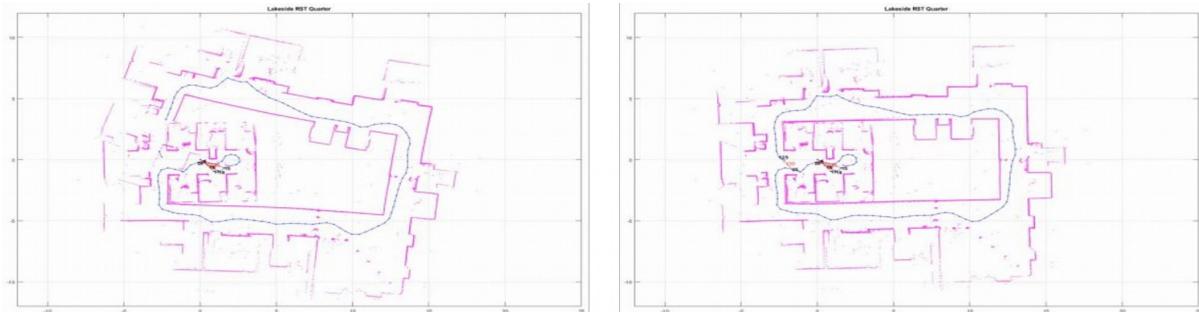


그림 83. Loop Closure 가 정상적으로 이루어지지 않은 지도(좌측), Loop Closure 가 정상적으로 이루어진 지도(우측)

이러한 문제에 대한 한가지 대응책은 이전에 갔던 장소의 몇몇 특징을 표지물로서 기억하여 위치추정 오류를 최소화하는 것이다. 오류 최소화를 최적화 문제로 풀면 더 정확한 지도 데이터 생성이 가능해질 수도 있다. 이러한 종류의 최적화를 시각적 SLAM의 번들 조정(Bundle Adjustment)이라고 한다.

위치추정 실패 및 지도상 위치를 상실하는 상황 발생

영상 및 포인트 클라우드 매핑 시 로봇의 이동 특성은 고려되지 않는다. 경우에 따라서 이러한 접근법으로 인해 불연속적인 위치추정값이 생성될 가능성이 있다. 예를 들자면 1m/s로 이동하는 로봇이 갑자기 10m 앞으로 급속 이동하는 계산 결과가 표시되는 경우를 들 수 있다. 이러한 유형의 위치추정 실패 문제는 복원 알고리즘을 사용하거나 이동 모델과 다수의 센서를 융합하여 센서 데이터에 기반한 계산을 수행함으로써 방지할 수 있다.

이동 모델에 센서 융합을 사용하는 방법에는 여러 가지가 있는데 널리 사용되는 방법은 위치추정에 칼만 필터링을 사용하는 것이다. 대부분의 차동 구동 로봇 및 사륜 차량은 일반적으로 비선형 이동 모델을 사용하므로, 확장 칼만 필터와 입자 필터(몬테카를로 위치추정)가 종종 사용되기도 한다.

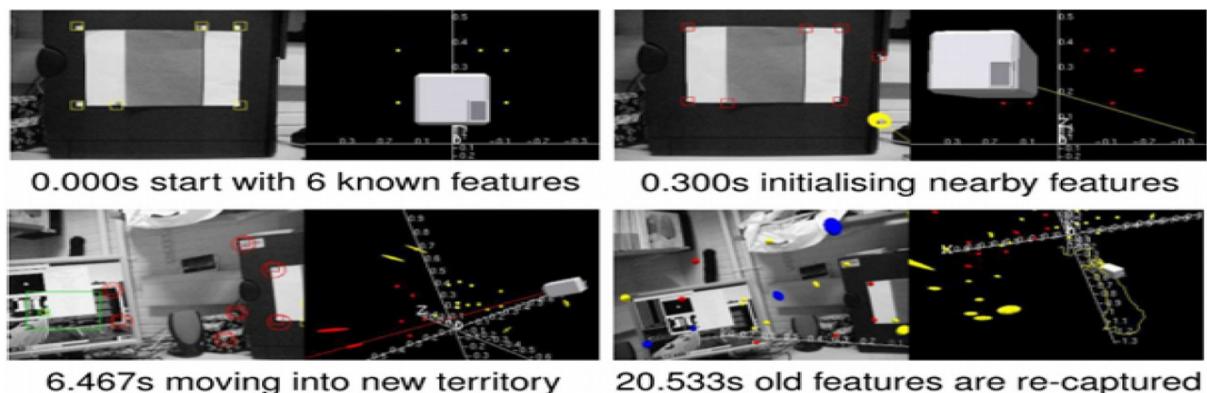


그림 84. Kalman Filter 기반의 SLAM 방법

위치추정에 실패할 경우 복원을 위한 대응책은 이전에 갔던 장소의 표지물을 키프레임으로서 기억하는 것이다. 표지물을 검색할 때는 고속 스캔이 가능한 방식으로 특징 추출 절차를 수행하는 방법과 영상 특징에 기반한 방법에는 BoF(Bag of Features) 및 BoVW(Bag of Visual Words)를 활용하기도 한다.

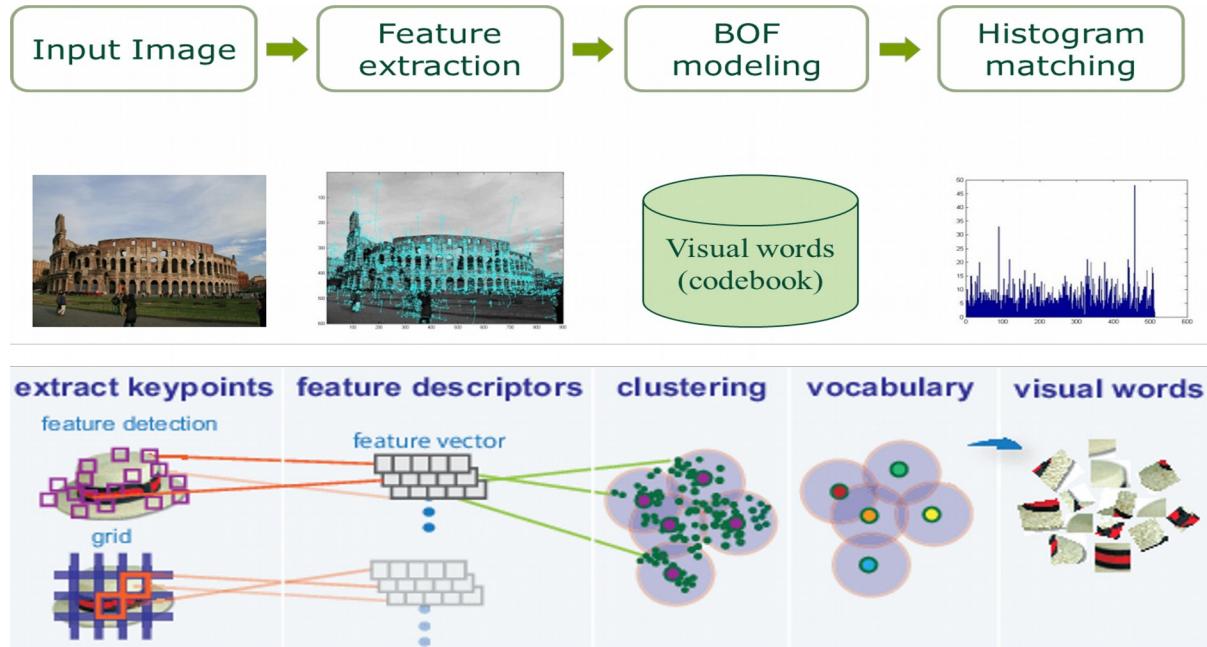


그림 85. Bag of Vision words 활용 예시

영상처리, 포인트 클라우드 처리 및 최적화에 소요되는 높은 계산비용

차량 하드웨어에 SLAM을 구현할 때는 계산 비용이 문제가 될 수 있다. SLAM을 위한 계산은 대부분 제한된 처리 능력을 갖는 소형 저전력 임베디드 마이크로프로세서에서 수행된다. 정확한 위치추정을 위해서는 높은 빈도로 포인트 클라우드 매칭과 영상 처리를 실행하는 것이 매우 중요하다. 게다가 루프 폐쇄와 같은 최적화 계산은 계산량이 높은 과정이지만, 문제는 그처럼 계산적 비용이 상당한 과정을 임베디드 마이크로컴퓨터에서 어떻게 실행하나는 데 있다.

이에 대한 대응책은 서로 다른 프로세스를 병렬로 실행하는 것이다. 매칭의 전처리 과정인 특징 추출 같은 과정은 병렬화에 비교적 적합하다. 멀티코어 CPU를 처리와 SIMD(Single Instruction Multiple Data) 계산, 임베디드 GPU에 이용하면 경우에 따라 속도를 더욱 향상할 수 있을 것이다. 또한 자세 그래프 최적화는 비교적 긴 주기에 걸쳐 수행될 수 있으므로 우선순위를 낮추고 일정한 간격으로 이 처리를 수행하는 것으로도 성능을 개선할 수 있다.

본 장에서는 다양한 SLAM 방법 중 Cartographer를 이용한 LiDAR SLAM & Navigation을 한번 실행해보고 이에 따른 지도 저장, 최적경로 추정을 통한 Navigation이 이루어지도록 해보도록 하자.

7.3.2. Turtlebot3를 이용한 SLAM 노드 실행 및 지도 저장

본 장에서 실습을 수행할 SLAM Algorithm은 Cartographer를 사용할 것이다. Cartographer는 Google이 2016년 오픈소스로 공개한 SLAM 알고리즘으로, 해당 알고리즘을 통해 google maps의 실내 지도를 작성하는데 사용되어왔다. Cartographer의 로컬 SLAM은 연속적으로 backend를 통해 위치를 주정하는 일을 수행한다. 각 서브맵은 지역적으로 일관되도록 되어 있지만 우리는 지역 SLAM이 시간이 지남에 따라 표류한다는 것을 인지하는 것이 핵심분야이다. 다른 서브시스템은 글로벌 SLAM이라고 하는데 백그라운드 스레드에서 실행되며, 루프 폐쇄 제약 조건을 찾는 것이 주 업무이다. 그것은 (노드에 입력된) matching 값을 비교한다. 또한 다른 센서 데이터를 통합하여 보다 높은 수준의 시야를 확보하고 가장 일관된 글로벌 솔루션을 파악한다는 장점이 있다.

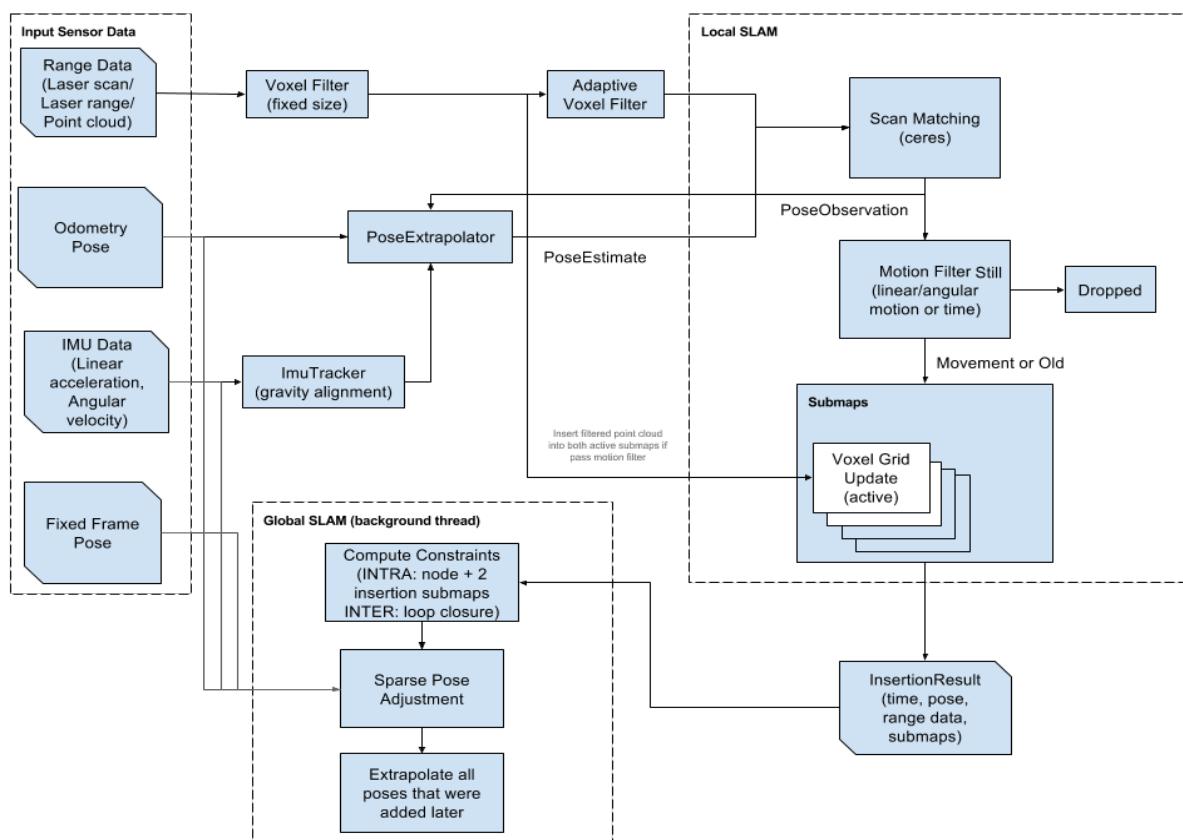


그림 86. Cartographer Workflow

SLAM 노드를 실행하기 위해 우선 turtlebot3의 robot launch 파일을 실행해야 한다. 라즈베리파이에 ssh로 접속하여 turtlebot3_bringup의 robot.launch.py 파일을 실행해주도록 하자.

```
$ ssh ubuntu@{IP_ADDRESS_OF_RASPBERRY_PI}
$ export TURTLEBOT3_MODEL=${TB3_MODEL}
$ ros2 launch turtlebot3_bringup robot.launch.py
```

Robot.launch.py 파일이 정상적으로 실행되어 RUN이라는 log 가 나오게 되었다면 이제 PC에서 SLAM 노드를 실행한다. 노드를 실행하게 되면 다음과 같은 RVIZ 창이 나오게 되고 지도를 통해 인식되는 공간 정보를 지속적으로 수집하고 있는 모습을 볼 수 있을 것이다.

```
$ export TURTLEBOT3_MODEL=${TB3_MODEL}  
$ ros2 launch turtlebot3_cartographer cartographer.launch.py
```

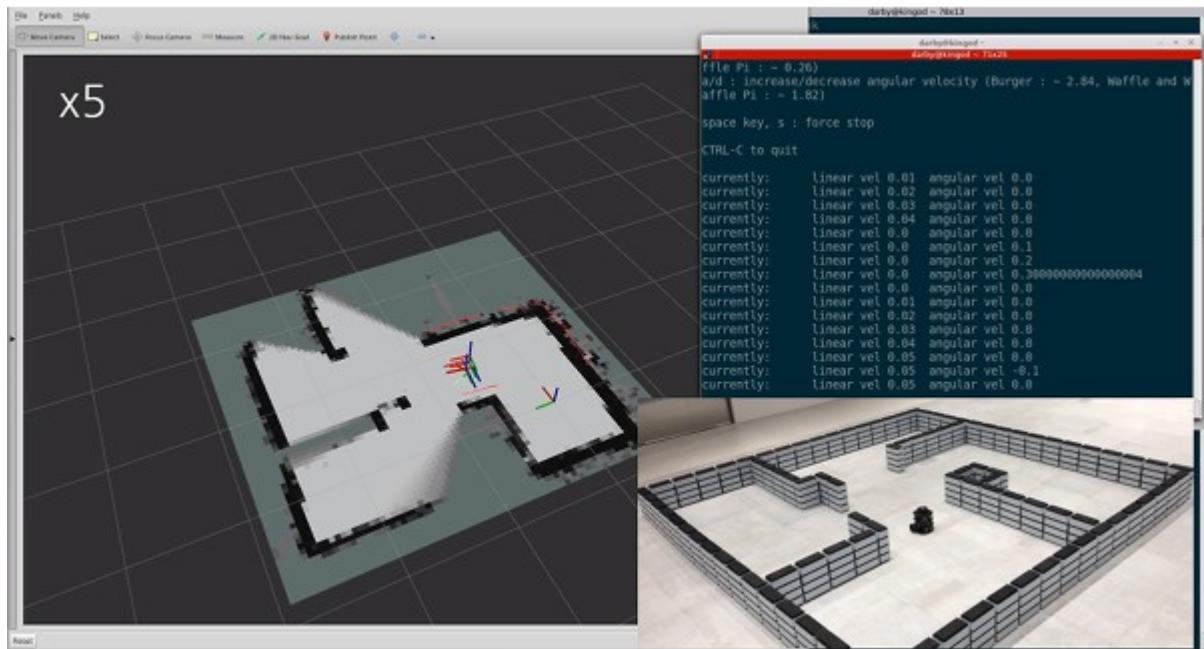


그림 87. Cartographer SLAM 노드 실행 화면

이제 새로운 Terminal 창을 열고 Teleoperation 노드를 활용하여 turtlebot3 를 이동시키며 지도를 작성하도록 한다. 이동 시 주의할 사항은 과도한 속도 이동 등은 자제한다.

```
$ export TURTLEBOT3_MODEL=burger  
$ ros2 run turtlebot3_teleop teleop_keyboard
```

Control Your TurtleBot3!

Moving around:

w
a s d
x

w/x : increase/decrease linear velocity
a/d : increase/decrease angular velocity
space key, s : force stop

CTRL-C to quit

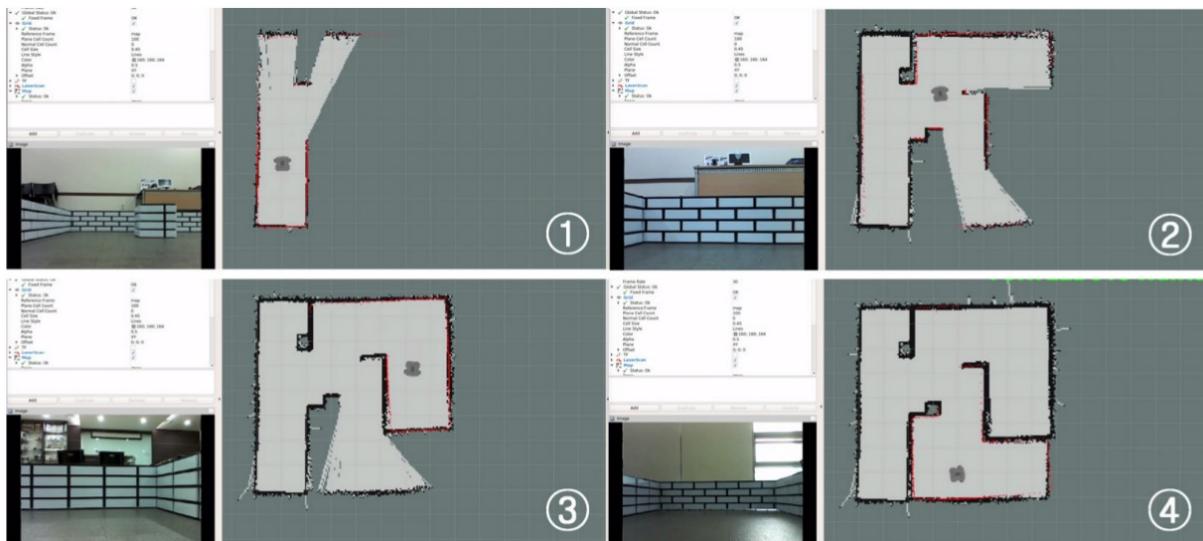


그림 88. Teleoperation을 통해 이동이 되는 과정

작성되어 지는 지도는 로봇의 Odom, LiDAR Scan, Tf 정보 등을 바탕으로 작성되게 된다. 원하는 영역에 대한 전체 지도 작성이 완료되었다면 다음과 같은 명령어를 통해 지도를 저장하도록 하자. 해당 저장 지도는 home 디렉토리에 map이라는 이름으로 저장이 될 것이다.

```
$ ros2 run nav2_map_server map_saver -f ~/map
```

작성된 지도는 pgm 파일과 yaml 파일로 구성되어 있으며 이 지도파일들은 ROS에서 흔히 사용되는 2 차원 점유 그리드 맵을 사용하게 된다. 아래 그림과 같이 검은색 지역이 장애물로 인식되어 있고 이동을 하면서 충돌없이 이동이 가능한 영역은 흰색으로, 이동할 수 있는지 없는지 알 수 없는 미지의 영역은 회색으로 표기된 것을 볼 수 있다.

7.3.3. Navigation 이란?

7.2.2 를 통해 우리는 주행로봇으로 이동체의 위치를 추정하기 위한 SLAM 알고리즘 중 cartographer 알고리즘에 대해 알아보았다. SLAM을 통해 주행로봇은 작업 환경에 능동적으로 대처할 수 있게 되었고 이를 바탕으로 자유롭게 이동할 수 있는 하나의 정보를 취득하게 된 것이다.

주행로봇은 어떤 임무가 주어졌을 때 자기 스스로 센서를 통해 주변의 환경과 자신의 위치나 상태를 파악해서 이동하고자하는 경로 계획을 해야 하며 이러한 경로는 주행 시간, 소모되는 에너지, 안정성에 대한 고려가 되어야 하고 계산된 경로를 따라가기 위한 경로 추종(path tracking)이 이루어져야 한다. 그리고 경로 계획은 크게 두 부분으로 나뉘어지는데 먼저 오프 라인(off-line)으로 이미 알려진 환경의 지도나 모델을 이용 하는 광역 경로 계획(global path planning)과 실시간으로 바뀌어가는 환경에 적응하기 위해 온라인(on-line)으로 상황을 판단하는 국지적 경로 계획(local path planning)이 있다.

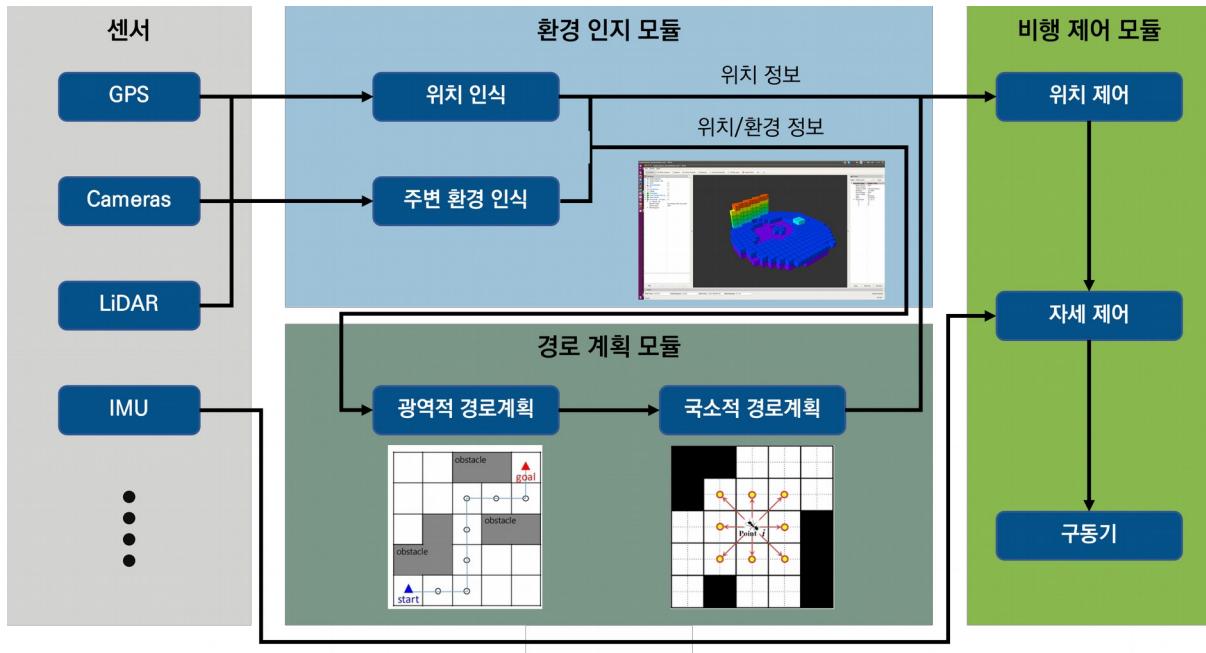


그림 88. 무인비행체의 환경인지 및 경로계획

광역적 경로 계획에서는 장애물과의 충돌이 발생하지 않기 위해 기하학적 조건만 만족하는 경로를 계획한다. 광역적 경로 계획의 결과물은 수 개의 경로점으로 이루어진 경로점들의 집합이며, 경로 점을 생성하는 과정에서 무인 비행체의 동역학은 고려되지 않는다. 광역적 경로계획은 크게 탐색 기반 알고리즘과 샘플링 기반 알고리즘을 통해 계획을 수립하게 된다.

탐색 기반 알고리즘

일반적으로 탐색 기반 알고리즘은 지도상의 장애물 정보를 그래프(graph)로 변환하여 탐색에 사용한다. 그래프는 꼭짓점(vertex)과 꼭짓점과 꼭짓점 사이를 잇는 엣지로 구성된다. 탐색 기반 알고리즘 중 가장 보편적으로 사용되는 A* 알고리즘은 출발 꼭짓점에서 목표 꼭짓점까지 최적 경로를 탐색하는 그래프 탐색 알고리즘이다. 최적 경로를 결정하기 위해서는 각 꼭짓점에 대한 비용 함수를 정의해야 하는데, A* 알고리즘에서 사용되는 비용 함수는 출발 꼭짓점에서 현재 꼭짓점까지의 경로 가중치와 현재 꼭짓점에서 목표 꼭짓점까지의 추정 경로 가중치의 합으로 표현된다. 일반적으로 탐색 기반 알고리즘은 고차원 공간에서는 비효율적이라고 여겨지기도 하지만, 추정 경로 가중치를 설계하는 방식에 따라 탐색 속도를 가속화시킬 수 있고 해의 최적성을 향상시킬 수 있다는 점에서 A* 알고리즘을 변형한 여러 형태의 알고리즘이 제안되어왔다. 그중에서도 LPA* 알고리즘에서는 다음 경로점 후보들이 가질 수 있는 목표 꼭짓점까지의 경로 가중치를 모두 조사하여 그중 가장 최소값을 추정 경로 가중치로 사용하였다. 즉, 기존의 A* 알고리즘보다 한 단계 앞을 예측할 수 있는 추정 경로 가중치를 사용함으로써 알고리즘이 최적 해로 수렴하지 못하고 극소점(Local minima)에 빠 질 가능성을 줄이고자 했다.

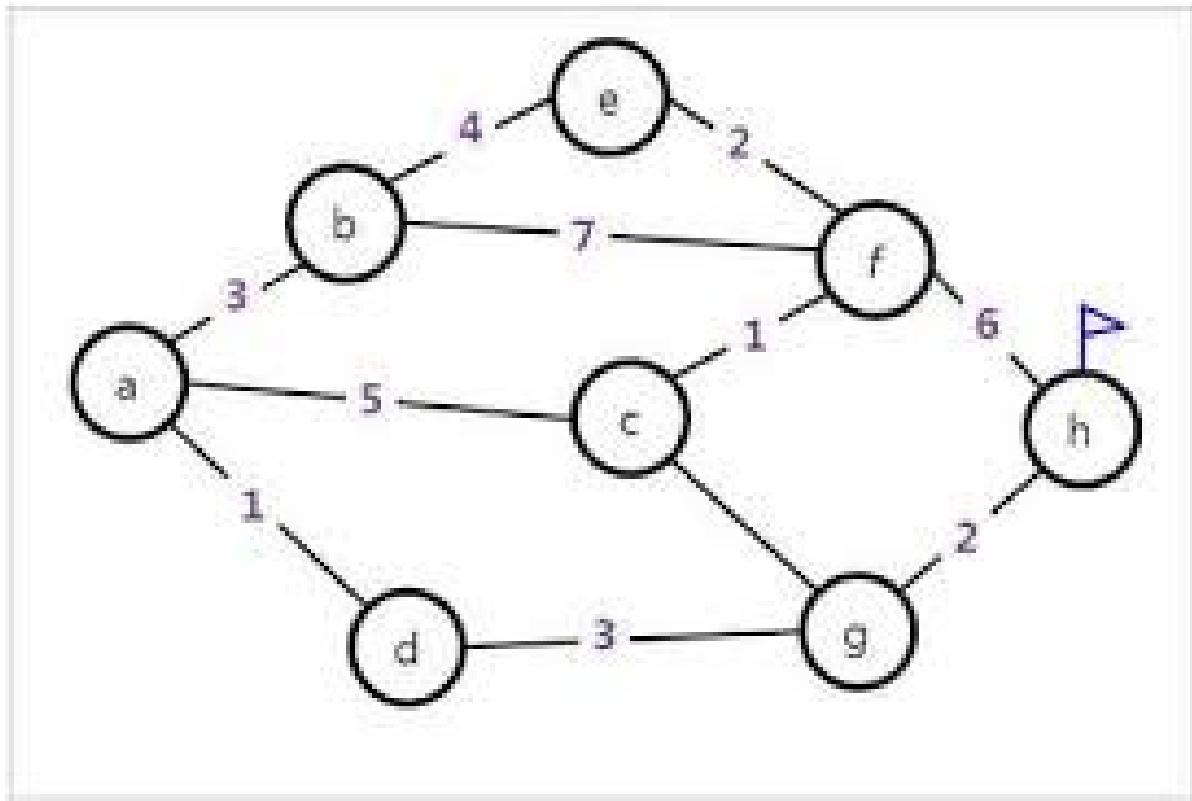


그림 89. LPA 기반 경로계획

샘플링 기반 알고리즘

샘플링 기반 알고리즘은 복잡한 고차원 공간에서도 빠른 시간 안에 적절한 경로를 효율적으로 찾아내기 위해 제안된 방법이다. 여러 샘플링 기반 알고리즘 중 가장 보편적으로 사용되는 RRT 알고리즘은 경로 계획을 위해 시작점에서부터 목표점까지 점진적으로 트리(Tree)를 구축하는 방식을 사용한다. 이를 위해 우선적으로 장애물과의 충돌이 없는 공간에서 무작위 샘플(Sample)을 추출하고, 샘플이 추출되면 그 샘플과 현재까지 구축된 트리에서 가장 가까운 노드 사이에 연결이 시도된다. 연결은 동역학적 구속 조건과 같은 제한조건을 만족하면서도 장애물과의 충돌이 없을 때만 가능하며, 연결이 가능한 경우에만 해당 샘플은 트리의 새 노드로 추가된다. 이러한 과정을 반복해 나가면서 시작점에서 목표점까지 하나의 연결된 궤적이 만들어질 때까지 트리를 아직 탐색되지 않는 공간으로 확장시키는 것이 RRT 알고리즘의 기본 개념이다.

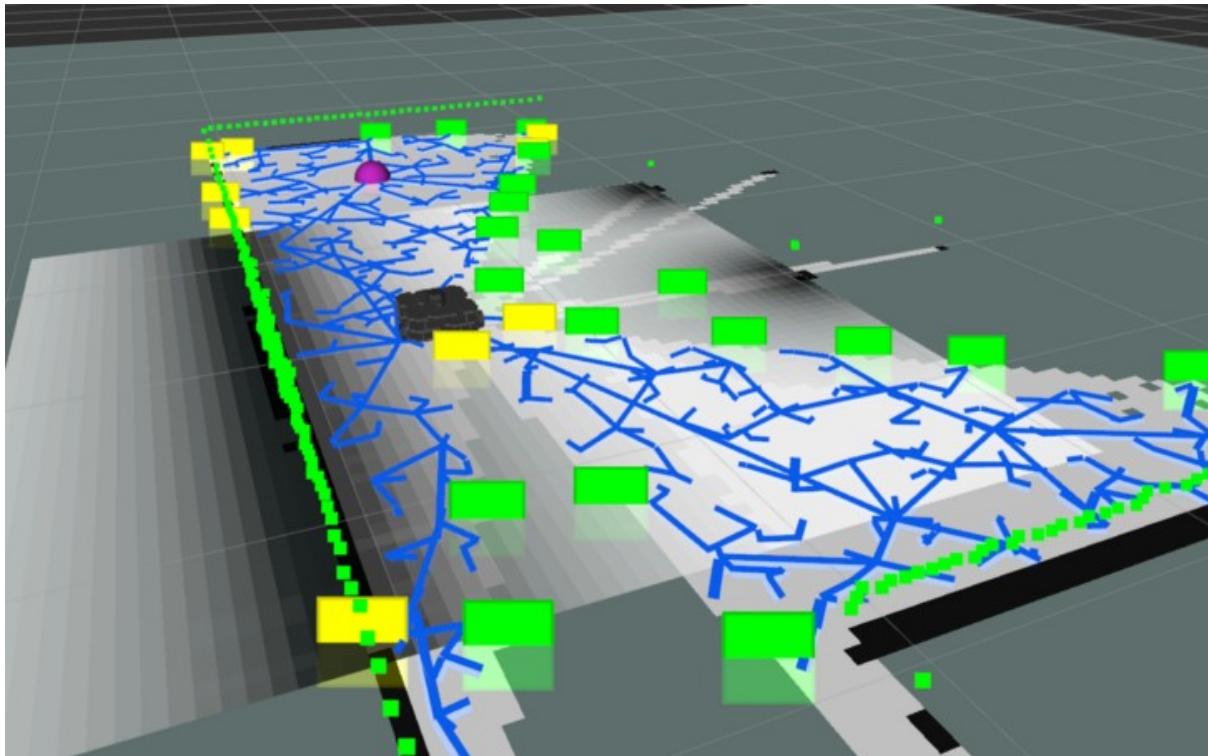


그림 89. Randomly Exploring Random Tree(RRT) 알고리즘을 사용하는 Navigation

국소적 경로 계획에서는 상위 단계에서 결정된 임의의 인접한 두 경로점 사이에 동적으로 실현 가능한 부드러운 궤적(Trajectory)을 생성한다. 즉 상위 단계에서는 경로점을 추종할 수 있는 주행로봇의 능력을 고려하지 않았지만, 하위 단계에서는 주행로봇의 동특성을 반영하여 무인 비행체가 보다 쉽게 추종 가능한 궤적을 생성함으로써 주 어진 궤적을 정확하게 추종하기 위한 제어 입력 생성을 용이하게 한다.

7.3.4. Turtlebot3 를 활용한 Navigation 노드 실행 및 경로 이동

위의 다양한 Navigation 알고리즘을 ROS 에서는 기본적으로 제공해주고 있다. 7.2.2.에서 작성한 지도파일을 통해 Navigation 을 수행해보도록 하자. 가장 먼저 turtlebot3 를 제어하기 위한 robot.launch.py 을 실행해주도록 하자.

```
$ ssh ubuntu@{IP_ADDRESS_OF_RASPBERRY_PI}
$ export TURTLEBOT3_MODEL=${TB3_MODEL}
$ ros2 launch turtlebot3_bringup robot.launch.py
```

Robot.launch.py 파일 실행을 통해 Run 이라는 log 가 나오게되었다면, 새로운 터미널 창을 열어 navigation 노드를 실행한다. Navigation 실행을 위해서는 map.yaml 파일을 인자정보로 전달해주어야 한다. 앞장에서 우리는 home 폴더에 저장을 하였기 때문에 home/map.yaml 파일을 경로로 불러올 것이다.

```
$ export TURTLEBOT3_MODEL=burger
$ ros2 launch turtlebot3_navigation2 navigation2.launch.py map:=$HOME/map.yaml
```

정상적으로 노드를 실행하였다면 RVIZ 가 실행되면서 현재 불러온 즈도의 위치와 turtlebot3 의 URDF 를 확인할 수 있을 것이다. 가장 먼저 현재 turtlebot3 의 위치를 초기화해주기 위해 상단의 2D Pose Estimate 버튼을 눌러 대략적인 로봇의 위치와 방향을 설정해주도록 한다. 설정을 해주고 나면 현재 라이더로 인식한 로봇의 장애물값과 실제 지도상의 장애물이 일치 될 것이다.

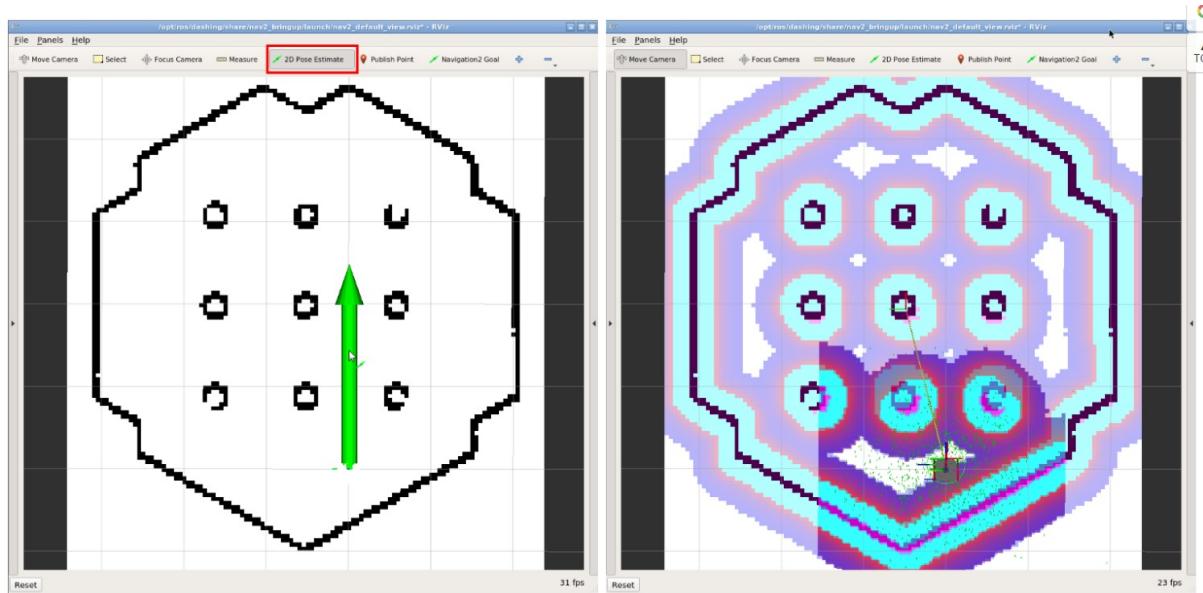


그림 90. 2D Pose Estimate 를 통한 초기 위치 보정

이제 표기되어있는 지도를 한번 보도록 하자. 지도에는 다양한 정보가 확인이 가능한데, 가장 눈에 띄는 것은 무수히 많은 초록색 화살표일 것이다. 해당 초록색 경로들이 바로 국소적 경로계획을 통해 turtlebot 이 이동할 수 있는 무수한 이동경로들을 표기해주는 것이다. 이러한 초록색의 확률값들을 사용하여 주행로봇이 어떻게 이동하면 좋을지 최선의 경로를 찾아 이동하게 되는 것이다.

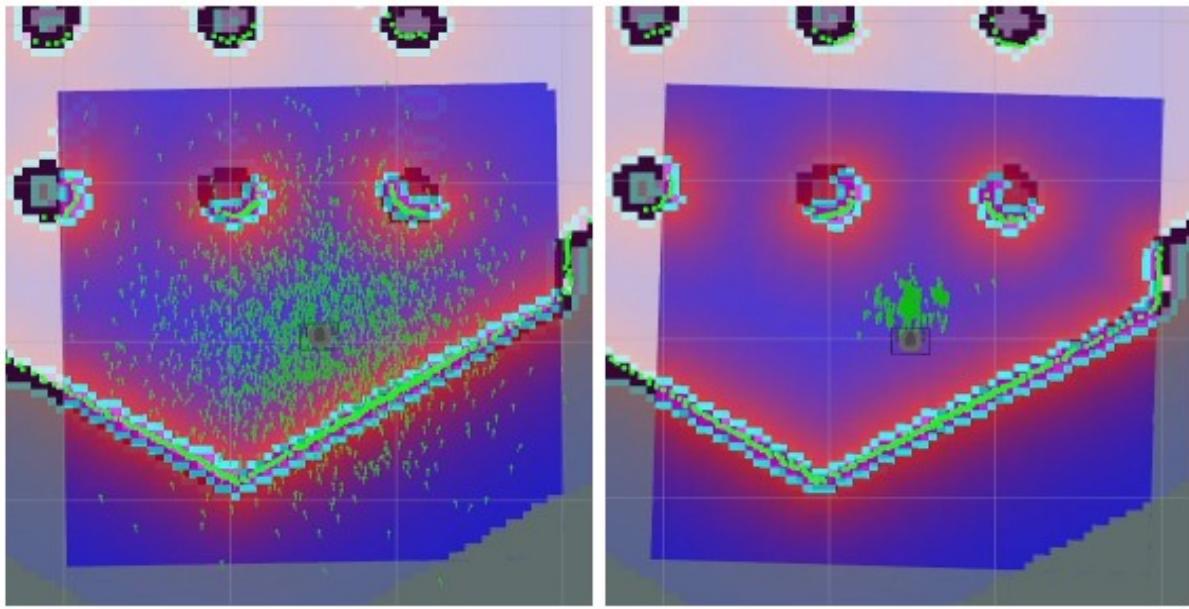


그림 91. 국소적 경로계획을 통해 표시되는 turtlebot 의 예상 이동경로

이제 광역적 경로계획을 수립하고 자동으로 이동해보는 것을 실습해보도록 하자. RVIZ의 상단 메뉴에 Navigation2 Goal이라는 버튼을 눌러주고 원하는 목적지와 방향을 설정해주도록 하자. 설정이 끝나게 되면 검정색으로 실제 목표까지의 광역적 경로가 수립이 되고 무수한 초록색 화살표 중 최선의 경로를 탐색하여 turtlebot이 이동하는 모습을 확인할 수 있다.

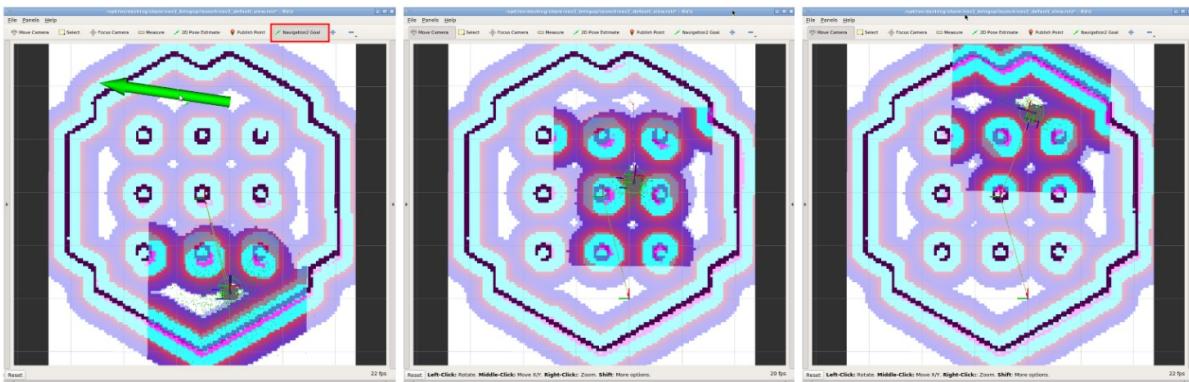


그림 91. Navigation2를 통한 광역적 경로계획 및 경로 추종을 통한 turtlebot 의 이동

7.4. 가상환경에서의 Turtlebot3 제어

7.4.1. Gazebo 를 활용한 Turtlebot3 시뮬레이션하기

기존의 Robot 프로그래밍 및 자율주행 프로젝트에서 가장 어려운 것은 실제 환경과 시뮬레이션 상의 고리감 뿐만 아니라 프로그램의 디버깅을 위해 각각 시뮬레이터를 만들어야 하는 문제점이 있었다. 이를 해결하기 위해 ROS는 Gazebo라는 시뮬레이션 tool을 활용하여 실제 환경과 유사한 상황에서 노드를 실행하고 검증이 이루어질 수 있도록 지원하고 있다.

Gazebo는 실제 환경과 유사한 기능을 갖춘 3D 시뮬레이터로 복잡한 실내외 환경에서의 로봇의 제어를 테스트하기 위해 만들어졌다. Gazebo는 게임엔진과 유사하지만, 실제와 유사한 물리법칙을 제공하여 로봇 알고리즘 테스트, 로봇 디자인, 회귀테스트를 수행할 수 있다는 장점이 있다. 특히 Gazebo는 ROS와 독립적 또는 종속적으로 개발할 수 있고, 다양한 인터페이스를 제공하고 있다.

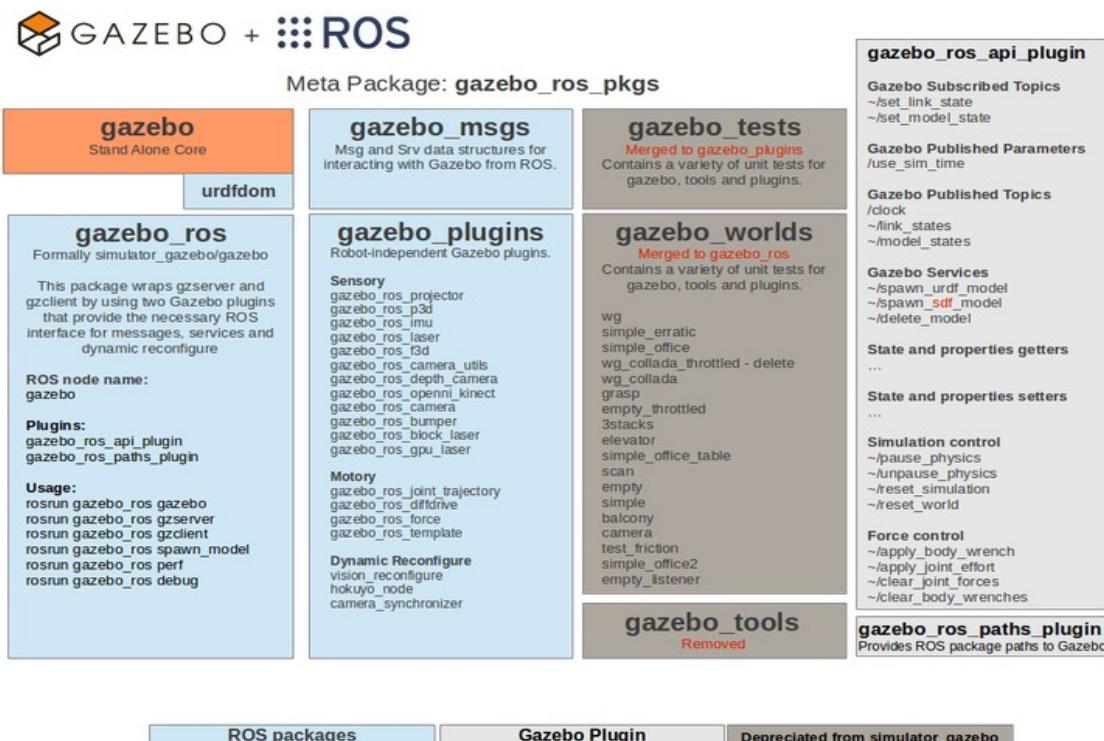


그림 92. Gazebo + ROS Interface

Turtlebot3의 경우에도 Gazebo를 활용한 Simulation을 제공하고 있으며, 해당 패키지 설치 및 실행을 통해 예제 code를 실행해보도록 하자. 7.1에서 설치했던 turtlebot3_ws/src 폴더로 이동하여 turtlebot3_simulations package를 다운받고 빌드를 실행한다.

```
$ cd ~/turtlebot3_ws/src/  
$ git clone -b dashing-devel https://github.com/ROBOTIS-GIT/turtlebot3_simulations.git  
$ cd ~/turtlebot3_ws && colcon build --symlink-install  
빌드가 완료되었다면 gazebo 모델 정보를 bashrc에 추가해주도록 한다.
```

```
$ echo 'export  
GAZEBO_MODEL_PATH=$GAZEBO_MODEL_PATH:~/turtlebot3_ws/src/turtlebot3/turtlebo  
t3_simulations/turtlebot3_gazebo/models' >> ~/.bashrc  
$ source ~/.bashrc
```

이제 시뮬레이션을 구동해보도록 하자. 기본 Simulation package에는 크게 3 가지 시뮬레이션을 구동할 수 있다. 가장 먼저 아무것도 없는 empty world 패키지를 실행해보도록 하자.

```
$ export TURTLEBOT3_MODEL=burger  
$ ros2 launch turtlebot3_gazebo empty_world.launch.py
```

최초 구동 시 초기설정을 위해 많은 시간이 소요되며 정상적으로 gazebo가 실행되면 turtlebot3 urdf가 표시되는 것을 알 수 있다.



그림 93. Empty_world 시뮬레이션 구동 화면

이제 다양한 시뮬레이션 환경에서 SLAM & Navigation 등을 테스트해볼 수 있도록 다양한 구조물이 배치되어 있는 world, house 패키지를 실행해보도록 하자.

```
$ export TURTLEBOT3_MODEL=waffle  
$ ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

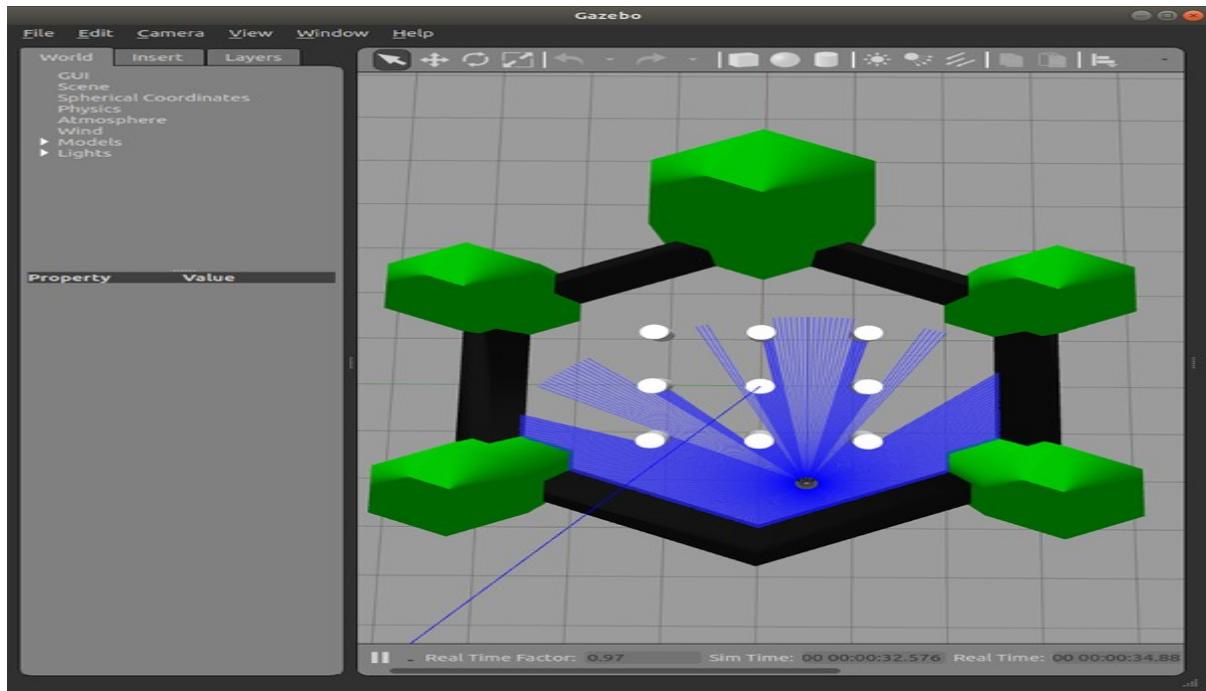


그림 94. Turtlebot3 world 실행 화면

```
$ export TURTLEBOT3_MODEL=waffle_pi  
$ ros2 launch turtlebot3_gazebo turtlebot3_house.launch.py
```

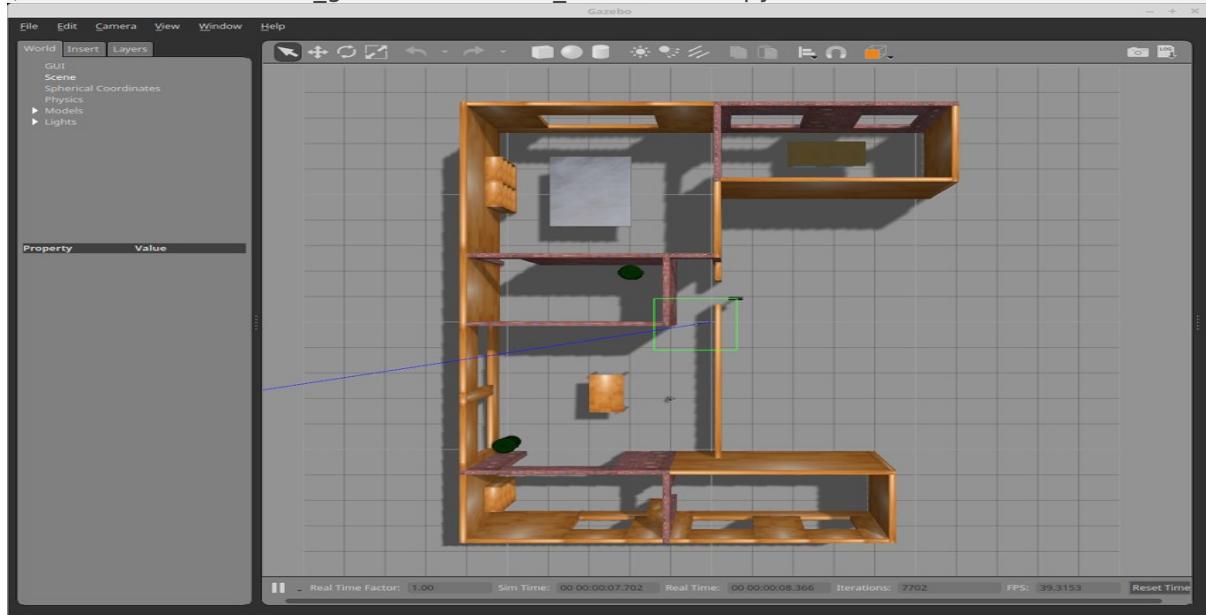


그림 95. Turtlebot3 house 실행화면

7.4.2. Gazebo 를 활용한 Turtlebot3 SLAM/Navigation Simulation 하기

이제 실제 환경이 아닌 가상환경 상에서 Turtlebot3 의 SLAM 을 실행해보고 지도 작성이 이루어질 수 있도록 실습해볼 것이다. 우선 Gazebo 가상환경을 실행하도록 하자.

```
$ export TURTLEBOT3_MODEL=burger  
$ ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py
```

이제 새로운 터미널을 열어 SLAM 을 실행해보도록 하자. SLAM Node 는 7.2 에서 사용하였던 Cartographer 를 사용할 것이다.

```
$ export TURTLEBOT3_MODEL=burger  
$ ros2 launch turtlebot3_cartographer cartographer.launch.py use_sim_time:=True
```

이제 Teleoperation Node 를 실행하여 Turtlebot3 를 이동하며 지도를 작성해보도록 하자.

```
$ export TURTLEBOT3_MODEL=burger  
$ ros2 run turtlebot3_teleop teleop_keyboard
```

Control Your TurtleBot3!

Moving around:

```
w  
a s d  
x
```

w/x : increase/decrease linear velocity
a/d : increase/decrease angular velocity
space key, s : force stop

CTRL-C to quit

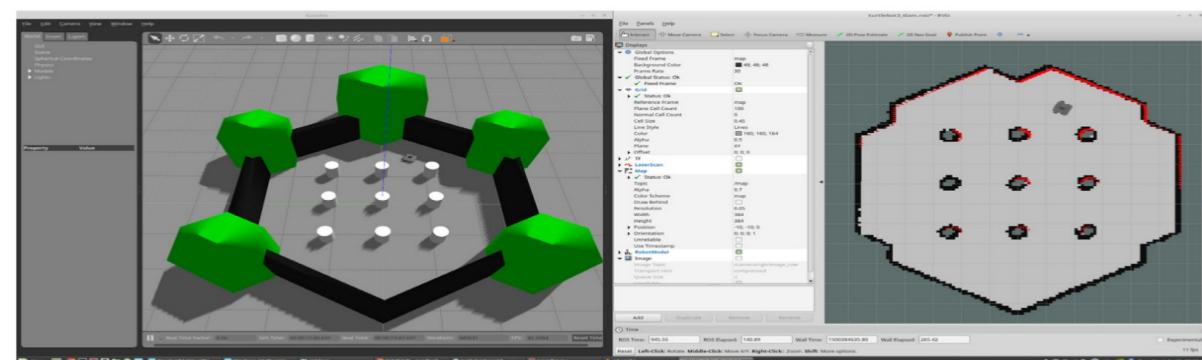


그림 96. Gazebo 환경에서 SLAM 이 이루어지고 있는 화면

지도 작성이 완료되었다면 map_saver 노드를 실행하여 저장해보도록 하자.

```
$ ros2 run nav2_map_server map_saver -f ~/map
```

저장이 완료되었다면 이제 Navigation 을 실행해 볼 차례이다. Navigation Node 를 실행하여 RVIZ 를 실행해보도록 하자.

```
$ export TURTLEBOT3_MODEL=burger  
$ ros2 launch turtlebot3_navigation2 navigation2.launch.py use_sim_time:=True  
map:=$HOME/map.yaml
```

정상적으로 노드를 실행하였다면 RVIZ 가 실행되면서 현재 불러온 즈도의 위치와 turtlebot3 의 URDF 를 확인할 수 있을 것이다. 가장 먼저 현재 turtlebot3 의 위치를 초기화해주기 위해 상단의 2D Pose Estimate 버튼을 눌러 대략적인 로봇의 위치와 방향을 설정해주도록 한다. 설정을 해주고 나면 현재 라이더로 인식한 로봇의 장애물값과 실제 지도상의 장애물이 일치 될 것이다.

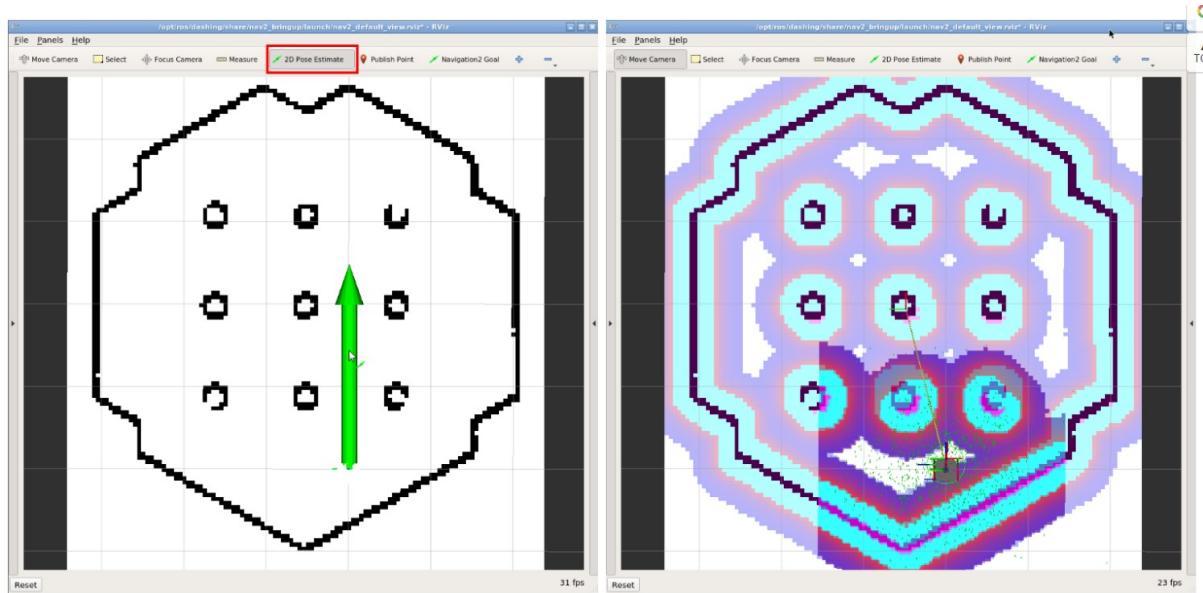


그림 97. 2D Pose Estimate 를 통한 초기 위치 보정

이제 표기되어있는 지도를 한번 보도록 하자. 지도에는 다양한 정보가 확인이 가능한데, 가장 눈에 띄는 것은 무수히 많은 초록색 화살표일 것이다. 해당 초록색 경로들이 바로 국소적 경로계획을 통해 turtlebot 이 이동할 수 있는 무수한 이동경로들을 표기해주는 것이다. 이러한 초록색의 확률값들을 사용하여 주행로봇이 어떻게 이동하면 좋을지 최선의 경로를 찾아 이동하게 되는 것이다.

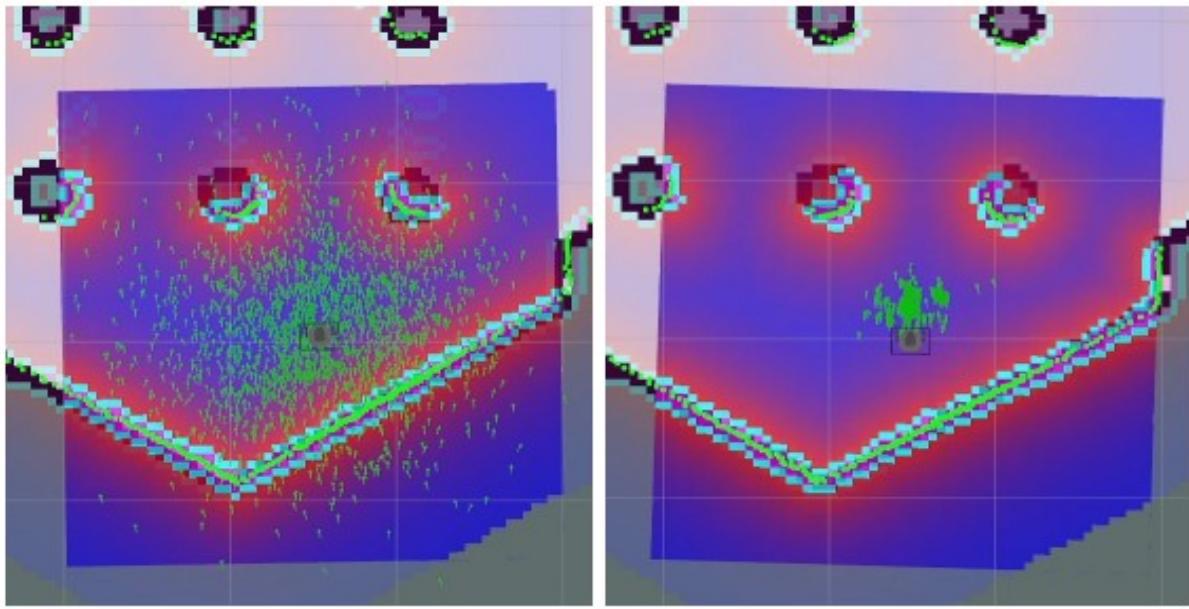


그림 98. 국소적 경로계획을 통해 표시되는 turtlebot 의 예상 이동경로

이제 광역적 경로계획을 수립하고 자동으로 이동해보는 것을 실습해보도록 하자. RVIZ 의 상단 메뉴에 Navigation2 Goal 이라는 버튼을 눌러주고 원하는 목적지와 방향을 설정해주도록 하자. 설정이 끝나게 되면 검정색으로 실제 목표까지의 광역적 경로가 수립이 되고 무수한 초록색 화살표 중 최선의 경로를 탐색하여 turtlebot 이 이동하는 모습을 확인할 수 있다.

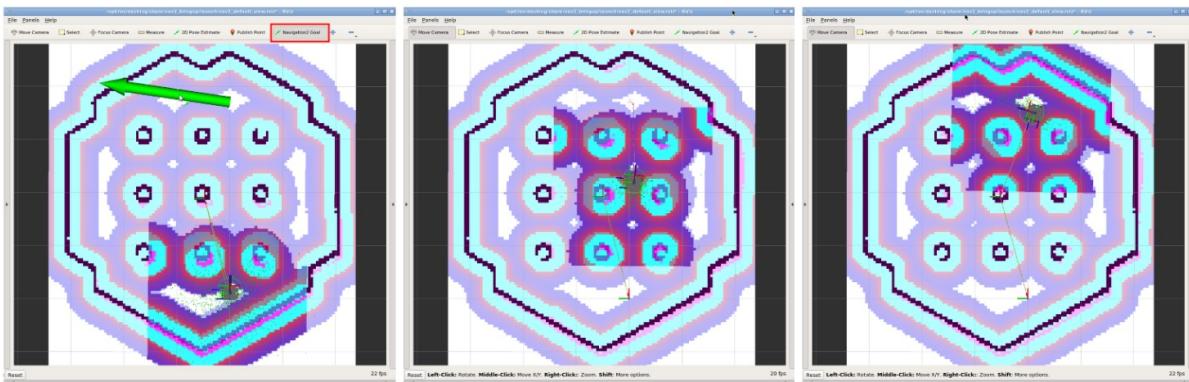


그림 99. Navigation2 를 통한 광역적 경로계획 및 경로 추종을 통한 turtlebot 의 이동

8. ROS2 기반 주행로봇 프로그래밍

7장에서 우리는 주행로봇 중 하나인 turtlebot3를 직접 환경설정하고 이동하는 프로그램들을 사용해보았다. 이제 8장에서는 직접 turtlebot3만을 위한 source code를 만들어보고 제어를 수행함으로써 실제 topic 값의 제어와 이동이 이루어질 수 있도록 해보자.

8.1. Teleoperation 구현하기

8.1.1. C++을 이용한 Turtlebot3 Teleoperation 구동하기

패키지 생성하기

우리는 6장에서 ROS 기본 프로그래밍 실습을 colcon_ws에서 작업하였다. 본 Node도 colcon_ws에서 작업을 진행해보도록 하자. 가장먼저 colcon_ws로 이동하여 teleoperation node 패키지를 생성해보도록 하자. Teleoperation은 사용자로부터 key 값을 입력받아 turtlebot3의 cmd_vel topic에 특정한 값을 보내주도록 설계되어 있다.

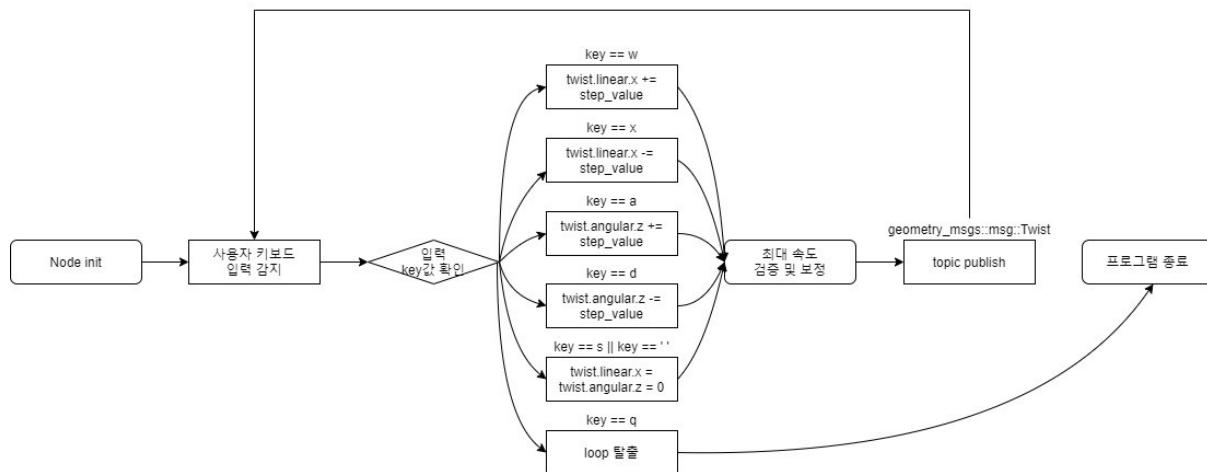


그림 100. Teleropation 동작 Process

Teleoperation을 위해 필요한 의존성 정보는 rclcpp, std_msgs, geometry_msgs가 필요한 의존성 정보이다.

```
$ cd ~/colcon_ws/src
$ ros2 pkg create turtlebot3_teleoperation_cpp -build-type ament_cmake -node-name
turtlebot3_teleoperation_node -dependencies rclcpp std_msgs geometry_msgs
```

패키지를 생성하였다면 ~/colcon_ws/src에 turtlebot3_teleoperation_cpp 패키지 폴더와 ROS 패키지가 갖추어야 할 내부 폴더 그리고 package.xml 파일들이 생성된다.

```
.  
|   include  
|   |   └── turtlebot3_teleoperation_cpp  
|   src  
|   |   └── turtlebot3_teleoperation_node.cpp  
|   CMakeLists.txt  
|   package.xml
```

3 directories, 2 files

이제 각각 class 선언 및 프로그래밍을 진행해보도록 하자.

Teleoperation.h

```
#ifndef TURTLEBOT3_EXAMPLE_TELEOPERATION_H  
#define TURTLEBOT3_EXAMPLE_TELEOPERATION_H  
  
#include <rclcpp/rclcpp.hpp>  
#include <std_msgs/msg/string.hpp>  
#include <geometry_msgs/msg/twist.hpp>  
  
#include <chrono>  
#include <functional>  
#include <memory>  
#include <string>  
#include <cmath>  
  
#include <stdio.h>  
#include <termios.h>  
#include <unistd.h>  
#include <fcntl.h>  
  
#include <iostream>  
#include <stdio.h>  
  
#define MAX_TURTLEBOT_LINEAR_SPEED 0.22  
#define MAX_TURTLEBOT_ANGULAR_SPEED 2.84  
  
using namespace std::chrono_literals;  
  
class Teleoperation : public rclcpp::Node  
{  
private:  
    rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr cmd_pub;  
    geometry_msgs::msg::Twist twist;  
  
    double target_twist_linear_x;  
    double target_twist_angular_z;  
  
    double step_linear_speed;  
    double step_angular_speed;  
  
    std::string command;  
    int command_cnt;  
  
public:  
    Teleoperation();  
    void teleoperation_node();  
    geometry_msgs::msg::Twist calculate_max_speed(geometry_msgs::msg::Twist twist);  
    void publishing_data(geometry_msgs::msg::Twist twist);
```

```

int getch();
int kbhit();

void stop_turtlebot();
static void interrupt_handler(int signal);
};

static Teleoperation* teleop;

#endif //TURTLEBOT3_EXAMPLE_TELEOPERATION_H

```

Teleoperation.cpp

```

#include "Teleoperation.h"

Teleoperation::Teleoperation() : Node("turtlebot3_teleoperation_cpp")
{
    this->cmd_pub = create_publisher<geometry_msgs::msg::Twist>("/cmd_vel", 10);

    this->target_twist_linear_x = 0;
    this->target_twist_angular_z = 0;

    this->step_linear_speed = 0.01;
    this->step_angular_speed = 0.1;
    this->command_cnt = 0;

    this->command = "Let's Control Turtlebot3!\n-----moving around:\n\tw\\na\\ts\\td\\n\\tx\\nw/x : increase/decrease linear velocity (Burger : ~ 0.22)\\na/d :\nincrease/decrease angular velocity (Burger : ~ 2.84)\\n\\nspace key, s : force stop\\n\\nq to quit\\n";
}

teleop = this;
signal(SIGINT, Teleoperation::interrupt_handler);
}

geometry_msgs::msg::Twist Teleoperation::calculate_max_speed(geometry_msgs::msg::Twist twist)
{
    if(abs(twist.linear.x) > MAX_TURTLEBOT_LINEAR_SPEED)
    {
        if(twist.linear.x < 0)
            twist.linear.x = MAX_TURTLEBOT_LINEAR_SPEED * -1;
        else
            twist.linear.x = MAX_TURTLEBOT_LINEAR_SPEED;
    }
    if(abs(twist.angular.z) > MAX_TURTLEBOT_ANGULAR_SPEED)
    {
        if(twist.angular.z < 0)
            twist.angular.z = MAX_TURTLEBOT_ANGULAR_SPEED * -1;
        else
            twist.angular.z = MAX_TURTLEBOT_ANGULAR_SPEED;
    }
    return twist;
}

void Teleoperation::publishing_data(geometry_msgs::msg::Twist twist)
{
    this->cmd_pub->publish(twist);
}

void Teleoperation::teleoperation_node()

```

```

{
    std::cout << command << std::endl;
    while(rclcpp::ok())
    {
        if(kbhit())
        {
            int key = getch();
            if(key == 'w')
                this->target_twist_linear_x += step_linear_speed;
            else if(key == 'x')
                this->target_twist_linear_x -= step_linear_speed;
            else if(key == 'a')
                this->target_twist_angular_z += step_angular_speed;
            else if(key == 'd')
                this->target_twist_angular_z -= step_angular_speed;
            else if(key == 's' || key == ' ')
            {
                this->target_twist_linear_x = 0;
                this->target_twist_angular_z = 0;
            }
            else if(key == 'q')
            {
                this->twist.linear.x = 0;
                this->twist.angular.z = 0;
                publishing_data(this->twist);
                RCLCPP_ERROR(this->get_logger(), "Stop Program");
                break;
            }
        }

        this->twist.linear.x = this->target_twist_linear_x;
        this->twist.angular.z = this->target_twist_angular_z;

        this->twist = calculate_max_speed(this->twist);

        std::cout << "currently : \t linear velocity : " << this->twist.linear.x << "\t angular
velocity : " << this->twist.angular.z << std::endl;

        this->command_cnt++;
        if(this->command_cnt % 20 == 0)
        {
            std::system("clear");
            std::cout << command << std::endl;
        }
        publishing_data(this->twist);
    }
}

int Teleoperation::getch()
{
    char buf = 0;
    struct termios old = {0};
    fflush(stdout);
    if(tcgetattr(0, &old) < 0)
        perror("tcgetattr()");
    old.c_lflag &= ~ICANON;
    old.c_lflag &= ~ECHO;
    old.c_cc[VMIN] = 1;
    old.c_cc[VTIME] = 0;
    if(tcsetattr(0, TCSANOW, &old) < 0)
        perror("tcsetattr ICANON");
}

```

```

if(read(0, &buf, 1) < 0)
    perror("read()");
old.c_lflag |= ICANON;
old.c_lflag |= ECHO;
if(tcsetattr(0, TCSADRAIN, &old) < 0)
    perror("tcsetattr ~ICANON");
return buf;
}

int Teleoperation::kbhit()
{
    struct termios oldt, newt;
    int ch;
    int oldf;

    tcgetattr(STDIN_FILENO, &oldt);
    newt = oldt;
    newt.c_lflag &= ~(ICANON | ECHO);
    tcsetattr(STDIN_FILENO, TCSANOW, &newt);
    oldf = fcntl(STDIN_FILENO, F_GETFL, 0);
    fcntl(STDIN_FILENO, F_SETFL, oldf | O_NONBLOCK);

    ch = getchar();

    tcsetattr(STDIN_FILENO, TCSANOW, &oldt);
    fcntl(STDIN_FILENO, F_SETFL, oldf);

    if(ch != EOF)
    {
        ungetc(ch, stdin);
        return 1;
    }
}

return 0;
}

void Teleoperation::stop_turtlebot()
{
    this->twist.linear.x = 0;
    this->twist.angular.z = 0;
    publishing_data(this->twist);
    RCLCPP_ERROR(this->get_logger(), "Keyboard Interrupt, Stop Program");
    exit(0);
}

void Teleoperation::interrupt_handler(int signal)
{
    if(signal == 2)
        teleop->stop_turtlebot();
}

```

turtlebot3_teleoperation_node.cpp

```

#include "Teleoperation.h"

int main(int argc, char** argv)
{
    rclcpp::init(argc, argv);
    auto teleoperation_node = std::make_shared<Teleoperation>();
    teleoperation_node->teleoperation_node();
    rclcpp::shutdown();
    return 0;
}

```

전체를 작성하였다면 이제 CMakeLists.txt 파일을 작성해주도록 하자.

```
cmake_minimum_required(VERSION 3.5)
project(turtlebot3_tutorial_ros2)

# Default to C99
if(NOT CMAKE_C_STANDARD)
  set(CMAKE_C_STANDARD 99)
endif()

# Default to C++14
if(NOT CMAKE_CXX_STANDARD)
  set(CMAKE_CXX_STANDARD 14)
endif()

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()

# find dependencies
find_packageament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)
find_package(geometry_msgs REQUIRED)

if(BUILD_TESTING)
  find_packageament_lint_auto REQUIRED)
  # the following line skips the linter which checks for copyrights
  # uncomment the line when a copyright and license is not present in all source files
  #set(ament_cmake_copyright_FOUND TRUE)
  # the following line skips cpplint (only works in a git repo)
  # uncomment the line when this package is not in a git repo
  #set(ament_cmake_cpplint_FOUND TRUE)
  ament_lint_auto_find_test_dependencies()
endif()

add_executable(turtlebot3_teleoperation_node src/ turtlebot3_teleoperation_node.cpp src/
/Teleoperation.h src /Teleoperation.cpp)
ament_target_dependencies(teleoperation_node rclcpp std_msgs geometry_msgs)

install(TARGETS
  teleoperation_node
DESTINATION lib/${PROJECT_NAME})

ament_package()
```

이제 소스코드를 보도록 하자. 먼저 노드를 `rclcpp::init`을 활용하여 초기화 한 후 무한루프를 돌며 `kbhit`을 이용하여 사용자가 키보드를 입력했는지 확인한다. 사용자의 키가 입력되었다면 `getch()`을 이용하여 사용자의 key 값을 받게 되는 것이다.

```
int Teleoperation::kbhit()
{
  struct termios oldt, newt;
  int ch;
  int oldf;

  tcgetattr(STDIN_FILENO, &oldt);
  newt = oldt;
  newt.c_iflag &= ~(ICANON | ECHO);
```

```

tcsetattr(STDIN_FILENO, TCSANOW, &newt);
oldf = fcntl(STDIN_FILENO, F_GETFL, 0);
fcntl(STDIN_FILENO, F_SETFL, oldf | O_NONBLOCK);

ch = getchar();

tcsetattr(STDIN_FILENO, TCSANOW, &oldt);
fcntl(STDIN_FILENO, F_SETFL, oldf);

if(ch != EOF)
{
    ungetc(ch, stdin);
    return 1;
}

return 0;
}

```

```

int Teleoperation::getch()
{
    char buf = 0;
    struct termios old = {0};
    fflush(stdout);
    if(tcgetattr(0, &old) < 0)
        perror("tcgetattr()");
    old.c_lflag &= ~ICANON;
    old.c_lflag &= ~ECHO;
    old.c_cc[VMIN] = 1;
    old.c_cc[VTIME] = 0;
    if(tcsetattr(0, TCSANOW, &old) < 0)
        perror("tcsetattr ICANON");
    if(read(0, &buf, 1) < 0)
        perror("read()");
    old.c_lflag |= ICANON;
    old.c_lflag |= ECHO;
    if(tcsetattr(0, TCSADRAIN, &old) < 0)
        perror("tcsetattr ~ICANON");
    return buf;
}

```

사용자의 키값에 따라 각각 속도를 매핑해준다.

```

if(key == 'w')
    this->target_twist_linear_x += step_linear_speed;
else if(key == 'x')
    this->target_twist_linear_x -= step_linear_speed;
else if(key == 'a')
    this->target_twist_angular_z += step_angular_speed;
else if(key == 'd')
    this->target_twist_angular_z -= step_angular_speed;
else if(key == 's' || key == ' ')
{
    this->target_twist_linear_x = 0;
    this->target_twist_angular_z = 0;
}
else if(key == 'q')
    break;
else
    RCLCPP_WARN(rclcpp::get_logger("rclcpp"), "Wrong input!");

```

Turtlebot3 Burger 의 최대 Linear 속도는 0.22 이고 최대 Angular 각도는 2.84 이다. 해당 속도를 초과하는 값은 입력받을 수 없기 때문에 calculate_max_speed()함수를 사용하여 최대 속도 여부를 판단한다.

```
geometry_msgs::msg::Twist Teleoperation::calculate_max_speed(geometry_msgs::msg::Twist twist)
{
    if(abs(twist.linear.x) > MAX_TURTLEBOT_LINEAR_SPEED)
    {
        if(twist.linear.x < 0)
            twist.linear.x = MAX_TURTLEBOT_LINEAR_SPEED * -1;
        else
            twist.linear.x = MAX_TURTLEBOT_LINEAR_SPEED;
    }
    if(abs(twist.angular.z) > MAX_TURTLEBOT_ANGULAR_SPEED)
    {
        if(twist.angular.z < 0)
            twist.angular.z = MAX_TURTLEBOT_ANGULAR_SPEED * -1;
        else
            twist.angular.z = MAX_TURTLEBOT_ANGULAR_SPEED;
    }
    return twist;
}
```

최대속도 검증 후 geometry_msgs::msg::Twist 타입의 객체를 publish 해주도록 한다.

```
void Teleoperation::publishing_data(geometry_msgs::msg::Twist twist)
{
    this->cmd_pub->publish(twist);
}
```

이제 빌드 및 실행을 해보도록 하자.

```
$ cd ~/colcon_ws
$ colcon build
$ ros2 launch turtlebot3_bringup robot.launch.py
$ ros2 run turtlebot3_tutorial_ros2 teleoperation_node
```

8.1.2. Python 을 이용한 Turtlebot3 Teleoperation

가장 먼저 python 패키지를 생성해주도록 하자.

```
$ cd ~/colcon_ws/src
$ ros2 pkg create turtlebot3_teleoperation_python --build-type ament-python --dependencies
std_msgs rclpy geometry_msgs
setup.py

from setuptools import find_packages
from setuptools import setup

package_name = 'turtlebot3_teleop'

setup(
    name=package_name,
    version='2.1.2',
    packages=find_packages(exclude=[]),
    data_files=[
        ('share/ament_index/resource_index/packages', ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
    ],
    install_requires=[
        'setuptools',
    ],
    zip_safe=True,
    author='Darby Lim',
    author_email='thlim@robotis.com',
    maintainer='Will Son',
    maintainer_email='willson@robotis.com',
    keywords=['ROS'],
    classifiers=[
        'Intended Audience :: Developers',
        'License :: OSI Approved :: Apache Software License',
        'Programming Language :: Python',
        'Topic :: Software Development',
    ],
    description=(
        'Teleoperation node using keyboard for TurtleBot3.'
    ),
    license='Apache License, Version 2.0',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'teleop_keyboard =
turtlebot3_teleoperation_python.script.turtlebot3_teleoperation:main'
        ],
    },
)
```

Turtlebot3_teleoperation.py

```
#!/usr/bin/env python

import os
import select
import sys
import rclpy
```

```

from geometry_msgs.msg import Twist
from rclpy.qos import QoSProfile

if os.name == 'nt':
    import msvcrt
else:
    import termios
    import tty

BURGER_MAX_LIN_VEL = 0.22
BURGER_MAX_ANG_VEL = 2.84

LIN_VEL_STEP_SIZE = 0.01
ANG_VEL_STEP_SIZE = 0.1

TURTLEBOT3_MODEL = os.environ['TURTLEBOT3_MODEL']

msg = """
Control Your TurtleBot3!
-----
Moving around:
    w
    a   s   d
    x

w/x : increase/decrease linear velocity (Burger : ~ 0.22, Waffle and Waffle Pi : ~ 0.26)
a/d : increase/decrease angular velocity (Burger : ~ 2.84, Waffle and Waffle Pi : ~ 1.82)

space key, s : force stop

CTRL-C to quit
"""

e = """
Communications Failed
"""

def get_key(settings):
    if os.name == 'nt':
        return msvcrt.getch().decode('utf-8')
    tty.setraw(sys.stdin.fileno())
    rlist, _, _ = select.select([sys.stdin], [], [], 0.1)
    if rlist:
        key = sys.stdin.read(1)
    else:
        key = ""

    termios.tcsetattr(sys.stdin, termios.TCSADRAIN, settings)
    return key

def print_vels(target_linear_velocity, target_angular_velocity):
    print('currently:\tlinear velocity {0}\t angular velocity {1}'.format(
        target_linear_velocity,
        target_angular_velocity))

def make_simple_profile(output, input, slop):
    if input > output:

```

```

        output = min(input, output + slop)
    elif input < output:
        output = max(input, output - slop)
    else:
        output = input

    return output

def constrain(input_vel, low_bound, high_bound):
    if input_vel < low_bound:
        input_vel = low_bound
    elif input_vel > high_bound:
        input_vel = high_bound
    else:
        input_vel = input_vel

    return input_vel

def check_linear_limit_velocity(velocity):
    if TURTLEBOT3_MODEL == 'burger':
        return constrain(velocity, -BURGER_MAX_LIN_VEL, BURGER_MAX_LIN_VEL)
    else:
        return constrain(velocity, -WAFFLE_MAX_LIN_VEL, WAFFLE_MAX_LIN_VEL)

def check_angular_limit_velocity(velocity):
    if TURTLEBOT3_MODEL == 'burger':
        return constrain(velocity, -BURGER_MAX_ANG_VEL, BURGER_MAX_ANG_VEL)
    else:
        return constrain(velocity, -WAFFLE_MAX_ANG_VEL, WAFFLE_MAX_ANG_VEL)

def main():
    settings = None
    if os.name != 'nt':
        settings = termios.tcgetattr(sys.stdin)

    rclpy.init()

    qos = QoSProfile(depth=10)
    node = rclpy.create_node('teleop_keyboard')
    pub = node.create_publisher(Twist, 'cmd_vel', qos)

    status = 0
    target_linear_velocity = 0.0
    target_angular_velocity = 0.0
    control_linear_velocity = 0.0
    control_angular_velocity = 0.0

    try:
        print(msg)
        while(1):
            key = get_key(settings)
            if key == 'w':
                target_linear_velocity =\
                    check_linear_limit_velocity(target_linear_velocity + LIN_VEL_STEP_SIZE)
                status = status + 1
                print_vels(target_linear_velocity, target_angular_velocity)
            elif key == 'x':

```

```

target_linear_velocity =\
    check_linear_limit_velocity(target_linear_velocity - LIN_VEL_STEP_SIZE)
status = status + 1
print_vels(target_linear_velocity, target_angular_velocity)
elif key == 'a':
    target_angular_velocity =\
        check_angular_limit_velocity(target_angular_velocity + ANG_VEL_STEP_SIZE)
    status = status + 1
    print_vels(target_linear_velocity, target_angular_velocity)
elif key == 'd':
    target_angular_velocity =\
        check_angular_limit_velocity(target_angular_velocity - ANG_VEL_STEP_SIZE)
    status = status + 1
    print_vels(target_linear_velocity, target_angular_velocity)
elif key == ' ' or key == 's':
    target_linear_velocity = 0.0
    control_linear_velocity = 0.0
    target_angular_velocity = 0.0
    control_angular_velocity = 0.0
    print_vels(target_linear_velocity, target_angular_velocity)
else:
    if (key == '\x03'):
        break

if status == 20:
    print(msg)
    status = 0

twist = Twist()

control_linear_velocity = make_simple_profile(
    control_linear_velocity,
    target_linear_velocity,
    (LIN_VEL_STEP_SIZE / 2.0))

twist.linear.x = control_linear_velocity
twist.linear.y = 0.0
twist.linear.z = 0.0

control_angular_velocity = make_simple_profile(
    control_angular_velocity,
    target_angular_velocity,
    (ANG_VEL_STEP_SIZE / 2.0))

twist.angular.x = 0.0
twist.angular.y = 0.0
twist.angular.z = control_angular_velocity

pub.publish(twist)

except Exception as e:
    print(e)

finally:
    twist = Twist()
    twist.linear.x = 0.0
    twist.linear.y = 0.0
    twist.linear.z = 0.0

    twist.angular.x = 0.0
    twist.angular.y = 0.0

```

```

twist.angular.z = 0.0

pub.publish(twist)

if os.name != 'nt':
    termios.tcsetattr(sys.stdin, termios.TCSADRAIN, settings)

if __name__ == '__main__':
    main()

setup.cfg

[develop]
script-dir=$base/lib/turtlebot3_teleoperation_python
[install]
install-scripts=$base/lib/turtlebot3_teleoperation_python
이제 빌드 및 실행을 해보도록 하자.

$ cd ~/colcon_ws
$ colcon build
$ ros2 launch turtlebot3_bringup robot.launch.py
$ ros2 run turtlebot3_teleoperation python turtlebot3_teleoperation.py

```

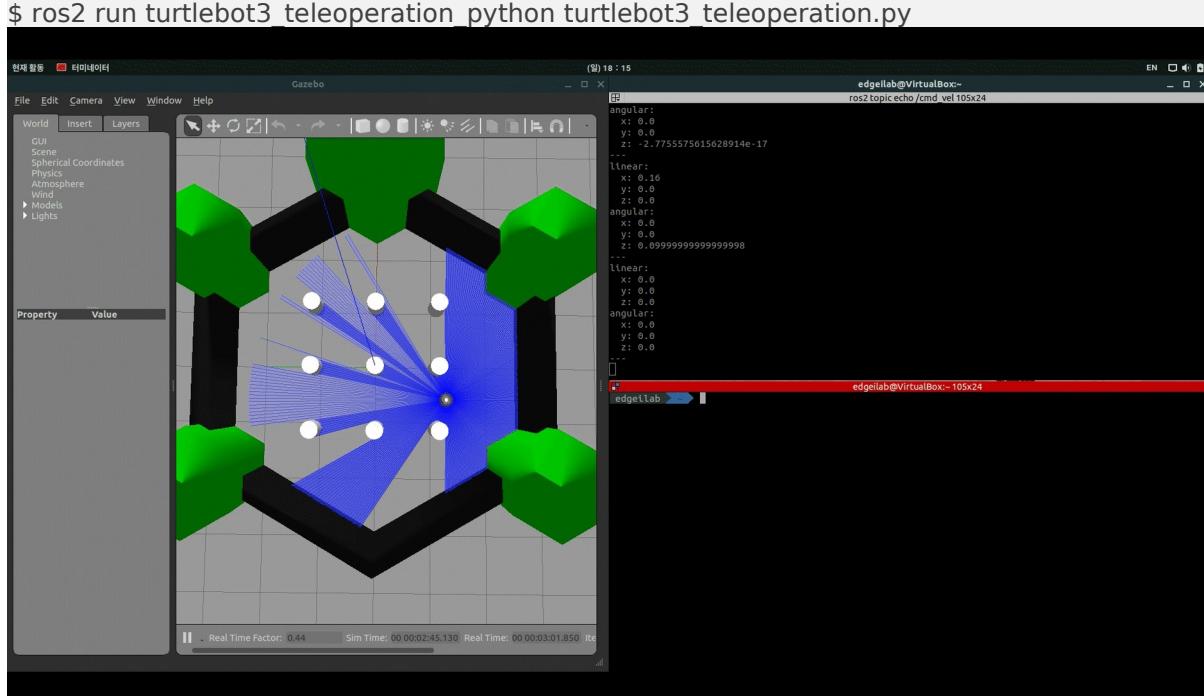


그림 101. Turtlebot3 teleoperation 실행모습

8.2. 배터리 정보 Subscribe 하기

8.2.1. C++을 이용한 배터리 정보 Subscribe 하기

패키지 생성하기

이번에 구현해 볼 패키지는 배터리 정보를 Subscribe 하는 것이다. Battery 정보는 sensor_msgs 타입으로 발행되고 있다. 해당 토픽을 subscribe 하여 화면에 출력하는 Process로 구현을 해보도록 하자.

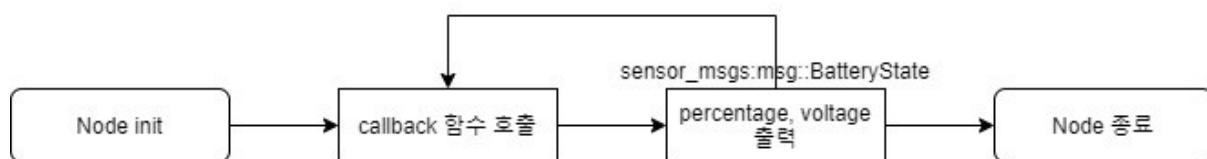


그림 102. Battery Subscirbe Process

```
$ ros2 pkg create turtlebot3_battery_sub_cpp -build-type ament_cmake -node-name turtlebot3_battery_sub_node -dependencies rclcpp std_msgs sensor_msgs
```

패키지를 생성하였다면 ~/colcon_ws/src에 turtlebot3_battery_sub_cpp 패키지 폴더와 ROS 패키지가 갖추어야 할 내부 폴더 그리고 package.xml 파일들이 생성된다.

```
.  
└── include  
    └── turtlebot3_battery_sub_cpp  
└── src  
    └── turtlebot3_battery_sub_node.cpp  
└── CMakeLists.txt  
└── package.xml
```

3 directories, 2 files

BatterySubscriber.h

```
#ifndef TURTLEBOT3_EXAMPLE_BATTERYSUBSCRIBER_H  
#define TURTLEBOT3_EXAMPLE_BATTERYSUBSCRIBER_H  
  
#include <rclcpp/rclcpp.hpp>  
#include <sensor_msgs/msg/battery_state.hpp>  
  
#include <chrono>  
#include <cstdlib>  
#include <memory>  
  
using namespace std::chrono_literals;  
using std::placeholders::_1;  
  
class BatterySubscriber : public rclcpp::Node  
{  
private:  
    rclcpp::Subscription<sensor_msgs::msg::BatteryState>::SharedPtr battery_subscriber;
```

```
public:  
    BatterySubscriber();  
    void battery_callback(const sensor_msgs::msg::BatteryState::SharedPtr msg) const;  
};
```

```
#endif //TURTLEBOT3_EXAMPLE_BATTERYSUBSCRIBER_H
```

BatterySubscriber.cpp

```
#include "BatterySubscriber.h"  
  
BatterySubscriber::BatterySubscriber() : Node("turtlebot3_battery_subscriber")  
{  
    this->battery_subscriber = this->create_subscription<sensor_msgs::msg::BatteryState>("/  
battery_state", 10, std::bind(&BatterySubscriber::battery_callback, this, _1));  
}  
  
void BatterySubscriber::battery_callback(const sensor_msgs::msg::BatteryState::SharedPtr  
msg) const  
{  
    double current_voltage = msg->voltage;  
    double current_percentage = msg->percentage;  
  
    RCLCPP_INFO(this->get_logger(), "Current Percentage : %f Current Voltage : %f",  
current_percentage, current_voltage);  
}
```

Turtlebot3_battery_subscribe_node.cpp

```
#include "BatterySubscriber.h"  
#include <rclcpp/rclcpp.hpp>  
  
int main(int argc, char** argv)  
{  
    rclcpp::init(argc, argv);  
    rclcpp::spin(std::make_shared<BatterySubscriber>());  
    rclcpp::shutdown();  
  
    return 0;  
}
```

CMakelists.txt

```
cmake_minimum_required(VERSION 3.5)  
project(turtlebot3_tutorial_ros2)  
  
# Default to C99  
if(NOT CMAKE_C_STANDARD)  
    set(CMAKE_C_STANDARD 99)  
endif()  
  
# Default to C++14  
if(NOT CMAKE_CXX_STANDARD)  
    set(CMAKE_CXX_STANDARD 14)  
endif()  
  
if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")  
    add_compile_options(-Wall -Wextra -Wpedantic)  
endif()  
  
# find dependencies  
find_package(ament_cmake REQUIRED)  
find_package(rclcpp REQUIRED)  
find_package(std_msgs REQUIRED)
```

```

find_package(sensor_msgs REQUIRED)

if(BUILD_TESTING)
    find_package(ament_lint_auto REQUIRED)
    # the following line skips the linter which checks for copyrights
    # uncomment the line when a copyright and license is not present in all source files
    #set(ament_cmake_copyright_FOUND TRUE)
    # the following line skips cpplint (only works in a git repo)
    # uncomment the line when this package is not in a git repo
    #set(ament_cmake_cpplint_FOUND TRUE)
    ament_lint_auto_find_test_dependencies()
endif()

add_executable(turtlebot3_battery_subscribe_node src /subscribe_battery_node.cpp src/
BatterySubscriber.cpp src/BatterySubscriber.h)
ament_target_dependencies(turtlebot3_battery_subscribe_node rclcpp std_msgs
sensor_msgs)
install(TARGETS
    turtlebot3_battery_subscribe_node
DESTINATION lib/${PROJECT_NAME})

ament_package()

```

소스코드는 다음과 같다. `BatterySubscriber` 클래스의 생성자에서 `battery_state` topic 을 subscribe 하는 `rclcpp::Subscription` 을 선언한다.

```

this->battery_subscriber =
this->create_subscription<sensor_msgs::msg::BatteryState>("/battery_state", 10,
std::bind(&BatterySubscriber::battery_callback, this, _1));

```

`battery_callback`을 통해 배터리 상태를 지역변수에 담은 후 `RCLCPP_INFO`를 통해 출력한다.

```

void BatterySubscriber::battery_callback(const sensor_msgs::msg::BatteryState::SharedPtr
msg) const
{
    double current_voltage = msg->voltage;
    double current_percentage = msg->percentage;

    RCLCPP_INFO(this->get_logger(), "Current Percentage : %f Current Voltage : %f",
current_percentage, current_voltage);
}

```

이제 빌드 후 실행을 해보도록 하자.

```

$ cd ~/colcon_ws
$ colcon build
$ ros2 launch turtlebot3_bringup robot.launch.py
$ ros2 run turtlebot3_tutorial ros2 subscribe_battery_node

```

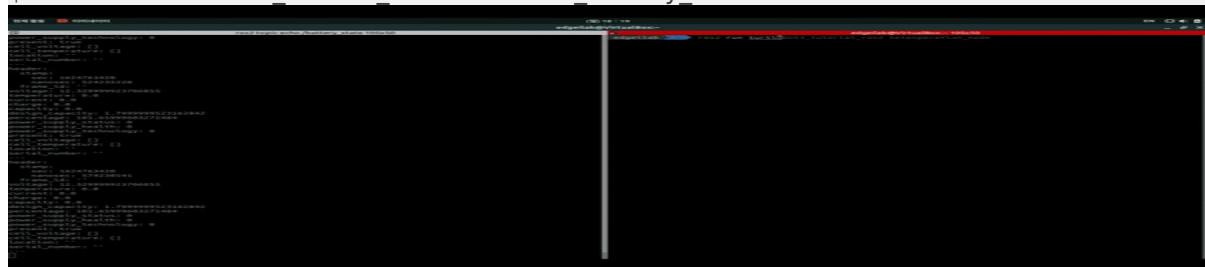


그림 103. `Battery_subscriber` 실행 모습

8.2.2. Python 을 이용한 배터리 정보 Subscribe 하기

8.3. 소리 출력하기

8.3.1. C++를 이용한 소리 출력 service 구현하기

8.4. 장애물 인식하기

8.4.1. C++을 이용한 장애물 인식하기

Work Flow

이번에 구현해 볼 패키지는 Topic 값을 subscribe 및 publishing 하며 특정 상황이 도달했을 경우 Service 를 호출하는 실습을 해보도록 하자. 해당 실습의 workflow 는 아래와 같다.

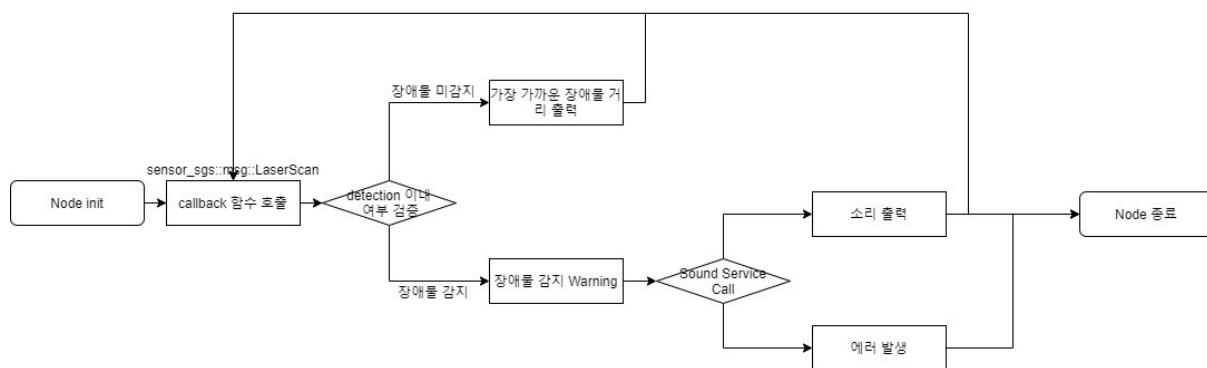


그림 104. Object Detection Work Flow

turtlebot3 가 이동하도록 /cmd_vel topic 에 값을 발행하게 된다. 이동하면서 /LaserScan 에서 값을 받아 장애물과의 거리를 계산하고 일정거리 이하로 인식되는 경우 sound 서비스에 명령을 내려보자.

패키지 생성하기

```
$ ros2 pkg object_detection_cpp -build-type ament_cmake -node-name object_detection_node  
-dependencies rclcpp std_msgs sensor_msgs turtlebot3_msgs geometry_msgs
```

패키지를 생성하였다면 ~/colcon_ws/src 에 turtlebot3_object_detection_cpp 패키지 폴더와 ROS 패키지가 갖추어야 할 내부 폴더 그리고 package.xml 파일들이 생성된다.

```
.  
└── include  
    └── turtlebot3_object_detection_cpp  
└── src  
    └── turtlebot3_object_detection_node.cpp  
└── CMakeLists.txt  
└── package.xml
```

3 directories, 2 files