ROBOTIS

로봇 운영체제
ROS 2

2 로봇 운영체제 ROS 2 특강 목차

- ROS 2의 중요 컨셉
- ROS 1과 2의 차이점을 통해 알아보는 ROS 2의 특징
- ROS 2와 DDS (Data Distribution Service)
- ROS 2 패키지 설치와 노드 실행
- ROS 2 노드와 메시지 통신
- ROS 2 토픽 (Topic), 서비스 (Service), 액션 (Action), 파라미터 (parameter)
- ROS 2 토픽,서비스,액션 정리 및 비교

ROBOTIS

ROS 2의 중요 컨셉

4 Why ROS 2? 10가지 중요 컨셉

(1) 시장 출시 시간 단축

ROS 2는 로봇 응용 프로그램을 개발하는 데 필요한 도구, 라이브러리 및 기능을 제공하므로 **본연의 중요한 로봇 개발 작업에 더 많은 시간을 할애할 수 있다는 것**이 가장 큰 장점이다. 특정 로봇 개발을 위해 처음부터 프레임워크를 만들고 통신 방법을 선정하고 디버깅 툴과 시각화 툴을 다시 만드는 비효율적인 방법에서 벗어날 수 있게 한다. 더불어 ROS 2는 상용 소프트웨어가 아닌 ROS 커뮤니티에서 개발해오고 있는 오픈소스이기 때문에 ROS 2를 사용할 부분과 사용 방법을 유연하게 결정할 수 있을 뿐만 아니라 필요에 따라 자유롭게 수정할 수 있다.

(2) 생산을 위한 설계

ROS는 로보틱스 R&D의 사실 상의 글로벌 표준으로서 ROS 1의 10년의 경험을 바탕으로, **ROS 2는 처음부터 산업용 수준으로 개발되고 높은 신뢰성과 안전을 중시**하고 있다. ROS 1의 아카데믹 성격과는 달리 ROS 2는 프로토 타이핑 개발부터 실제 생산에 이르기 위해 ROS 2의 설계, 개발 및 프로젝트 관리는 업계 이해 관계자들로부터 얻은 실직적인 요구 사항을 기반으로 하고 있다.

(3) 멀티 플랫폼

ROS 2는 **Linux, Windows, macOS**에서 개발, 지원, 테스트 진행하고 있기에 자율성, 백엔드 관리 및 사용자 인터페이스의 원활한 개발 및 배포가 가능하다. 계층형 지원 모델을 사용하고 있기에 실시간 운영체제(Real-time OS) 및 임베디드 OS(Embedded OS)와 같은 새로운 플랫폼으로의 포팅에 대한 관심과 투자를 통해 도입 및 홍보도 할 수 있다.

5 Why ROS 2? 10가지 중요 컨셉

(4) 다중 도메인

이전의 ROS 1과 마찬가지로 ROS 2는 실내에서 실외, 가정에서 자동차, 수중에서 우주, 소비자에서 산업에 이르기까지 **다양한 로봇 응용 분야에서 사용**할 수 있다.

(5) 벤더 선택 가능

ROS 2는 로봇공학 라이브러리와 응용 프로그램을 통신 기능으로부터 분리하여 추상화 작업을 해두었다. 추상화된 부분에는 오픈소스 솔루션 방식과 독점 솔루션을 포함한 여러 가지 방법론을 제공하며 그 이외의 핵심 라이브러리 및 사용자 **애플리케이션은 사용자가 원하는 형태로 개발, 수정하여 추가 가능**하다.

(6) 공개 표준 기반

ROS 2의 기본 통신 방법은 IDL, DDS 및 DDS-I RTPS과 같이 **제조 산업에서 항공 산업까지도 사용되고 있는 산업 표준**을 사용하고 있다.

(7) 자유 재량 허용 범위가 넓은 오픈소스 라이센스 채택

ROS 2 코드는 **Apache 2.0 라이센스**를 기본 라이센스로 사용하여 지적 재산권에 영향을 주지 않으면서 자유 재량으로 넓은 범위로 사용 가능하다.

6 Why ROS 2? 10가지 중요 컨셉

(8) 글로벌 커뮤니티

ROS 커뮤니티는 10년 이상 ROS 프로젝트는 소프트웨어에 기여하고 개선하는 수십만 명의 개발자와 사용자로 구성된 글로벌 커뮤니티를 육성함으로써 로봇 공학을 위한 방대한 **소프트웨어 에코 시스템**을 만들어 왔다. **ROS 2는 커뮤니티를 위해, 커뮤니티에 의해 개발**되었다.

(9) 산업 지원

ROS 2 기술 운영위원회(ROS 2 Technical Steering Committee)의 멤버쉽에서 알 수 있듯이 ROS 2에 대한 업계 지원은 강력하다. **전 세계의 크고 작은 회사는 제품을 개발할 뿐만 아니라 ROS 2에 오픈소스 기여를 하기 위해 자원을 투입**하고 있다.

(10) ROS 1과의 상호 운용성 확보

ROS 2에는 두 시스템 간의 양방향 통신을 처리하는 ROS 1에 대한 브리지가 포함되어 있다. 기존 ROS 1 애플리케이션이 있는 경우 브리지를 통해 ROS 2 테스트를 시작하고 요구 사항 및 사용 가능한 자원에 따라 점차적으로 애플리케이션을 포팅할 수 있다.

ROBOTIS

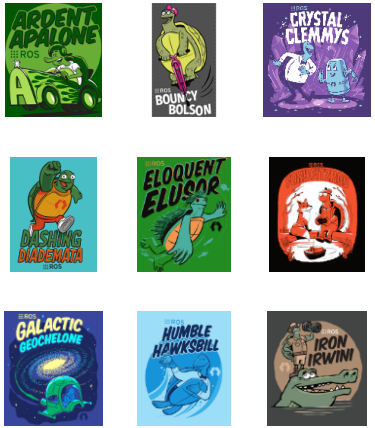
ROS 1과 2의 차이점을 통해 알아보는 ROS 2의 특징

ROS 2 vs ROS 1

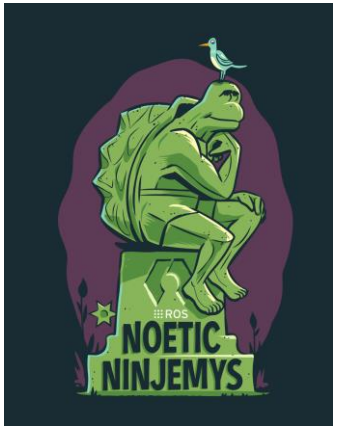
- **2024.05.23 - ROS 2 Jazzy Jalisco (LTS, 5 years support)**
- 2023.05.23 - ROS 2 Iron Irwini
- **2022.05.23 - ROS 2 Humble Hawksbill (LTS, 5 years support)**
- 2021.05.23 - ROS 2 Galactic Geochelone
- 2020.06.05 - ROS 2 Foxy Fitzroy release (LTS, 3 years support)
- 2019.11.22 - ROS 2 Eloquent Elusor release
- 2019.05.31 - ROS 2 Dashing Diademata release (First LTS, 2 years support)
- 2018.12.14 - ROS 2 Crystal Clemmys release
- 2018.07.02 - ROS 2 Bouncy Bolson release
- **2017.12.08 - ROS 2 Ardent Apalone release (1st version)**

- 2017.09.13 - ROS2 Beta3 release (code name R2B3)
- 2017.07.05 - ROS2 Beta2 release (code name R2B2)
- **2016.12.19 - ROS2 Beta1 release (code name Asphalt)**

- 2016.10.04 - ROS2 Alpha8 release (code name Hook.and.Loop)
- 2016.07.14 - ROS2 Alpha7 release (code name Glue Gun)
- 2016.06.02 - ROS2 Alpha6 release (code name Fastener)
- 2016.04.06 - ROS2 Alpha5 release (code name Epoxy)
- 2016.02.17 - ROS2 Alpha4 release (code name Duct tape)
- 2015.12.18 - ROS2 Alpha3 release (code name Cement)
- 2015.11.03 - ROS2 Alpha2 release (code name Baling wire)
- **2015.08.31 - ROS2 Alpha1 release (code name Anchor)**



ROS 2 Jazzy Jalisco (LTS, 5 years support)



- ROS Noetic Ninjemys
- Released May, 2020
- LTS, supported until May, 2025

9 ROS 1과 2의 차이점

ROS 1은 2007년에 개발이 시작되어 지금은 대학, 연구 기관, 산업계, 로봇 자작의 취미활동까지 폭넓게 이용되고 있다. 원래 **ROS 1은 Willow Garage사가 개인 서비스 로봇인 PR2개발에 필요한 미들웨어 형태의 로봇 개발 프레임워크를 다양한 개발 툴과 함께 오픈 소스로 공개 한 것으로 시작**하였다.
따라서 개발 환경으로는 PR2의 초기 컨셉을 그대로 이어받아 다음과 같은 **제한 사항**이 있었다.

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

10 ROS 1과 2의 차이점

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 이용**되고 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 기능**을 정리하면 다음과 같다.

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

11 ROS 1과 2의 차이점

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 2**라고, 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.

12 ROS 2의 20가지 특징 1. Platforms

(1) Platforms

ROS 2부터는 3대 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 Jazzy Jalisco 기준으로 보았을 때 Linux는 Ubuntu Noble (24.04), Windows는 Windows 10 버전, macOS는 Mojave (10.14)버전을 지원하고 있다. Linux의 경우에는 Linux 배포판중 일반 사용자가 가장 많은 Canonical의 Ubuntu 진영에서 ROS 2 TSC[13]로 가입되어 있어서 관련된 내용을 많이 볼 수 있다. 리눅스 이용자라면 Canonical에서 연재 중인 아래 참고 자료 [14], [15]의 내용을 참고하면 ROS 2 사용에 도움이 될 듯싶다. 그리고 Linux, macOS는 ROS 1 부터 지원하고 있었는데 이번 ROS 2에서는 Microsoft가 ROS 2 TSC로 들어오고 Windows용 패키지 및 테스트, Visual Studio Code Extension for ROS [16]까지 준비하는 등 굉장한 노력을 기울여 Windows 사용자들도 ROS 2를 쉽게 사용할 수 있게 되었다. Windows 사용자라면 [17], [18]의 Windows 관련 문서를 참고하기를 추천한다.

13 ROS 2의 20가지 특징 3. Security

(2) Security

ROS 1에서는 항상 보안이 문제였다. 노드를 관리하는 ROS master[23]의 하나의 IP와 포트만 노출되면 모든 시스템을 죽일 수 있었으며, 보안 입장에서는 TCPROS[24]는 뻥 뚫린 큰 구멍에 가까웠다. 하지만 ROS 1은 이러한 부족한 부분을 매우기 보다는 로봇 개발에 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.

하지만 시간이 흘러 이러한 취약점은 **상용 로봇에 ROS를 도입할 수 없게 만드는 첫번째 이유이자 가장 큰 걸림돌**이 되었다. 이에 ROS 2에서는 디자인 설계부터 이 부분을 명확히 짚고 넘어갔다. 우선, TCP 기반의 통신은 **OMG(Object Management Group)**[25]에서 산업용으로 사용 중인 **DDS(Data Distribution Service)**[26]를 도입하였고, 자연스럽게 **DDS-Security** 이라는 DDS 보안 사양[27]를 ROS에 적용하여 보안에 대한 이슈를 통신단부터 해결하였다. 또한 ROS 커뮤니티에서는 **SROS 2(Secure Robot Operating System 2)**[28]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 문서([29], [30], [31], [32])의 `Security` 관련 문서를 참고하길 바라며 지난 ROSCon2019에서의 워크샵에서 진행한 `Is your robot secure? ROS 1 & ROS 2 Security Workshop`[33]의 문서도 참고하면 도움이 될 것 같다. 우리는 추후 강좌를 통해 직접 보안 설정 및 프로그램하는 실습을 진행해보기로 하자.

3. Communication

ROS 1에서는 자체 개발한 TCPROS[24]와 같은 통신 라이브러리를 사용하고 있던 반면, ROS 2은 리얼타임 퍼블리시와 서브스크라이브 프로토콜인 **RTPS(Real Time Publish Subscribe)**[20]를 지원하는 통신 미들웨어 **DDS**[26], [34]를 사용하고있다. DDS는 **OMG**(Object Management Group, [25])에 의해 **표준화**가 진행되고 있으며, 상업적인 용도에도 적합하다는 평가가 지배적이다. DDS에서는 IDL(Interface Description Language, [35])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 지원**하고 있어서 기존 ROS 1에서 각 노드들의 정보를 관리하였던 ROS마스터가 없어도 여러 DDS 프로그램 간에 통신 할 수 있다. 또한 노드 간의 통신을 조정하는 **QoS(Quality of Service)**[36] 매개변수를 설정할 수 있어서 TCP 처럼 데이터 손실을 방지함으로써 신뢰도를 높이거나, UDP 처럼 통신 속도를 최우선하여 사용할 수도 있다. 이러한 다양한 기능을 갖춘 DDS를 이용하여 ROS1의 **퍼블리시, 서브스크라이브 형 메세지 전달은 물론, 실시간 데이터 전송, 불안정한 네트워크에 대한 대응, 보안 강화** 등이 강화되었다. **DDS의 채용은 ROS1에서 ROS2로 바뀌면서 가장 큰 변화점**이다.

4. Middleware interface

앞서 설명한 DDS는 다양한 기업에서 통신 미들웨어 형태로 제공하고 있다. 그 벤더로는 [37]의 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 DDS**가 있다. 참고로 이 중 Gurum Network는 유일하게 대한민국 기업으로 DDS를 순수 국산 기술로 개발하여 상용화에 성공한 기업이다.

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

5. Node manager (discovery)

ROS 1에서의 필수 실행 프로그램으로는 roscore[39]가 있다. 이를 실행시키면 ROS Master [40], ROS Parameter Server [41], rosout logging node [42]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 있게해 주는 매우 중요한 중매 역할을 수행했었다. 이 때문에 ROS 1에서는 노드 사이의 연결을 위해 네임 서비스를 마스터에서 실행했었어야 했고, 이 ROS Master가 연결이 끊기거나 죽는 경우 모든 시스템이 마비되는 단점이 있었다.

ROS 2에서는 roscore가 없어지고 3가지 프로그램이 각각 독립 수행으로 바뀌었다. 특히, **ROS Master의 경우 완전히 삭제**되었는데 이는 DDS를 사용함에 따라 노드를 **DDS**의 Participant [43]개념으로 취급하게 되었으며, **Dynamic Discovery** [43], [44]기능을 이용하여 **DDS 미들웨어**를 통해 직접 검색하여 노드를 연결 할 수 있게 되었다. 이제 ROS 2에서는 roscore는 Bye~ Bye~ 다.

6. Languages

ROS 2의 프로그램 언어로는 ROS 1과 마찬가지로 **다양한 프로그래밍 언어를 지원**할 예정이다. 아직까지는 C++[45], Python[46]가 주력 언어라고 볼 수 있는데 이 주력 언어도 아래와 같이 큰 변화가 있었다. ROS 2 배포판[9]마다 조금씩 다르기는 하지만 Jazzy를 기준으로 보았을 때, **C++은 C++17, 파이썬은 Python 3.8이 기본 요구 사항**으로 되어있다. 이다. 같은 언어를 쓰더라도 ROS 1을 사용했을 때에는 뭔가 올드한 느낌이었고 최신 언어들이 제공하는 기능들을 못쓰고 그림에 떡이였는데 이제는 최신의 아름다운 언어를 쓰는 기분이다. 이는 사용해보면 알게 될 것이다. 물론 새로운걸 배운다는 것은 어쩔 수 없는 엔지니어의 숙명이다.

- ROS 1: C++03, Python 2.7
- ROS 2: C++17, Python 3.8

7. Build system

ROS 2에서는 새로운 빌드 시스템인 **ament**[47]을 사용한다. ament는 ROS 1에서 사용되는 빌드 시스템인 catkin[48]의 업그레이드 버전이다. ROS 1의 catkin이 CMake만을 지원했던 반면, ament는 CMake를 사용하지 않는 Python 패키지 관리도 가능하다. 즉, ROS 2에 와서는 Python 패키지는 비로서 처음으로 완전 독립을 이루게 되었는데 ROS 1에서 Python 코드가 있는 패키지는 setup.py 파일이 CMake 내에서 사용자 정의 로직으로 처리되었다. 하지만 ROS 2에서 Python 패키지는 setup.py 파일의 모든 기능을 순수 Python 모듈과 동등한 수준으로 개발할 수 있게 되었다. 마지막으로 TMI일 수 있으나 catkin과 ament는 이음동의어로 버드나무의 화수를 의미하며 ROS 1의 개발 주체인 Willow Garage 뒷 마당에 있던 버드나무 화수를 보고 지었다고 한다. 즉, 뭔가 심오한 뜻이 있는 것은 아니다.

- ROS 1: rosbuild → catkin (CMake)
- ROS 2: ament (CMake), Python setuptools (Full support)

8. 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_tools[49]가 이용되었고 지금에 와서는 colcon[50]을 추천하고 있다. **colcon**은 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. 이는 추후 이어지는 강좌들에서 더 자세히 다루도록 하겠다.

20 ROS 2의 20가지 특징 10. Build options

9. Build options

ROS 2에서는 빌드 관련 내용들이 모두 변경되면서 빌드 옵션에도 새로운 변화[47]가 생겼다. 그중 사용하면서 가장 좋았던 3가지를 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.

10. Build options

셋째는 `**No devel space**`이다.

catkin은 패키지를 빌드 한 후 devel 이라는 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.

11. Version control system

ROS는 수 많은 소스 코드 공여자로부터 만들어가는 코드의 집합이기 때문에 개인은 물론 소속도 정말 다양하고 각 코드들의 리포지토리도 제각각이다. 예를 들어 어느 패키지는 GitHub를 이용하고 어떤 것은 Bitbucket를 이용한다. 그리고 사용하는 버전 관리 시스템(Version Control System, VCS)도 Git, Mercurial, Subversion, Bazaar 등 다양하다.

- ROS 1: rosws → wstool, rosinstall (*.rosinstall)
- ROS 2: vcstool (*.repos)

ROS 커뮤니티에서는 이러한 다양한 리포지토리와 혼재된 버전 관리 시스템을 사용하더라도 ROS를 사용함에 있어서 불편함이 없도록 통합적인 툴이 필요했다. ROS 1에서는 처음에 rosws[51]이라는 툴에서 wstool[52]을 이용하였다가 최근 ROS 2에서는 **vcstool**[55]으로 통합하였다.

11. Version control system

현재 ROS 1에서도 vcstool를 사용하고 있는 상황이다. vcstool은 여러 리포지토리 작업을 보다 쉽게 관리할 수 있도록 설계된 버전 관리 시스템(VCS) 툴이다. 이 툴은 ROS 2를 소스코드로부터 설치해본 사람이라면 자신도 모르게 사용했을 것이다. 아래의 명령어 2줄을 살펴보자. 우선 wget을 통하여 ros2.repos라는 파일을 받게 되는데 이 파일에는 vcs 타입은 무었이고, 리포지토리 주소는 어떻게 되며, 설치해야하는 브랜치는 어떤 것인지가 명시된 파일이다. 이러한 정보가 기재된 *.repos 파일을 이용하여 다양한 리포지토리, 다양한 vcs를 지원하며 패키지들을 관리할 수 있도록 하는 것을 의미한다. 특히 ROS 2에서는 기존 vcs 툴을 통폐합하여 **vcstool** 이라는 이름으로 제공되어 사용에 매우 편리하게 되었다. 자세한 사용법은 [56]의 README 파일을 참고하도록 하자.

```
wget https://raw.githubusercontent.com/ros2/ros2/foxy/ros2.repos
vcs import src < ros2.repos
```

24 ROS 2의 20가지 특징 12. Client library

12. Client library

ROS 기반의 프로그래밍을 작성한다는 것은 ROS 구조[57]에서 유저 코드 영역(user land)을 다룬다는 것으로 그 밑에는 ROS 클라이언트 라이브러리 (ROS Client Library)이 있고, 이 클라이언트 라이브러리는 앞서 설명한 미들웨어(middleware interface)를 사용하고 있다는 것을 알고 있어야 한다. 여기서 유저는 개발 목적에 따라 C/C++, Python, Java, Node.js 등을 사용할 것이다. ROS 에서는 초창기 부터 이러한 멀티 프로그래밍 언어를 지원하고 있는데 ROS 1에서는 roscpp, rospy, roslisp 등 각 프로그래밍 언어에 대해 클라이언트 라이브러리 (Client Library, [58])를 제공했다. 한편, ROS 2에서는 ROS 클라이언트 라이브러리를 **RCL(ROS Client Library)**이라는 이름으로 제공한다. 그리고 프로그래밍 언어별로 **rclcpp, rclc, rclpy, rcljava, rclobjc, rclada, rclgo, rclnodejs** [59]등으로 제공된다. 또한 ROS 2는 앞서 설명한바와 같이 C이면 C99, C++ 이라면 C++ 14/17, Python라면 Python 3 (3.5+) 등 최신 기술 사양에 대응하고 있다. 각 C++/Python ROS Client Library API는 [60], [61]을 미리 봐둔다면 도움일 될 것이다.

[참고 링크]
[57] http://design.ros2.org/articles/ros_middleware_interface.html
[58] https://docs.ros.org/en/rolling/Concepts/Basic/About-Client-Libraries.html
[59] https://github.com/fkromer/awesome-ros2#client-libraries
[60] http://docs.ros2.org/foxy/api/rclcpp/index.html
[61] http://docs.ros2.org/foxy/api/rclpy/index.html

13. Life cycle

로봇 개발에 있어서 로봇의 현재 상태를 파악하고 현재 상태에서 다른 상태로 변경되는 상태천이 제어는 수십년간 로봇공학에서도 주요 연구 주제로 다루었던 중요한 부분 중에 하나이다. 특히 태스크 수행 측면에서 현재의 상태 파악과 천이는 멀티 태스크 수행에서 빠질 수 없는 중요한 부분일 것이고 복수의 로봇 복수의 복합 태스크, 서비스 수행과 같은 상위 레벨의 프로그램일수록 더 중요하게 다루어지는 부분이다. ROS 1에서는 이러한 기능을 구현하기 위해서는 SMACH[62]과 같은 상태 천이를 관리하는 독립적인 패키지를 사용했어야 했고 클라이언트 라이브러리에서는 상태 관리하는 부분이 없었기에 사용자가 임의로 클라이언트 라이브러리 부분까지 수정하여 사용했어야 했다.

ROS 2에서는 이러한 니즈를 반영하여 패키지의 각 노드들의 현재 상태를 모니터링하고 상태를 제어 가능한 **lifecycle**[63]을 클라이언트 라이브러리에 포함시켰으며 이를 통해 ROS 시스템 상태를 보다 효과적으로 제어할 수 있게 되었다. 이를 이용하게 되면 기존 ROS 1에서는 할 수 없었던 노드의 상태를 모니터링하고 상태를 천이시키거나 노드를 상태에 따라 재시작하거나 교체할 수도 있게 된다.

[참고 자료]
[62] http://wiki.ros.org/smach
[63] http://design.ros2.org/articles/node_lifecycle.html

14. Multiple nodes

ROS 1의 초기에는 하나의 프로세스에서 여러 노드를 실행 할 수 없었다. 하지만 이러한 요구는 지속적으로 제기되었고 하나의 프로세스에서 여러 노드를 작성하기 위해 nodelet[64]라는 새로운 기능이 ROS 1에 추가되었다. 이는 하드웨어 리소스가 제한적이거나 노드 간에 수 많은 메시지를 보내야 할 때 유용하게 사용되었다.

ROS 2에서 nodelet이 사용되지는 않고 RCL에 포함되어 있다. 이름은 컴포넌트(**components**, [65])라고 부르며 ROS 2에서는 이 컴포넌트를 사용하여 동일한 실행 파일에서 복수의 노드를 수행할 수 있게 되었다. 이를 사용하게 되면 노드의 실행 파일 수준은 더 세분화 시킬 수 있으며 프로세스 내 통신 **IPC(intra-process communication)**[66]기능을 이용하여 ROS 2의 통신 오버 헤드를 제거 할 수 있어서 더 효율적인 ROS 2 응용 프로그램을 작성 가능하다.

[참고 자료]
[64] http://wiki.ros.org/nodelet
[65] https://docs.ros.org/en/rolling/Concepts/Intermediate/About-Composition.html
[66] https://docs.ros.org/en/rolling/Tutorials/Demos/Intra-Process-Communication.html

15. Threading model

ROS 1에서 개발자는 단일 스레드 실행 또는 다중 스레드 실행 중 하나만 선택할 수 있었다. ROS 2에서는 더 세분화 된 실행 모델(executor)을 C++과 Python에서 사용할 수 있으며 사용자가 정의한 실행기도 제공되는 RCL API를 이용하여 쉽게 구현할 수 있다. **Single Threaded Executor**, **Multi Threaded Executor** [67], [68]에 대한 설명은 이어지는 강좌에서 예제와 함께 알아보도록 하자.

[참고 자료]
[67] https://github.com/ros2/rclcpp/tree/master/rclcpp/src/rclcpp/executors
[68] https://github.com/ros2/rclpy/blob/master/rclpy/rclpy/executors.py

16. Messages (topic, service, action)

ROS 2에서도 기존 ROS 1의 메시지(Messages, [69])과 마찬가지로 단일 데이터 구조를 메시지라고 정의하며 정해진 또는 사용자가 정의한 메시지를 사용할 수 있으며, 각 패키지 이름과 마찬가지로 이름과 각 지정된 형식으로 메시지를 고유하게 식별할 수 있다. 사용처도 기존과 마찬가지로 Topic, Service, Action 등에서 사용하며 기존과 비슷한 형태([70], [71])로 사용 가능하다.

여기에 ROS 2에서는 OMG(Object Management Group, [25])에서 정의된 IDL(Interface Description Language, [35], [72])을 사용하여 메시지 정의 및 직렬화를 더 쉽게, 더 포괄적으로 다룰 수 있게 되었다. IDL을 이용하게 되면 기존 ROS 메시지 컨셉과 마찬가지로 다양한 프로그래밍 언어로 작성된 메시지를 사용할 수 있다. 이전에 CORBA (일명, 코바)를 써본 사람들은 IDL이 친숙할 것이다. ROS2에서는 기존 **msg**, **srv**, **action** 파일 이외에도 **IDL**을 지원한다. 그리고 ROS 2가 DDS를 채용하게 되면서 기존 메시지들과 DDS 규칙을 맞추는 작업이 진행되었다. ROS 인터페이스 유형과 DDS IDL 유형 간의 맵핑은 [73]번 자료를 참고하면 좋을 듯 싶으며 전체적으로 다듬어진 정리표는 [74]번 글을 참고하도록 하자.

16. Messages (topic, service, action)

그리고, ROS 2에서는 DDS를 사용하면서 메시지를 이용한 Topic, Service, Action 등의 컨셉은 변하지 않으나 사용 방법은 상당히 많이 바뀌었다. 이 부분에 대한 설명은 이어지는 강좌를 통해 예제와 함께 하나 하나 알아가보자.

[참고 자료]
[69] http://wiki.ros.org/Messages
[70] http://design.ros2.org/articles/interface_definition.html
[71] http://design.ros2.org/articles/legacy_interface_definition.html
[72] http://design.ros2.org/articles/idl_interface_definition.html
[73] http://design.ros2.org/articles/mapping_dds_types.html
[74] https://docs.ros.org/en/rolling/Concepts/Basic/About-Interfaces.html

17. Command Line Interface

대부분의 CLI 타입의 명령어 사용법은 기존 ROS 1과 매우 비슷해서 약간의 이름 변경과 일부 옵션 사용법만 익힌다면 사용시 큰 차이는 없다. 자주 사용되는 명령어를 예를 들어 보자면 아래와 같은 차이 정도이다. ROS 1 명령어에 비해 명령어가 약간 길어진듯 보이긴 하지만 자주 쓰는 명령어는 "alias rt='ros2 topic list'"와 같이 설정하여 사용하면 되기에 큰 무리는 없다. 더욱이 ROS 2의 CLI 형태의 명령어는 ROS 2 TSC 멤버이자 Ubuntu 개발 업체인 Canonical이 담당하고 있어서 더욱 믿음이 간다. 자세한 설명은 [75]를 참고하기 바라며 실습을 해보고 싶다면 [76] 발표 자료를 참고하면 좋다.

- ROS 1: 'rostopic list'
- ROS 2: 'ros2 topic list'

[참고 자료]

[75] https://ubuntu.com/blog/ros-2-command-line-interface
[76] https://roscon.ros.org/2018/presentations/ROSCon2018_ROS2HandsOn.pdf

18. roslaunch

ROS의 실행 시스템은 대표적으로 `run`과 `launch`가 있는데 `run`은 단일 프로그램 실행, `launch`는 사용자 지정 프로그램 실행을 수행한다. 사용면에서는 `run`에 비해 다양한 설정을 할 수 있는 `launch` 사용이 월등히 사용 빈도가 높다. `launch`는 사용자가 실행하고자하는 프로그램의 각종 설정을 기술하고 기술된 설정에 맞추어 각종 프로그램을 실행하도록 도와준다. 사용자가 지정하는 설정에는 실행할 프로그램, 실행할 위치, 전달할 인수 등 시스템 전체의 구성 요소를 쉽게 재사용 할 수 있도록 하고있다. 그 목적과 컨셉은 ROS 2에서도 크게 다르지 않다.

ROS 1과 ROS 2의 차이점[77]을 살펴보면 다양한 파일 사용이다. ROS 1에서는 `roslaunch` 파일이 특정 `XML` 형식을 사용해었다. 이 형식을 이용해도 다양한 설정을 추가하여 프로그램을 실행 시킬 수 있어서 매우 편했는데, ROS 2에서는 `XML` [78], [79] 형식 이외에도 **`Python`이 새롭게 채용되어 조건문 및 Python 모듈을 추가로 사용하여 보다 복잡한 논리와 기능을 사용할 수 있게 되었다.** 어떤 방식으로 사용하는지에 대한 추가 설명은 [80] 튜토리얼을 참고하도록 하자.

[참고 자료]
[77] http://design.ros2.org/articles/roslaunch.html
[78] http://design.ros2.org/articles/roslaunch_xml.html
[79] http://design.ros2.org/articles/roslaunch_frontend.html
[80] https://docs.ros.org/en/rolling/Tutorials/Intermediate/Launch/Launch-Main.html

19. Graph API

ROS 1에서 `rqt_graph` [81]툴을 자주 사용했을 것이다. 이는 ROS에서 각 노드와 토픽, 메시지 등이 고유의 이름을 가지고 있고 매핑이 이루어져 각 노드와 노드간의 토픽, 메시지의 관계를 그래프화 시킬 수 있었기 때문이다. 단, 한가지 불편한 점은 ROS 1에서는 시작할 때만 노드와 토픽간의 매핑이 이루어져서 그 관계를 그래프로 표현할 수 있었기 때문에 실시간으로 변화되는 그래프를 볼 수 없었다. 이에 ROS 2에서는 노드 시작할 때뿐만 아니라 실행 도중에 다시 매핑도 가능하며, 그 결과를 바로 그래프로 표현할 수 있게하려 하고 있다.

[참고 자료]
[81] http://wiki.ros.org/rqt_graph
[82] https://docs.ros.org/en/rolling/Concepts.html

20. Embedded Systems

로봇 개발에 있어서 실시간성을 담보 받으며 모터 및 센싱을 제어하는 부분은 매우 중요하게 다루어져 왔다. 이 강좌 초반에서 언급했듯이 ROS 커뮤니티에서도 이를 위해 ROS 2에서는 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. 하지만 실시간성이라는 것은 상위 소프트웨어에서 다루기에는 제약이 많다. 오히려 Embedded Systems 안에서 해결하는게 더 적합하다고 보고 있다.

이에 ROS 2 개발 초기부터 Embedded Systems에 대한 관심이 높았는데 초기 개발 컨셉은 ROS의 창시자인 Morgan Quigley가 ROSCon2015에서 `ROS 2 on “small” embedded systems`이라는 이름으로 발표한 자료[83] 및 영상[84]를 보면 도움이 될 것이다. ROS 1에서도 Embedded Systems을 지원하지 않는 것은 아니였다. 단, 매우 기초적인 수단이라고 볼 수 있는 임베디드 보드와 메시지를 주고 받을 때 시리얼(rosserial, [85])로 통하여 통신하였다. ROS 2에서는 한발 더 나아가 기존 시리얼 통신, 블루투스 및 와이파이 통신을 지원하거나 RTOS (Real-Time Operating System)를 사용하고 기존 DDS 대신 eXtremely Resource Constrained Environments (DDS-XRCE)를 사용 하는 등 임베디드 보드에서 직접 ROS 프로그래밍을 하여 하드웨어 펌웨어로 구현된 노드를 실행할 수도 있다. 이 방법론에는 여러 가지가 있을 수 있는데 현재 ARM 사를 포함한 다양한 MCU 제조 업체에서 이를 지원하기 위하여 다양한 방법론을 내놓고 있는 상태이고, eProsima, BOSCH, ROBOTIS, FIWARE, Amazon, Renesas 등에서 다음 참고 자료와 같이 다양한 임베디드 지원 방법에 대해 개발, 공개하고 있다. 관심

20. Embedded Systems

ROS 1: rosserial, mROS
ROS 2: micro-ROS, XEL Network, ros2arduino, Renesas, DDS-XRCE(Micro-XRCE-DDS), AWS ARCLM

[참고 자료]
[83] https://roscon.ros.org/2015/presentations/ros2_on_small_embedded_systems.pdf
[84] https://vimeo.com/142150576
[85] http://wiki.ros.org/rosserial
[86] https://micro-ros.github.io/
[87] http://xelnetwork.robotis.com/
[88] https://github.com/ROBOTIS-GIT/ros2arduino
[89] https://www.renesas.com/us/en/solutions/key-technology/robot/robot-operating-system.html
[90] https://micro-xrce-dds.docs.eprosima.com/en/latest/

ROBOTIS

ROS 2와 DDS (Data Distribution Service)

36 ROS의 메시지 통신

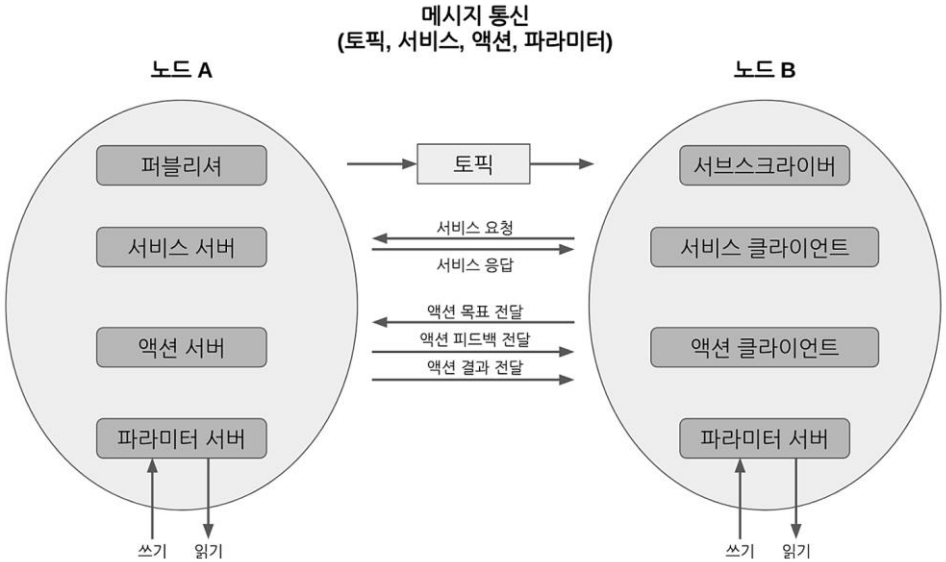
로봇 운영체제 ROS에서 중요시 여기는 몇 가지 용어 정의 및 메시지, 메시지 통신에 대해 먼저 알아보도록 하자. 특히, **메시지 통신**은 ROS 프로그래밍에 있어서 **ROS 1과 2의 공통된 중요한 핵심 개념**이기에 ROS 프로그래밍에 들어가기 전에 꼭 이해하고 넘어가야 할 부분이다.

ROS에서는 프로그램의 재사용성을 극대화하기 위하여 최소 단위의 실행 가능한 프로세스라고 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 데이터**를 서로 주고받게 설계해야만 한다.

37 ROS의 메시지 통신

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 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 구분된다.

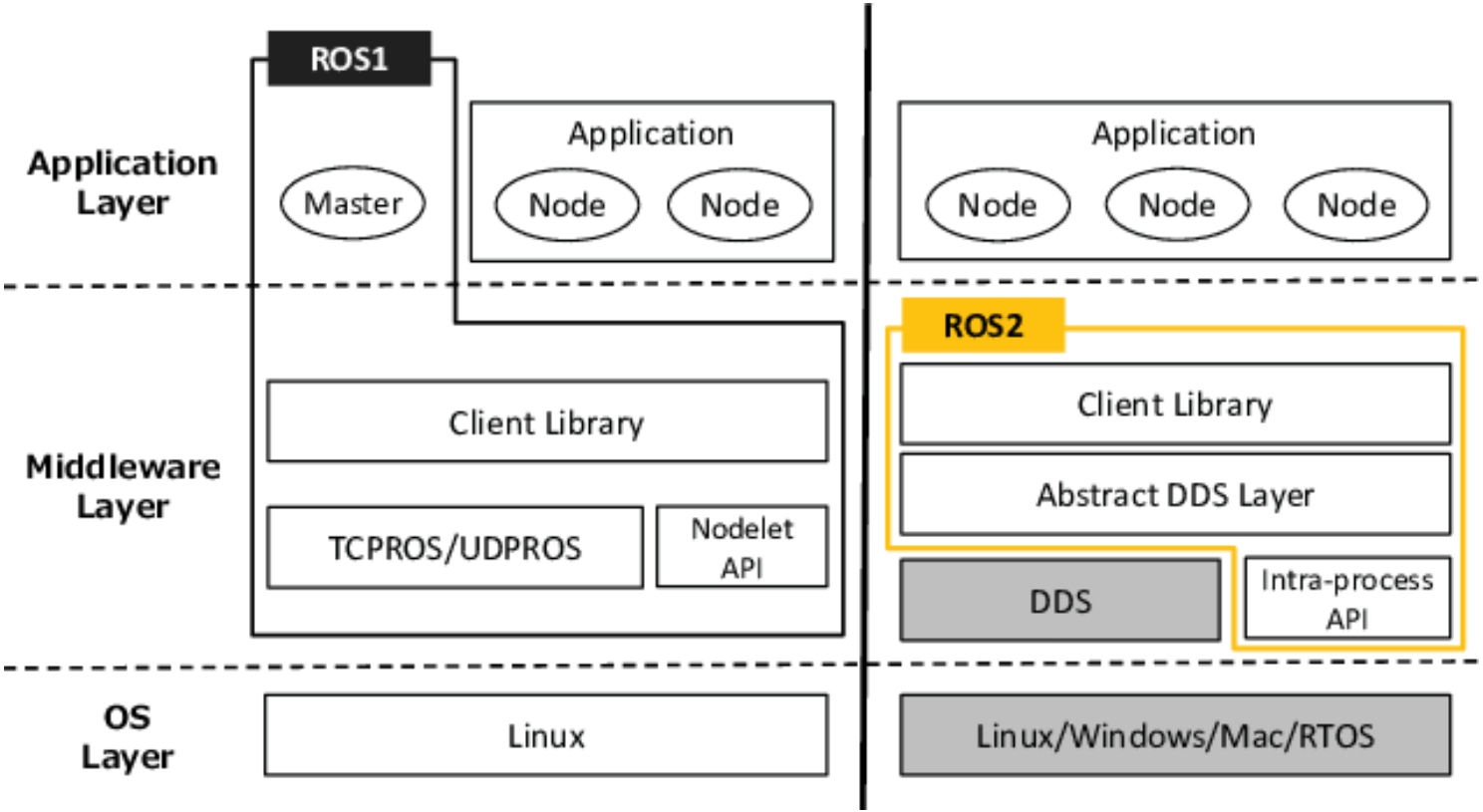


38 ROS 2와 DDS

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 개념**을 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.

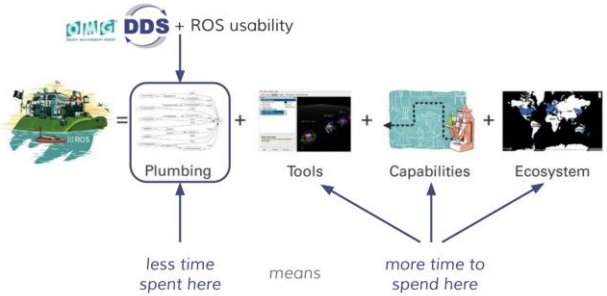
ROS 1에서는 자체 개발한 TCPROS와 같은 통신 라이브러리를 사용하고 있던 반면, ROS 2에서는 OMG(Object Management Group)에 의해 표준화된 **DDS(Data Distribution Service)**의 리얼타임 퍼블리시와 서브스크라이브 프로토콜인 **DDSI-RTPS(Real Time Publish Subscribe)**를 사용하고 있다. ROS 2 개발 초기에는 기존 TCPROS를 개선하거나 ZeroMQ, Protocol Buffers 및 Zeroconf 등을 이용하여 미들웨어처럼 사용하는 방법도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

ROS 2와 DDS


ROS1
Application Layer
Application
Master
Node
Node
Application
Node
Node
Node
ROS2
Middleware Layer
Client Library
TCPROS/UDPROS
Nodelet API
Client Library
Abstract DDS Layer
DDS
Intra-process API
OS Layer
Linux
Linux/Windows/Mac/RTOS

40 ROS 2와 DDS

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 프로토콜로인 **DDSI-RTPS**을 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 측면에서도 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 다음 그림과 같이 개발자 및 사용자로 하여금 통신 미들웨어에 대한 개발 및 이용 부담을 줄여 진짜로 집중해 ...게 되었다.**



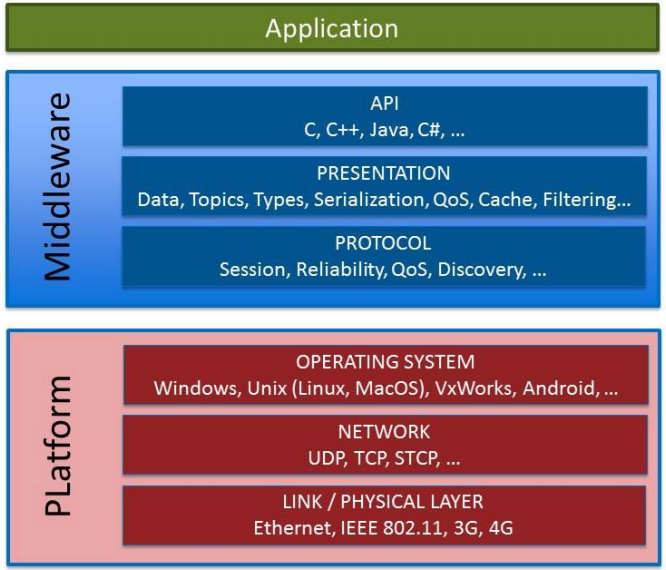
41 DDS란?

자~ 이제 기본 배경이 되는 썰을 풀었으니 본격적으로 DDS에 대해 알아보자. 처음 DDS를 ROS 2에 도입하자는 이야기가 나왔을 때, DDS라는 단어 자체를 처음 들어봤기에 너무 어려웠다. 결론부터 말하자면 DDS는 데이터 분산 시스템이라는 용어로 OMG에서 표준을 정하고자 만든 트레이드 마크(TM)였다. 그냥 용어이고 그 실체는 데이터 통신을 위한 미들웨어이다.

DDS가 ROS 2의 미들웨어로 사용하는 만큼 그 자체에 대해 너무 자세히 알 필요는 없을 듯싶고 ROS 프로그래밍에 필요한 개념만 알고 넘어가면 될 듯싶다. 우선 정의부터 알아보자. **DDS**는 **Data Distribution Service**, 즉 **데이터 분산 서비스**의 약자이다.

DDS란?

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 ISO 7 계층 레이어[32]에서 호스트 계층(Host layers)에 해당되는 4~7 계층에 해당되고 ROS 2에서는 위에서 언급한 다음 그림과 같이 운영 체제와 사용자 애플리케이션 사이에 있는 소프트웨어 계층으로 이를 통해 시스템의 다양한 구성 요소를 보다 쉽게 통신하고 데이터를 공유할 수 있게 된다.



43 DDS의 10가지 특징

DDS의 특징은 다양하겠지만 DDS를 ROS 2의 미들웨어로 사용해보면서 느낀 장점은 아래와 같이 10가지이다. 여기서는 이 10가지에 대해 하나씩 정리해보고 각 기능들은 이어지는 강좌에서 실습을 통해 더 자세히 알아보자.

1. **Industry Standards**
2. **OS Independent**
3. **Language Independent**
4. **Transport on UDP/IP**
5. **Data Centricity**
6. **Dynamic Discovery**
7. **Scalable Architecture**
8. **Interoperability**
9. **Quality of Service (QoS)**
10. **Security**

44 DDS의 특징 1. 산업 표준

1. 산업 표준

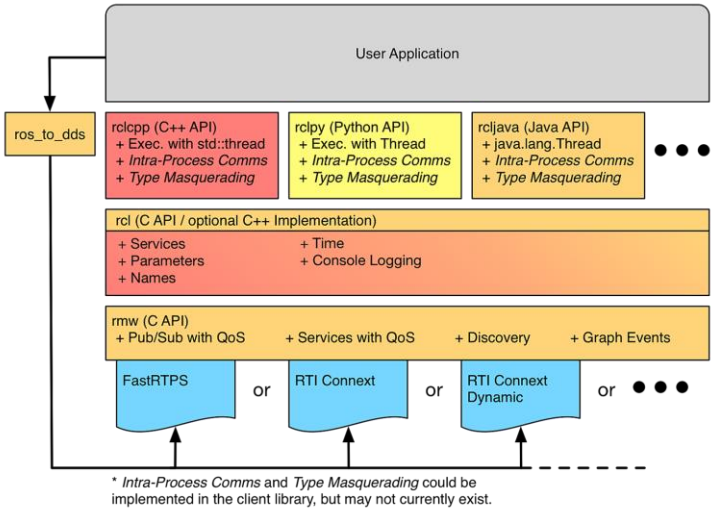
DDS는 분산 객체에 대한 기술 표준을 제정하기 위해 1989년에 설립된 비영리 단체인 OMG(Object Management Group, 객체 관리 그룹)가 관리하고 있는 만큼 산업 표준으로 자리 잡고 있다. 지금까지 OMG가 진행하여 ISO 승인된 표준으로는 UML, SysML, CORBA 등이 있다. 2001년에 시작된 DDS 표준화 작업도 잘 진행되어 지금에 와서는 OpenFMB, Adaptive AUTOSAR, MD PnP, GVA, NGVA, ROS 2와 같은 시스템들에서 DDS를 사용하며 산업 표준의 기반이 되고 있다. 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 마련**되었다고 생각한다.

2. 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 2에도 매우 적합하다고 볼 수 있다.

3. 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 하기 그림과 같이 DDS를 **RMW(ROS middleware)**으로 디자인되었으며 벤더 별로 각 RMW가 제작되었으며, 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.



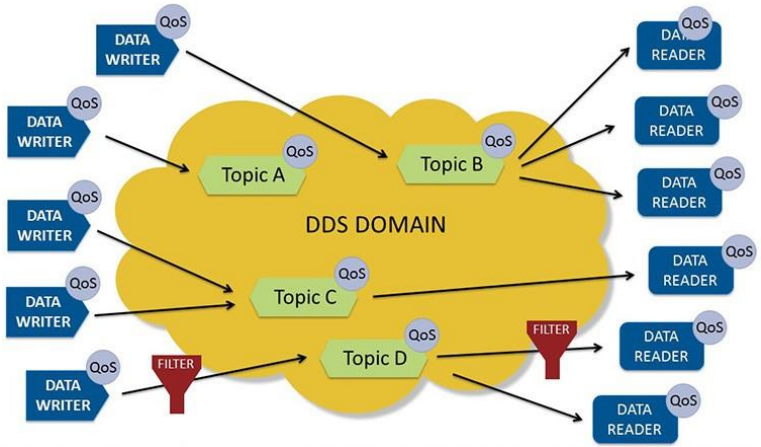
4. UDP 기반의 전송 방식

DDS 벤더 별로 **DDS Interoperability Wire Protocol (DDSI-RTPS)**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 돕고 있다. UDP 기반이라는 것이 ROS 1에서의 TCPROS가 TCP 기반이었던 것에 비해 매우 큰 변화인데 UDP의 멀티캐스트(multicast)는 브로드캐스트(broadcast)처럼 여러 목적지로 동시에 데이터를 보낼 수 있지만, 불특정 목적지가 아닌 특정된 도메인 그룹에 대해서만 데이터를 전송하게 된다. 참고로 ROS 2에서는 `**ROS_DOMAIN_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.

* 참고로 일부 RMW 기능에는 TCP 기반으로 구현되는 경우도 있다.

5. 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 방식**이라는 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.**



6. 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 실행했었어야 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.

7. 확장 가능한 아키텍처

OMG의 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 제공**되어 사용할 수 있으며 단일 표준 통신 계층에서 많은 복잡성을 흡수하여 **분산 시스템 개발을 더욱 단순화 시켜 편의성을 높였다.**

특히 ROS와 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 기능**이다.

8. 상호 운용성

ROS 2에서 통신 미들웨어로 사용하고 있는 DDS는 상호 운용성을 지원하고 있다. 즉, DDS의 표준 사양을 지키고 있는 벤더 제품을 사용한다면 A라는 회사의 제품을 사용하였다가도 B라는 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 벤더로는 10 곳이 있는데 이 중 ROS 2를 지원하는 업체는 ~~ADLink~~, Eclipse Foundation, Eprosima, Gurum Network, RTI로 총 5 곳이며 DDS 제품명으로는 ~~ADLINK의 OpenSplice~~, Eclipse Foundation의 **Cyclone DDS**, Eprosima의 **Fast DDS**, Gurum Network의 **Gurum DDS**, RTI의 **Connext DDS**가 있다. 이 중 Fast DDS와 Cyclone DDS는 오픈 소스를 지향하고 있기에 자유롭게 사용 가능하며 기술 지원을 개별적으로 받기 원한다면 상용 제품인 Connext DDS, Gurum DDS를 사용하면 된다.

9. 서비스 품질 (QoS)

ROS 2에서는 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 QoS를 사용할 수 있다.

DDS 사양상 설정 가능한 QoS 항목은 22가지인데 ROS 2에서는 현재 TCP처럼 데이터 손실을 방지함으로써 신뢰도를 우선시하거나 (**reliable**), UDP처럼 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 DDS는 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.

* 이 QoS 관련 부분은 실제 코딩에서 어떻게 설정하느냐가 매우 중요하기에 **reliability, History, Durability,**

10. 보안

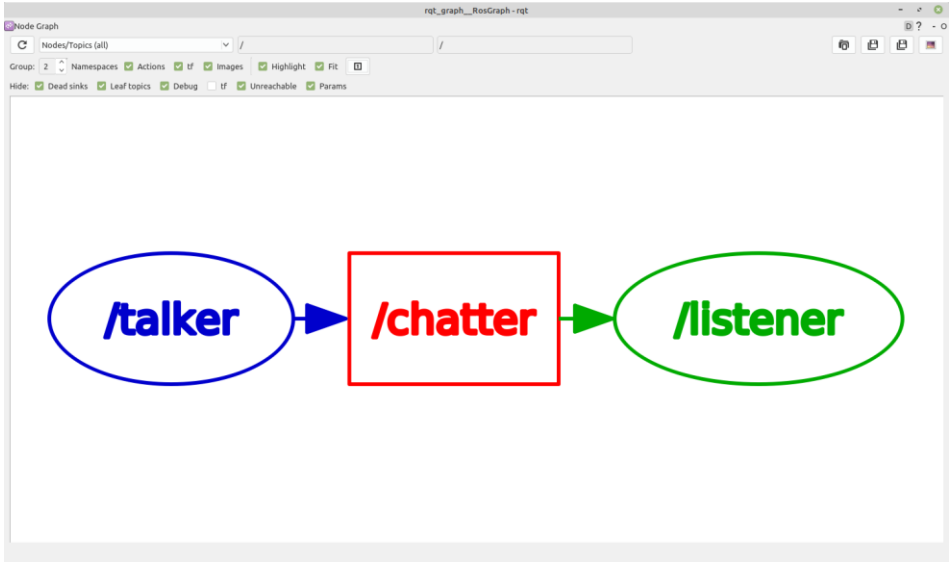
ROS 1의 가장 큰 구멍이었던 보안 부분은 ROS 2 개발에서 DDS으로 해결되었다. DDS의 사양에는 **DDS-Security**이라는 DDS 보안 사양을 ROS에 적용하여 보안에 대한 이슈를 통신단부터 해결하였다. 또한 ROS 커뮤니티에서는 **SROS 2(Secure Robot Operating System 2)**라는 툴을 개발하였고 보안 관련 RCL 서포트 및 보안 관련 프로그래밍에 익숙지 않은 로보틱스 개발자를 위해 보안을 위한 툴킷을 만들어 배포하고 있다. 이 부분에 대한 설명도 추후 이어지는 강좌에서 실습을 통해 더 자세히 알아보기로 하자.

DDS 사용 예시: 기본적인 퍼블리셔 노드와 서브스크라이브 노드 실행

```
$ ros2 run demo_nodes_cpp listener
[INFO]: I heard: [Hello World: 1]
[INFO]: I heard: [Hello World: 2]
[INFO]: I heard: [Hello World: 3]
[INFO]: I heard: [Hello World: 4]
[INFO]: I heard: [Hello World: 5]
```

```
$ ros2 run demo_nodes_cpp talker
[INFO]: Publishing: 'Hello World: 1'
[INFO]: Publishing: 'Hello World: 2'
[INFO]: Publishing: 'Hello World: 3'
[INFO]: Publishing: 'Hello World: 4'
[INFO]: Publishing: 'Hello World: 5'
```

```
$ rqt_graph
```



DDS 사용 예시: RMW 변경 방법

- RMW_IMPLEMENTATION 변수로 RMW 변경 가능
  - rmw_cyclonedds_cpp
  - rmw_fastrtps_cpp
  - rmw_connext_cpp
  - rmw_gurumdds_cpp

```
$ export RMW_IMPLEMENTATION=rmw_cyclonedds_cpp
$ ros2 run demo_nodes_cpp listener
[INFO]: I heard: [Hello World: 1]
[INFO]: I heard: [Hello World: 2]
[INFO]: I heard: [Hello World: 3]
(생략)
```

```
$ export RMW_IMPLEMENTATION=rmw_cyclonedds_cpp
$ ros2 run demo_nodes_cpp talker
[INFO]: Publishing: 'Hello World: 1'
[INFO]: Publishing: 'Hello World: 2'
[INFO]: Publishing: 'Hello World: 3'
(생략)
```

DDS 사용 예시: RMW의 상호 운용성 테스트

```
$ export RMW_IMPLEMENTATION=rmw_cyclonedds_cpp
$ ros2 run demo_nodes_cpp listener
[INFO]: I heard: [Hello World: 1]
[INFO]: I heard: [Hello World: 2]
[INFO]: I heard: [Hello World: 3]
(생략)
```

```
$ export RMW_IMPLEMENTATION=rmw_fastrtps_cpp
$ ros2 run demo_nodes_cpp talker
[INFO]: Publishing: 'Hello World: 1'
[INFO]: Publishing: 'Hello World: 2'
[INFO]: Publishing: 'Hello World: 3'
(생략)
```

DDS 사용 예시: Domain 변경 방법

```
$ export ROS_DOMAIN_ID=11
$ ros2 run demo_nodes_cpp talker
[INFO]: Publishing: 'Hello World: 1'
[INFO]: Publishing: 'Hello World: 2'
[INFO]: Publishing: 'Hello World: 3'
(생략)
```

```
$ export ROS_DOMAIN_ID=12
$ ros2 run demo_nodes_cpp listener
(서로 다른 도메인을 사용하고 있기에 아무런 반응이 없을 것이다.)
```

```
$ export ROS_DOMAIN_ID=11
$ ros2 run demo_nodes_cpp listener
[INFO]: I heard: [Hello World: 13]
[INFO]: I heard: [Hello World: 14]
[INFO]: I heard: [Hello World: 15]
(생략)
```

DDS 사용 예시: QoS 테스트 (Reliability = RELIABLE vs BEST_EFFORT)

```
$ sudo tc qdisc add dev lo root netem loss 10%
```

```
$ ros2 run demo_nodes_cpp talker
[INFO]: Publishing: 'Hello World: 1'
[INFO]: Publishing: 'Hello World: 2'
[INFO]: Publishing: 'Hello World: 3'
[INFO]: Publishing: 'Hello World: 4'
[INFO]: Publishing: 'Hello World: 5'
[INFO]: Publishing: 'Hello World: 6'
[INFO]: Publishing: 'Hello World: 7'
[INFO]: Publishing: 'Hello World: 8'
[INFO]: Publishing: 'Hello World: 9'
[INFO]: Publishing: 'Hello World: 10'
[INFO]: Publishing: 'Hello World: 11'
[INFO]: Publishing: 'Hello World: 12'
[INFO]: Publishing: 'Hello World: 13'
[INFO]: Publishing: 'Hello World: 14'
[INFO]: Publishing: 'Hello World: 15'
```

```
$ ros2 run demo_nodes_cpp listener
[INFO]: I heard: [Hello World: 1]
[INFO]: I heard: [Hello World: 2]
[INFO]: I heard: [Hello World: 3]
[INFO]: I heard: [Hello World: 4]
[INFO]: I heard: [Hello World: 5]
[INFO]: I heard: [Hello World: 6]
[INFO]: I heard: [Hello World: 7]
[INFO]: I heard: [Hello World: 8]
[INFO]: I heard: [Hello World: 9]
[INFO]: I heard: [Hello World: 10]
[INFO]: I heard: [Hello World: 11]
[INFO]: I heard: [Hello World: 12]
[INFO]: I heard: [Hello World: 13]
[INFO]: I heard: [Hello World: 14]
[INFO]: I heard: [Hello World: 15]
```

DDS 사용 예시: QoS 테스트 (Reliability = RELIABLE vs BEST_EFFORT)

```
$ ros2 run demo_nodes_cpp talker
[INFO]: Publishing: 'Hello World: 1'
[INFO]: Publishing: 'Hello World: 2'
[INFO]: Publishing: 'Hello World: 3'
[INFO]: Publishing: 'Hello World: 4'
[INFO]: Publishing: 'Hello World: 5'
[INFO]: Publishing: 'Hello World: 6'
[INFO]: Publishing: 'Hello World: 7'
[INFO]: Publishing: 'Hello World: 8'
[INFO]: Publishing: 'Hello World: 9'
[INFO]: Publishing: 'Hello World: 10'
[INFO]: Publishing: 'Hello World: 11'
[INFO]: Publishing: 'Hello World: 12'
[INFO]: Publishing: 'Hello World: 13'
[INFO]: Publishing: 'Hello World: 14'
[INFO]: Publishing: 'Hello World: 15'
```

```
$ ros2 run demo_nodes_cpp listener_best_effort
[INFO]: I heard: [Hello World: 1]
[INFO]: I heard: [Hello World: 3]
[INFO]: I heard: [Hello World: 4]
[INFO]: I heard: [Hello World: 5]
[INFO]: I heard: [Hello World: 6]
[INFO]: I heard: [Hello World: 7]
[INFO]: I heard: [Hello World: 8]
[INFO]: I heard: [Hello World: 10]
[INFO]: I heard: [Hello World: 11]
[INFO]: I heard: [Hello World: 12]
[INFO]: I heard: [Hello World: 13]
[INFO]: I heard: [Hello World: 14]
[INFO]: I heard: [Hello World: 15]
```

```
$ sudo tc qdisc delete dev lo root netem loss 10%
```

ROS 2 패키지 설치와 노드 실행

Turtlesim 패키지 설치와 노드

```
$ sudo apt update
$ sudo apt install ros-foxy-turtlesim
```

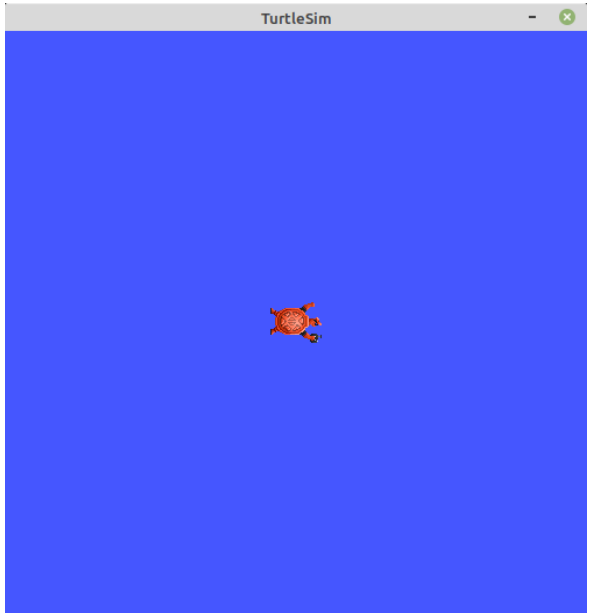
```
$ ros2 pkg list
ackermann_msgs
ackermann_steering_controller
action_msgs
action_tutorials_cpp
action_tutorials_interfaces
action_tutorials_py
actionlib_msgs
(생략)
```

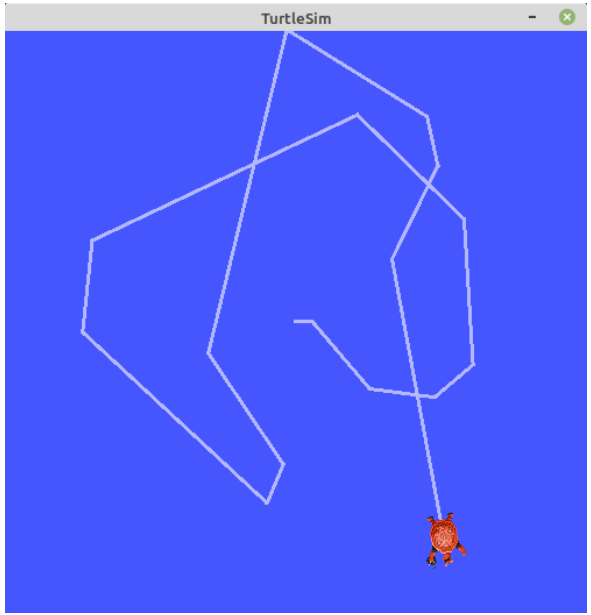
```
$ ros2 pkg executables turtlesim
turtlesim draw_square
turtlesim mimic
turtlesim turtle_teleop_key
turtlesim turtlesim_node
```

패키지의 노드 실행

```
$ ros2 run turtlesim turtlesim_node
```

```
$ ros2 run turtlesim turtle_teleop_key
```





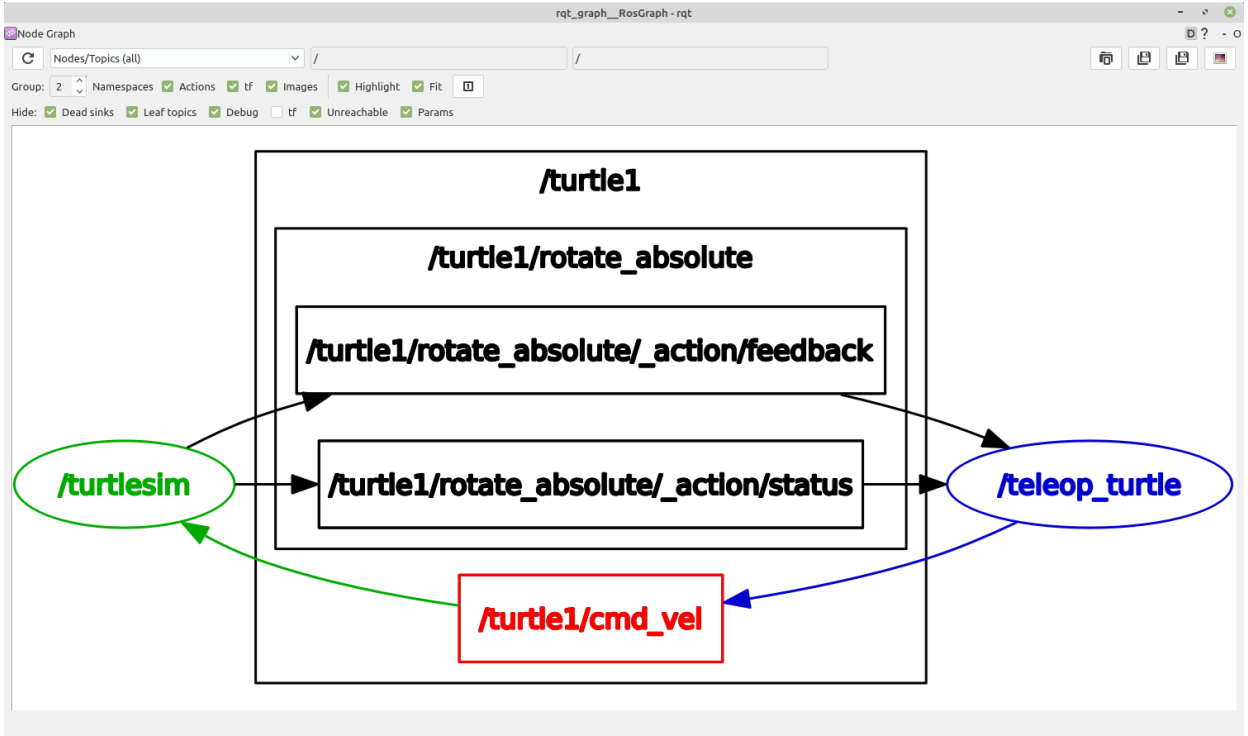
노드, 토픽, 서비스, 액션의 조회

```
$ ros2 node list
/turtlesim
/teleop_turtle

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

$ ros2 action list
/turtle1/rotate_absolute
```

```
$ 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
```

rqt_graph 로 보는 노드와 토픽의 그래프 뷰

```
$ rqt_graph
```

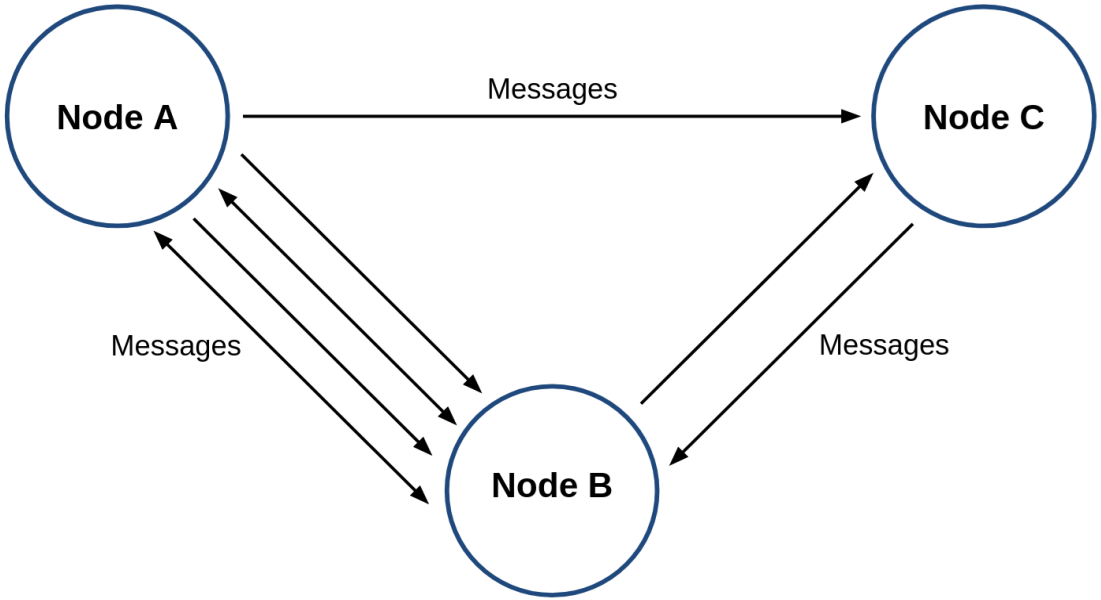


ROBOTIS

ROS 2 노드와 메시지 통신

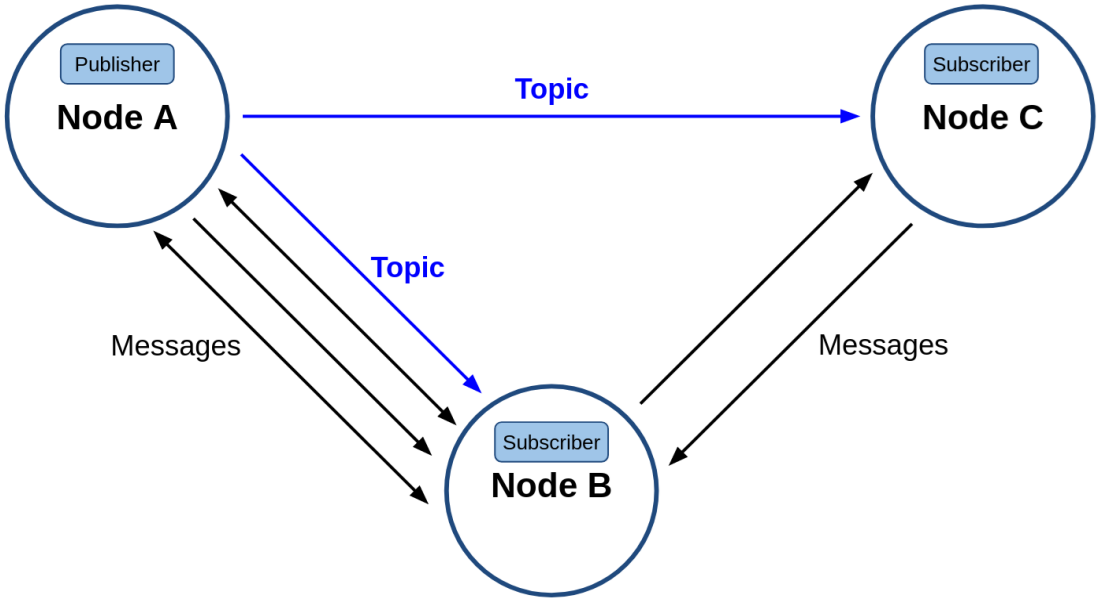
노드(node)와 메시지 통신(message communication)

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.



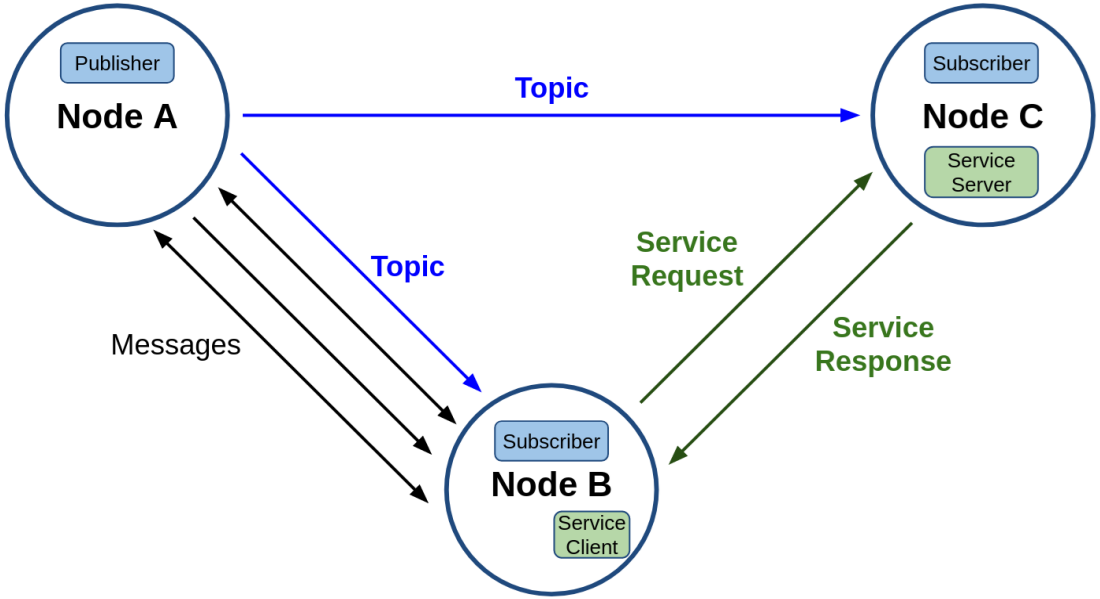
노드(node)와 메시지 통신(message communication)

토픽(topic)은 아래 그림의 `Node A - Node B`, `Node A - Node C`처럼 **비동기식 단방향 메시지 송수신 방식**으로 **msg 메시지 형태**의 **메시지를 발간하는 Publisher**와 **메시지를 구독하는 Subscriber** 간의 통신이라고 볼 수 있다. 이는 **1:N, N:1, N:N 통신도 가능**하며 ROS 메시지 통신에서 가장 널리 사용되는 통신 방법이다.



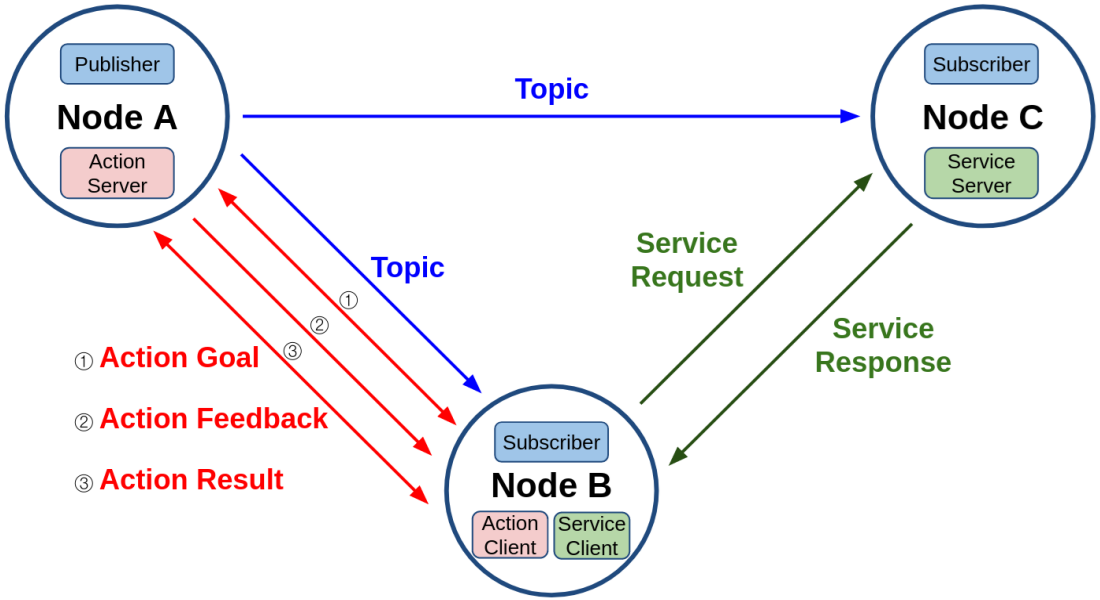
노드(node)와 메시지 통신(message communication)

서비스(Service)는 아래 그림의 `Node B - Node C`처럼 **동기식 양방향 메시지 송수신 방식**으로 **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 결과 값을 전달하는 서버 단과의 통신이라고 볼 수 있다. 서비스 요청 및 응답(Request/Response) 또한 위에서 언급한 msg 메시지의 변형으로 **srv 메시지**라고 한다.



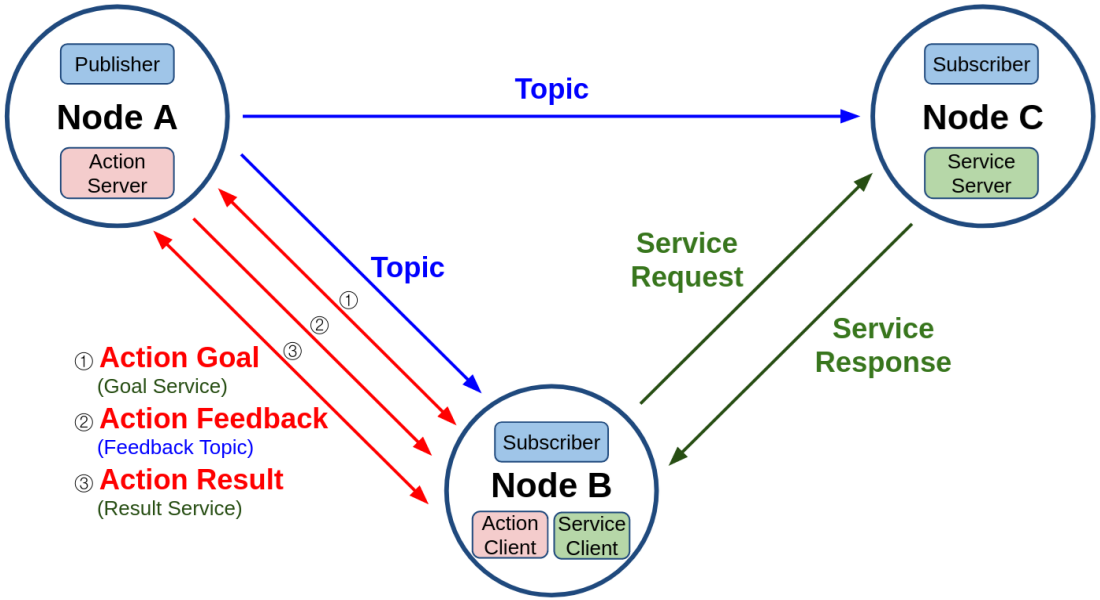
노드(node)와 메시지 통신(message communication)

액션(Action)은 아래 그림의 `Node A - Node B`처럼 **비동기식+동기식 양방향 메시지 송수신 방식**으로 **액션 목표 Goal를 지정하는 Action client**과 액션 목표를 받아 특정 태스크를 수행하면서 중간 결과 값에 해당되는 **액션 피드백(Feedback)과 최종 결과 값에 해당되는 액션 결과(Result)를 전송하는 Action server** 간의 통신이라고 볼 수 있다.



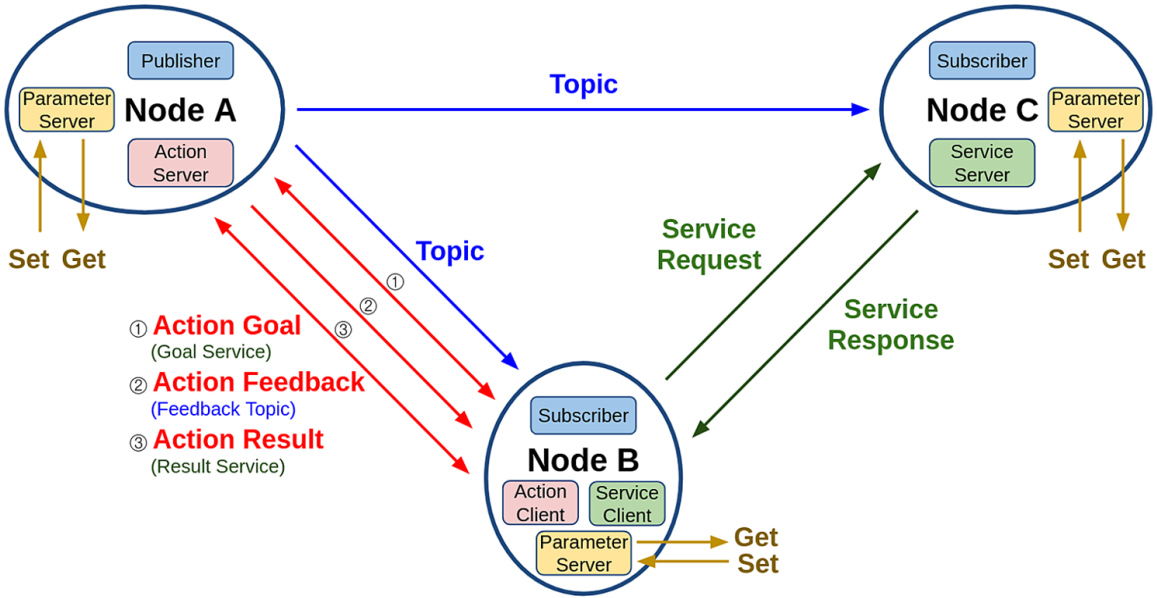
노드(node)와 메시지 통신(message communication)

액션의 구현 방식을 더 자세히 살펴보면 아래 그림과 같이 **토픽(topic)과 서비스(service)의 혼합**이라고 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 메시지**라고 한다.



노드(node)와 메시지 통신(message communication)

파라미터(Parameter)는 아래 그림의 각 노드에 파라미터 관련 **Parameter server**를 실행시켜 **외부의 Parameter client 간의 통신으로 파라미터를 변경하는 것으로 서비스와 동일**하다고 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.

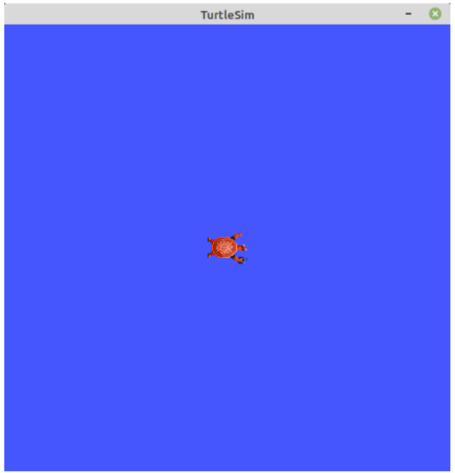


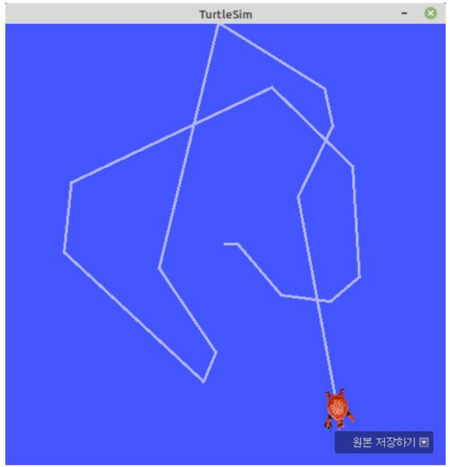
메시지 통신 테스트

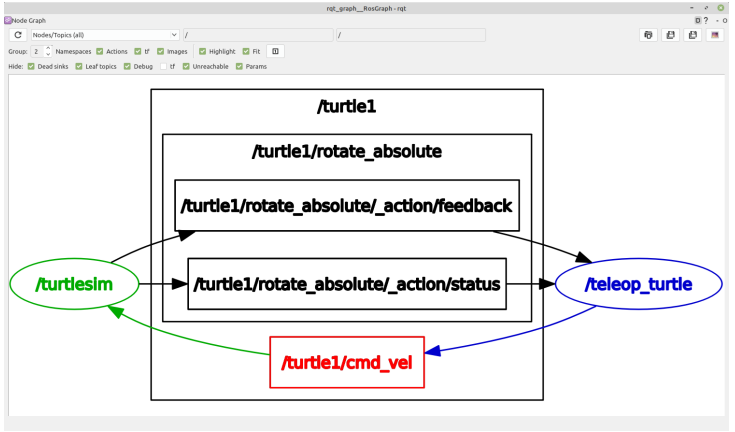
```
$ ros2 run turtlesim turtlesim_node
```

```
$ ros2 run turtlesim turtle_teleop_key
```

```
$ rqt_graph
```





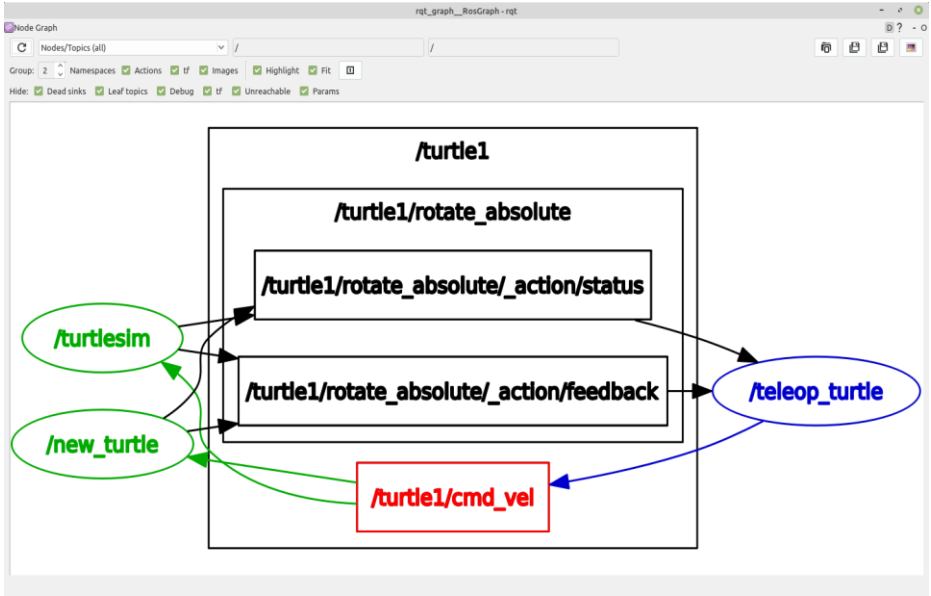


노드 목록 (ros2 node list)

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

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

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



노드 정보 (ros2 node info)

```
$ 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:
```

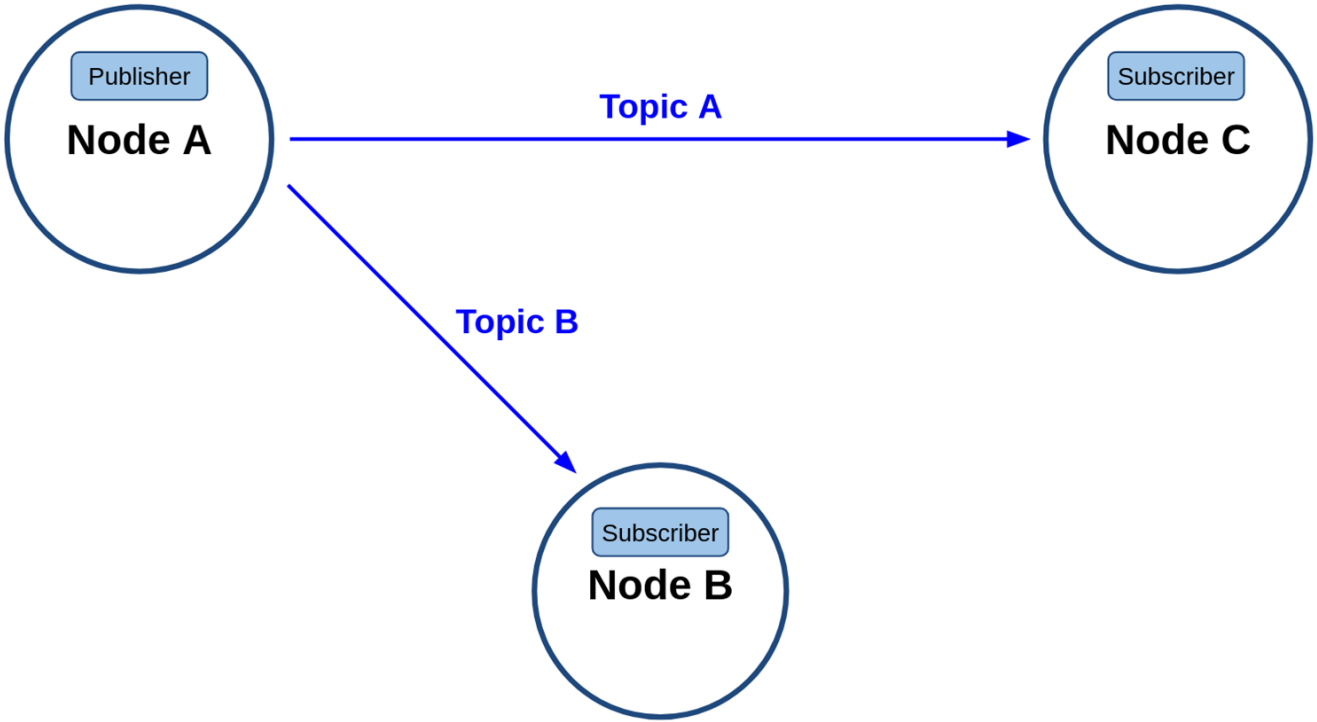
노드 정보 (ros2 node info)

```
$ 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
```

ROBOTIS

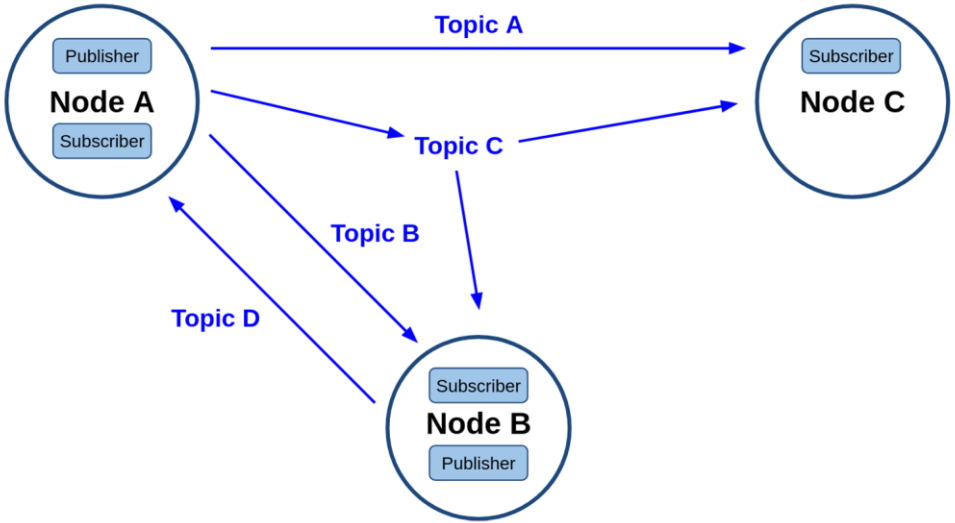
ROS 2 토픽 (Topic)

토픽 (Topic)


Publisher
Node A
Topic A
Subscriber
Node C
Topic B
Subscriber
Node B

토픽 (Topic)

`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.



토픽 목록 확인 (ros2 node info)

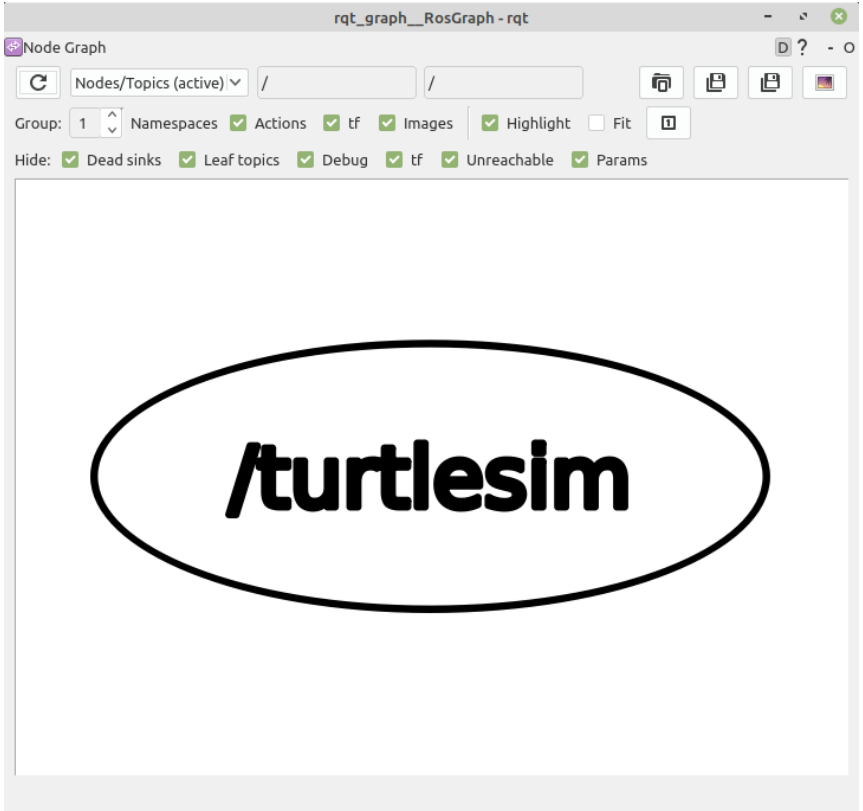
```
$ ros2 run turtlesim turtlesim_node
```

```
$ ros2 node info /turtlesim
/turtlesim
  Subscribers:
    /turtle1/cmd_vel: geometry_msgs/msg/Twist
    (생략)
  Publishers:
    /turtle1/color_sensor: turtlesim/msg/Color
    /turtle1/pose: turtlesim/msg/Pose
    (생략)
  Services:
    /clear: std_srvs/srv/Empty
    /kill: turtlesim/srv/Kill
    (생략)
```

토픽 목록 확인 (ros2 topic list)

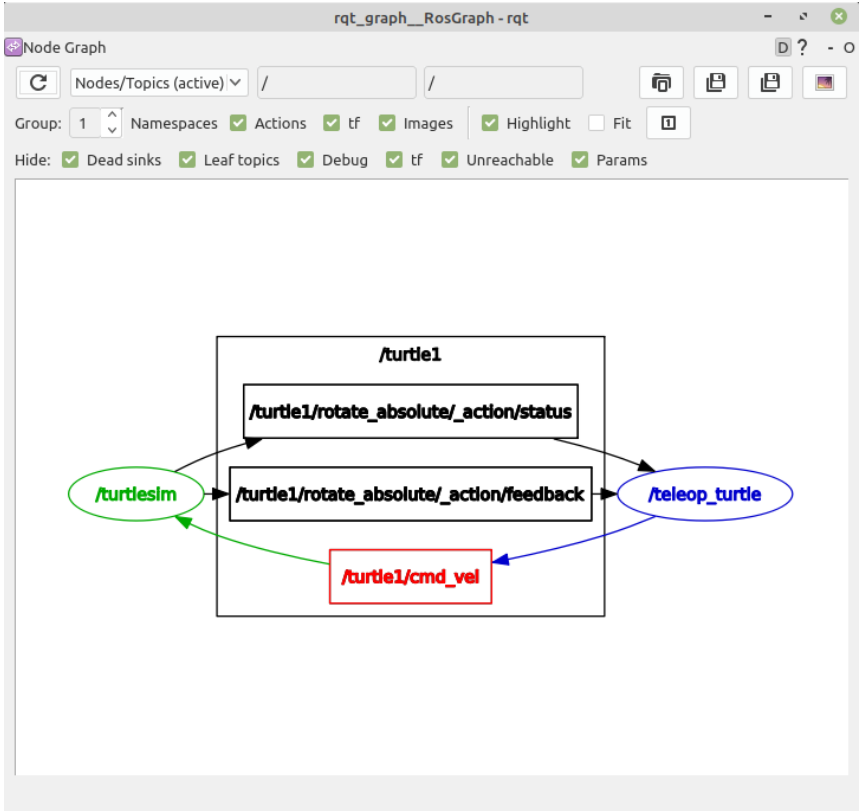
```
$ ros2 topic list -t
/parameter_events [rcl_interfaces/msg/ParameterEvent]
/rosout [rcl_interfaces/msg/Log]
/turtle1/cmd_vel [geometry_msgs/msg/Twist]
/turtle1/color_sensor [turtlesim/msg/Color]
/turtle1/pose [turtlesim/msg/Pose]
```

```
$ rqt_graph
```

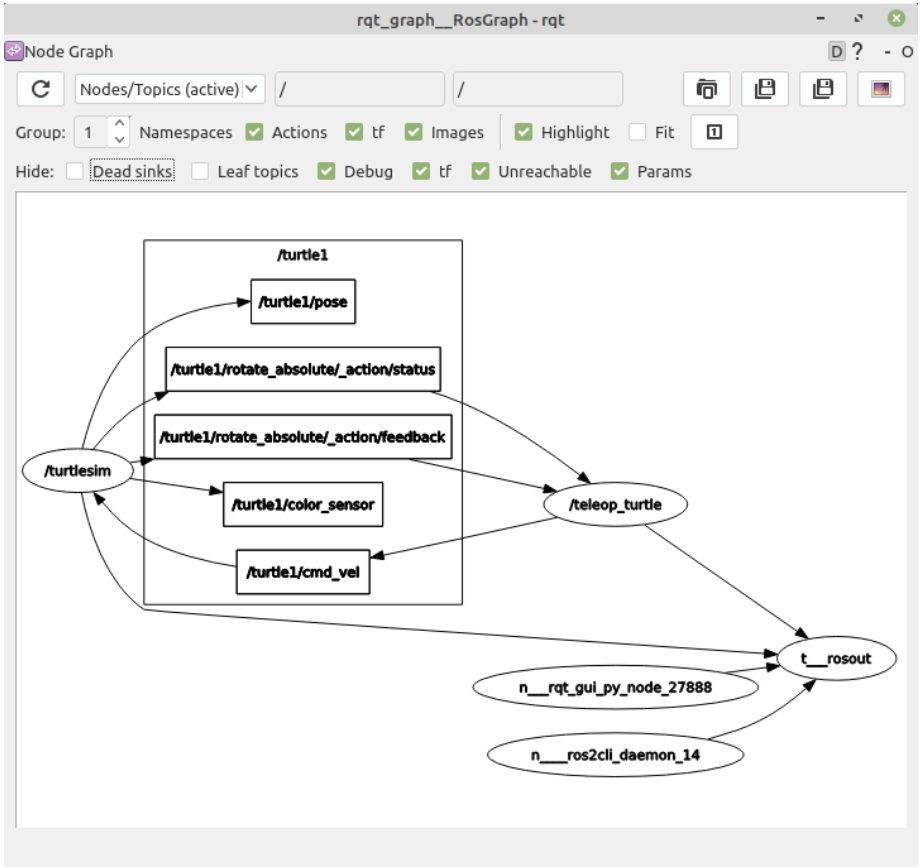


토픽 목록 확인 (rqt)

```
$ ros2 run turtlesim turtle_teleop_key
```



토픽 목록 확인 (rqt)


rqt_graph__RosGraph - rqt
Node Graph
Nodes/Topics (active)
/
/
Group: 1
Namespaces
Actions
tf
Images
Highlight
Fit
Hide:
Dead sinks
Leaf topics
Debug
tf
Unreachable
Params
/turtle1
/turtle1/pose
/turtle1/rotate_absolute/_action/status
/turtle1/rotate_absolute/_action/feedback
/turtle1/color_sensor
/turtle1/cmd_vel
/turtlesim
/teleop_turtle
t__rosout
n__rqt_gui_py_node_27888
n__ros2cli_daemon_14

토픽 정보 확인 (ros2 topic info)

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

토픽 내용 확인 (ros2 topic echo)

```
$ 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
```

토픽 대역폭 확인 (ros2 topic bw)

```
$ 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)
```

토픽 주기 확인 (ros2 topic hz)

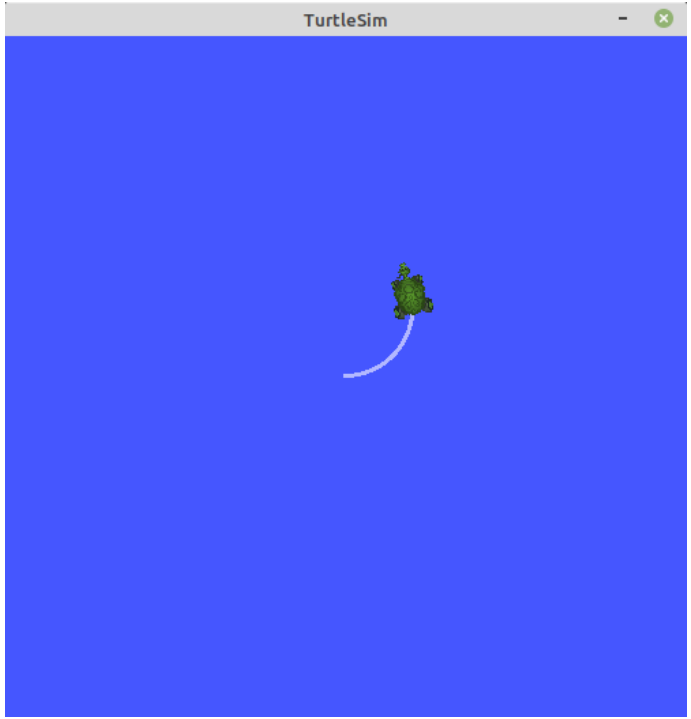
```
$ ros2 topic hz /turtle1/cmd_vel
average rate: 33.212
min: 0.029s max: 0.089s std dev: 0.00126s window: 2483
(생략)
```

토픽 지연 시간 확인 (ros2 topic delay)

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

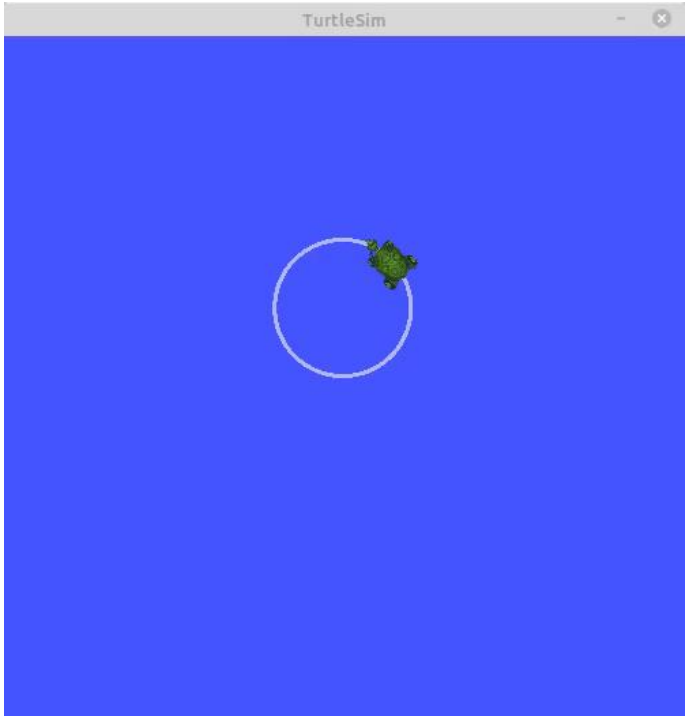
토픽 발행 (ros2 topic pub)

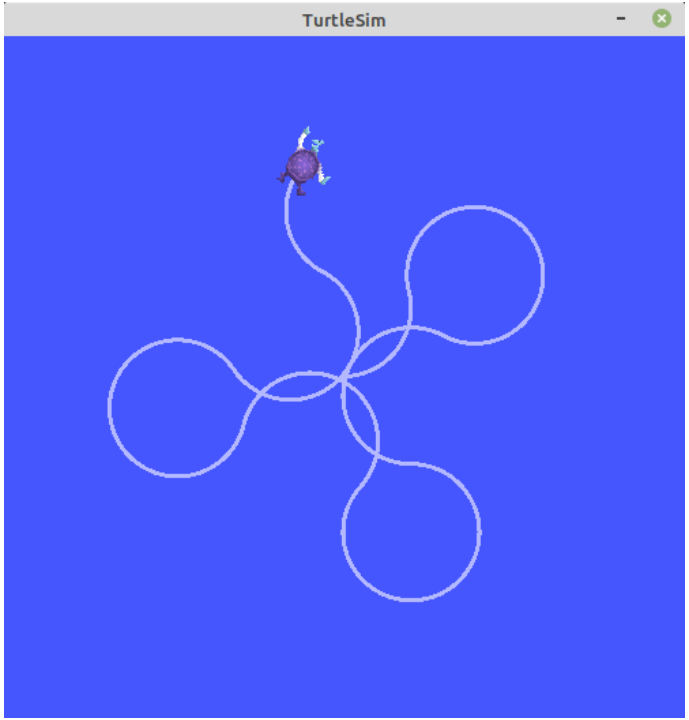
```
$ ros2 topic pub --once /turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 1.8}}"
```



토픽 발행 (ros2 topic pub)

```
$ 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}}"
```





bag 기록 (ros2 bag record)

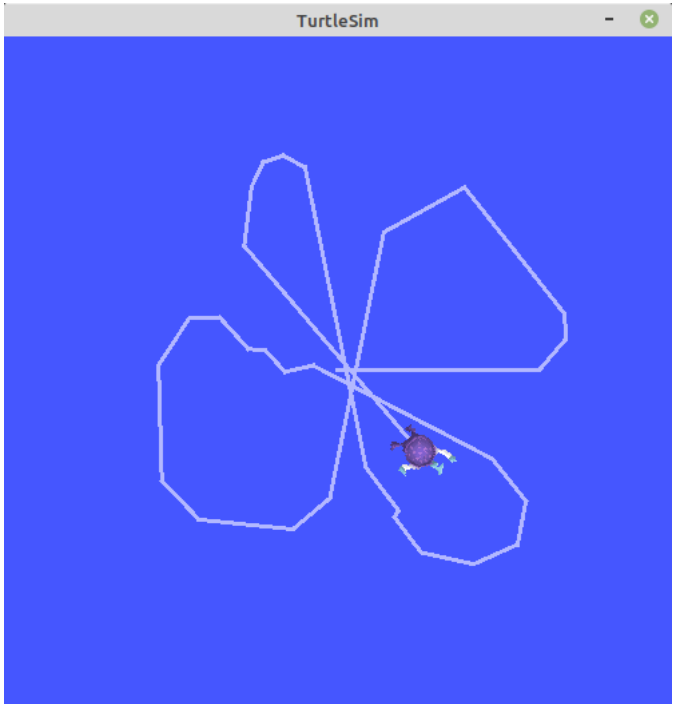
```
$ ros2 bag record /turtle1/cmd_vel
[INFO]: Opened database 'rosbag2_2024_06_20-20_21_39'.
[INFO]: Listening for topics...
[INFO]: Subscribed to topic '/turtle1/cmd_vel'
```

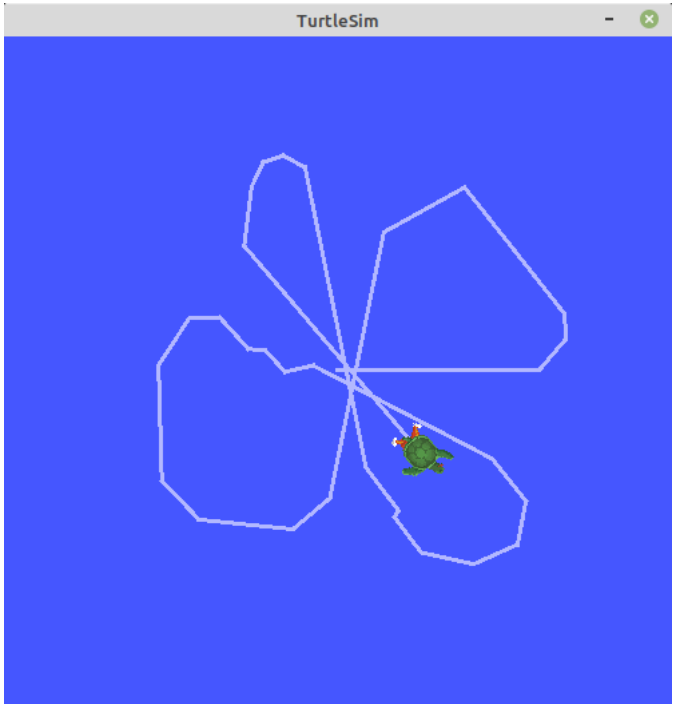
bag 정보 (ros2 bag info)

```
$ 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 (ros2 bag play)

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





93 ROS 인터페이스 (interface) 이란?

ROS의 노드 간에 데이터를 주고받을 때에는 **토픽, 서비스, 액션이 사용되는데 이 때 사용되는 데이터의 형태를 ROS 인터페이스(interface)** [7]라고 한다. ROS 인터페이스에는 ROS 2에 새롭게 추가된 **IDL**(interface definition language)과 ROS 1부터 ROS 2까지 널리 사용 중인 **msg**, **srv**, **action** 이 있다. 토픽, 서비스, 액션은 각각 msg, srv, action interface를 사용하고 있으며 정수, 부동 소수점, 불리언과 같은 단순 자료형을 기본으로 하여 메시지 안에 메시지를 품고 있는 간단한 데이터 구조 및 메시지들이 나열된 배열과 같은 구조도 사용할 수 있다.

[**단순 자료형**]
- 예) 정수(integer), 부동 소수점(floating point), 불(boolean)
- https://github.com/ros2/common_interfaces/tree/foxy/std_msgs

[**메시지 안에 메시지를 품고 있는 간단한 데이터 구조**]
- 예) geometry_msgs/msgs/Twist의 `Vector3 linear`
- https://github.com/ros2/common_interfaces/blob/foxy/geometry_msgs/msg/Twist.msg

[**메시지들이 나열된 배열과 같은 구조**]
- 예) sensor_msgs/msgs/LaserScan 의 `float32[] ranges`
- https://github.com/ros2/common_interfaces/blob/foxy/sensor_msgs/msg/LaserScan.msg

메시지 인터페이스 (message interface, msg)

[teleop_turtle]

Publisher
메시지 발행

[/turtle1/cmd_vel]

geometry_msgs/msg/Twist
메시지

[turtlesim]

Subscriber
메시지 구독

[geometry_msgs/msgs/Twist]

Vector3 linear
Vector3 angular

[geometry_msgs/msgs/Vector3]

float64 x
float64 y
float64 z

[geometry_msgs/msgs/Vector3]

float64 x
float64 y
float64 z

95 메시지 인터페이스 (message interface, msg)

토픽은 고유의 인터페이스를 가지고 있는데 이를 **메시지 인터페이스**라 부르며, 파일로는 **msg 파일**을 가르킨다.

예를 들어, 위 예제에서 /turtle1/cmd_vel 토픽은 geometry_msgs/msgs/Twist 형태이다. 이름이 좀 긴데 풀어서 설명하면 기하학 관련 메시지를 모아둔 geometry_msgs 패키지의 msgs 분류의 Twist 데이터 형태라는 것이다.

Twist 데이터 형태를 자세히 보면 Vector3 linear과 Vector3 angular 이라고 되어 있다. 이는 메시지 안에 메시지를 품고 있는 것으로 Vector3 형태에 linear 이라는 이름의 메시지와 Vector3 형태에 angular 이라는 이름의 메시지, 즉 2개의 메시지가 있다는 것이며 Vector3는 다시 float64 형태에 x, y, z 값이 존재한다.

다시 말해 **geometry_msgs/msgs/Twist 메시지 형태는 float64 자료형의 linear.x, linear.y, linear.z, angular.x, angular.y, angular.z 라는 이름의 메시지인 것이다. 이를 통해 병진 속도 3개, 회전 속도 3개를 표현할 수 있게 된다.**

ros2 interface 명령어

```
$ ros2 interface show geometry_msgs/msg/Twist
Vector3 linear
Vector3 angular

$ ros2 interface show geometry_msgs/msg/Vector3
float64 x
float64 y
float64 z
```

ros2 interface 명령어

```
$ ros2 interface list
Messages:
   action_msgs/msg/GoalInfo
   action_msgs/msg/GoalStatus
   action_msgs/msg/GoalStatusArray
(생략)

Services:
   action_msgs/srv/CancelGoal
   composition_interfaces/srv/ListNodes
(생략)

Actions:
   action_tutorials_interfaces/action/Fibonacci
   example_interfaces/action/Fibonacc
(생략)
```

ros2 interface 명령어

```
$ ros2 interface packages
action_msgs
action_tutorials_interfaces
actionlib_msgs
builtin_interfaces
(생략)
```

ros2 interface 명령어

```
$ ros2 interface package turtlesim
turtlesim/srv/Spawn
turtlesim/msg/Color
turtlesim/msg/Pose
turtlesim/srv/TeleportAbsolute
turtlesim/srv/Kill
turtlesim/action/RotateAbsolute
turtlesim/srv/SetPen
turtlesim/srv/TeleportRelative
```

ros2 interface 명령어

```
$ ros2 interface proto geometry_msgs/msg/Twist
"linear:
  x: 0.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
"
```

ROBOTIS

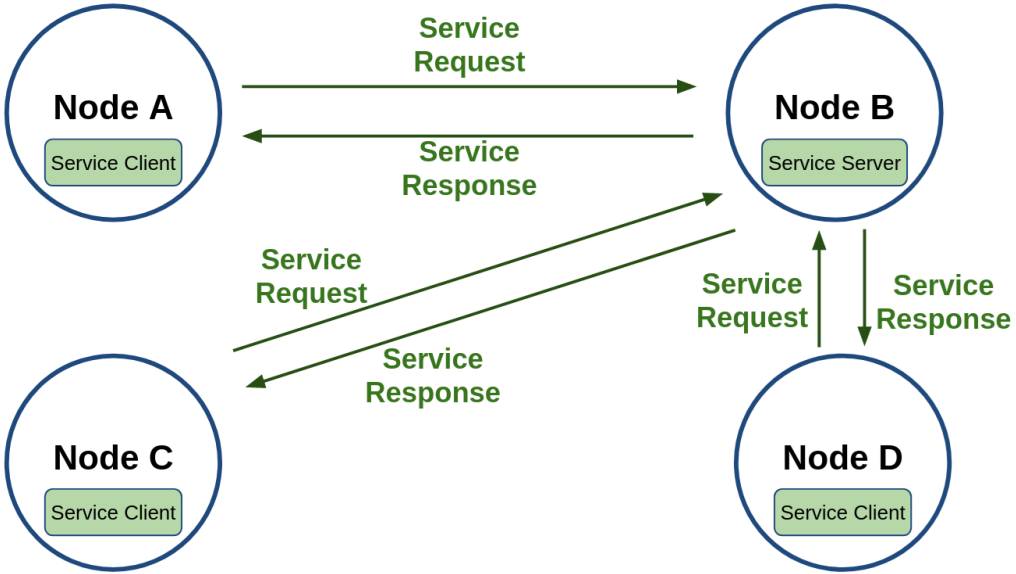
ROS 2 서비스 (Service)

서비스 (Service)


Service
Request
Node A
Service Client
Node B
Service Server
Service
Response

103 서비스 (Service)

서비스는 다음 그림과 같이 동일 서비스에 대해 복수의 클라이언트를 가질 수 있도록 설계되었다. 단, 서비스 응답은 서비스 요청이 있었던 서비스 클라이언트에 대해서만 응답을 하는 형태로 그림의 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.



서비스 목록 확인 (ros2 service list)

```
$ ros2 run turtlesim turtlesim_node
```

```
$ 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
```

서비스 형태 확인 (ros2 service type)

```
$ ros2 service type /clear
std_srvs/srv/Empty

$ ros2 service type /kill
turtlesim/srv/Kill

$ ros2 service type /spawn
turtlesim/srv/Spawn
```

```
$ 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)
```

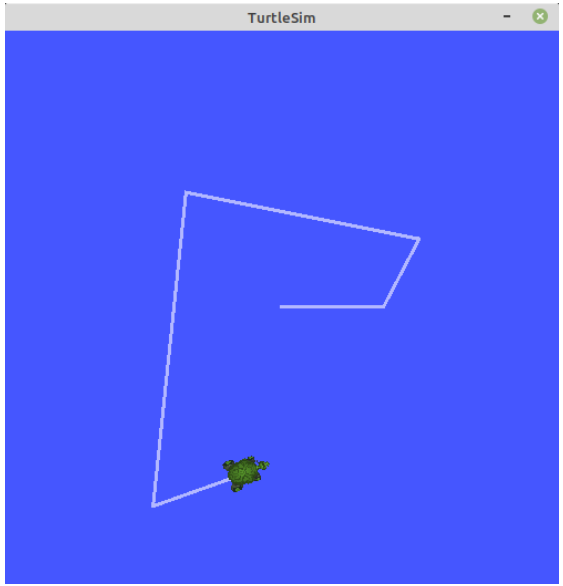
서비스 찾기 (ros2 service find)

```
$ ros2 service find std_srvs/srv/Empty
/clear
/reset

$ ros2 service find turtlesim/srv/Kill
/kill
```

서비스 요청 (ros2 service call)

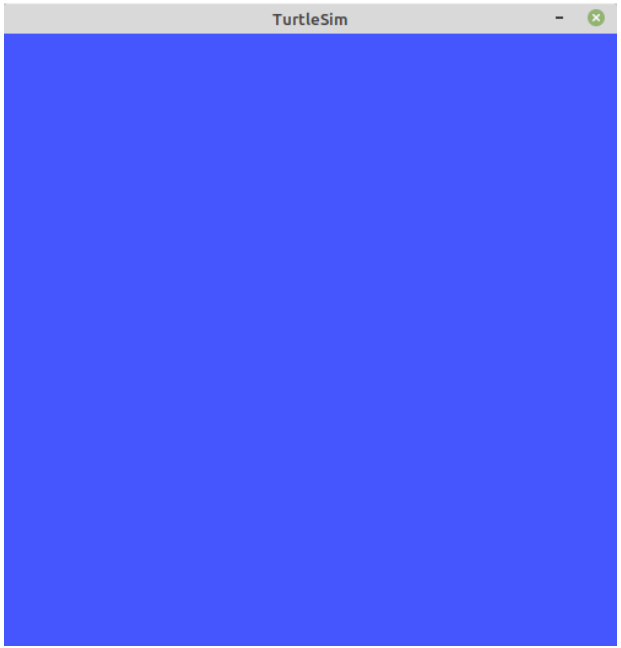
```
$ ros2 run turtlesim turtle_teleop_key
```

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

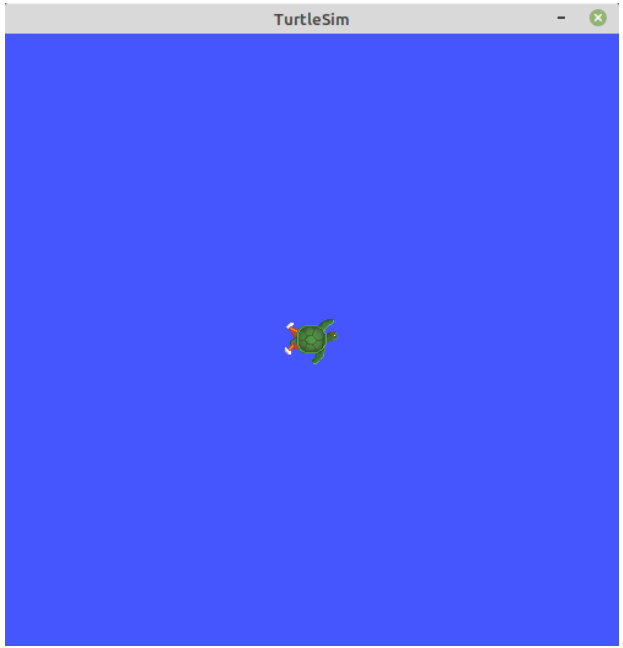


서비스 요청 (ros2 service call)

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

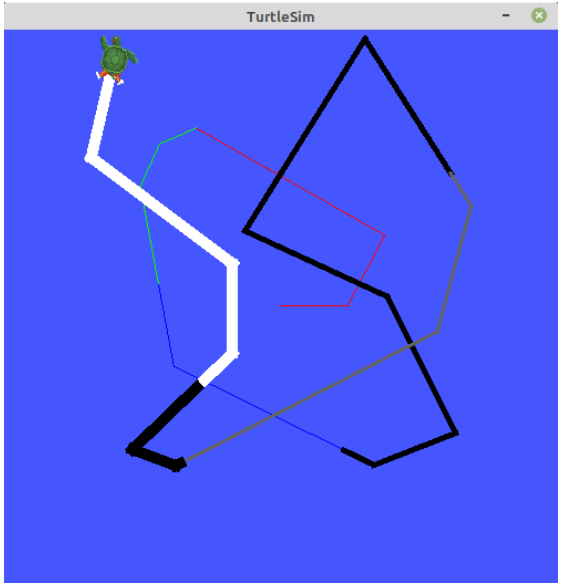



```
$ ros2 service call /reset std_srvs/srv/Empty
requester: making request: std_srvs.srv.Empty_Request()
response:
std_srvs.srv.Empty_Response()
```



서비스 요청 (ros2 service call)

```
$ 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()
```



서비스 요청 (ros2 service call)

```
$ 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')
```

서비스 요청 (ros2 service call)


```
$ 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
/turtle2/cmd_vel
/turtle2/pose
/turtle3/cmd_vel
/turtle3/color_sensor
/turtle4/cmd_vel
/turtle4/color_sensor
/turtle4/pose
```

112 서비스 인터페이스 (service interface, srv)

서비스 또한 토픽과 마찬가지로 별도의 인터페이스를 가지고 있는데 이를 **서비스 인터페이스**라 부르며, 파일로는 **srv 파일**을 가르킨다. 서비스 인터페이스는 메시지 인터페이스의 확장형이라고 볼 수 있는데 위에서 서비스 요청때 실습으로 사용하였던 /spawn 서비스를 예를 들어 설명하겠다.

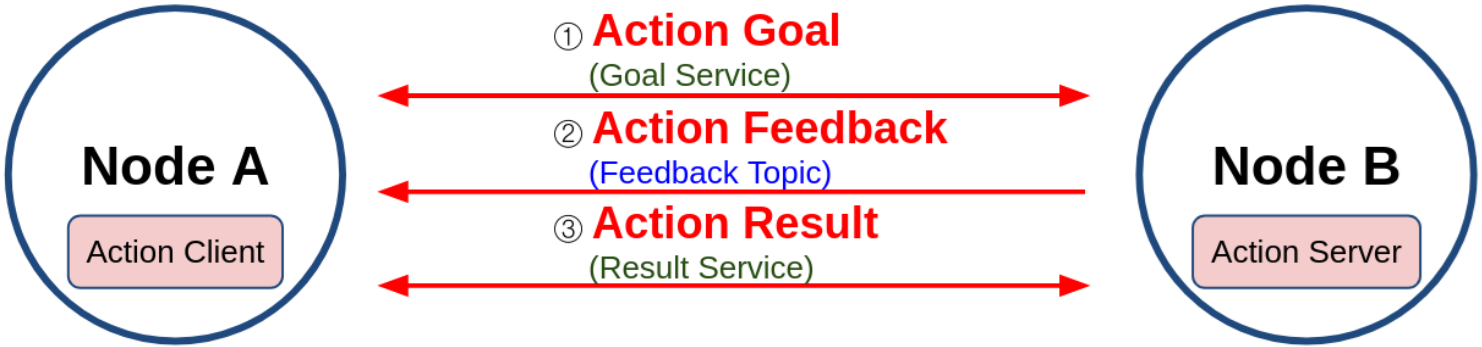
/spawn 서비스에 사용된 Spawn.srv 인터페이스를 알아보기 위해서는 원본 파일을 참고해도 되고, `**ros2 interface show**` 명령어를 이용하여 확인할 수 있다. 이 명령어를 이용하면 아래의 결과 값과 같이 `turtlesim/srv/Spawn.srv`은 float32 형태의 x, y, theta 그리고 string 형태로 name 이라고 두개의 데이터가 있음을 알 수 있다. 여기서 특이한 것은 `---` 이다. 이는 구분자라 부른다. 서비스 인터페이스는 메시지 인터페이스와는 달리 **서비스 요청 및 응답(Request/Response) 형태**로 구분되는데 **요청(Request)과 응답(Response)을 나누어 사용하기 위해서 `---`를 구분자로 사용**하게 된다. 즉 x, y, theta, name 은 서비스 요청에 해당되고 서비스 클라이언트 단에서 서비스 서버단에 전송하는 값이 된다. 그리고 서비스 서버단은 지정된 서비스를 수행하고 name 데이터를 서비스 클라이언트단에 전송하게 된다.

```
$ ros2 interface show turtlesim/srv/Spawn.srv
float32 x
float32 y
float32 theta
string name
---
string name
```

ROBOTIS

ROS 2 액션 (Action)

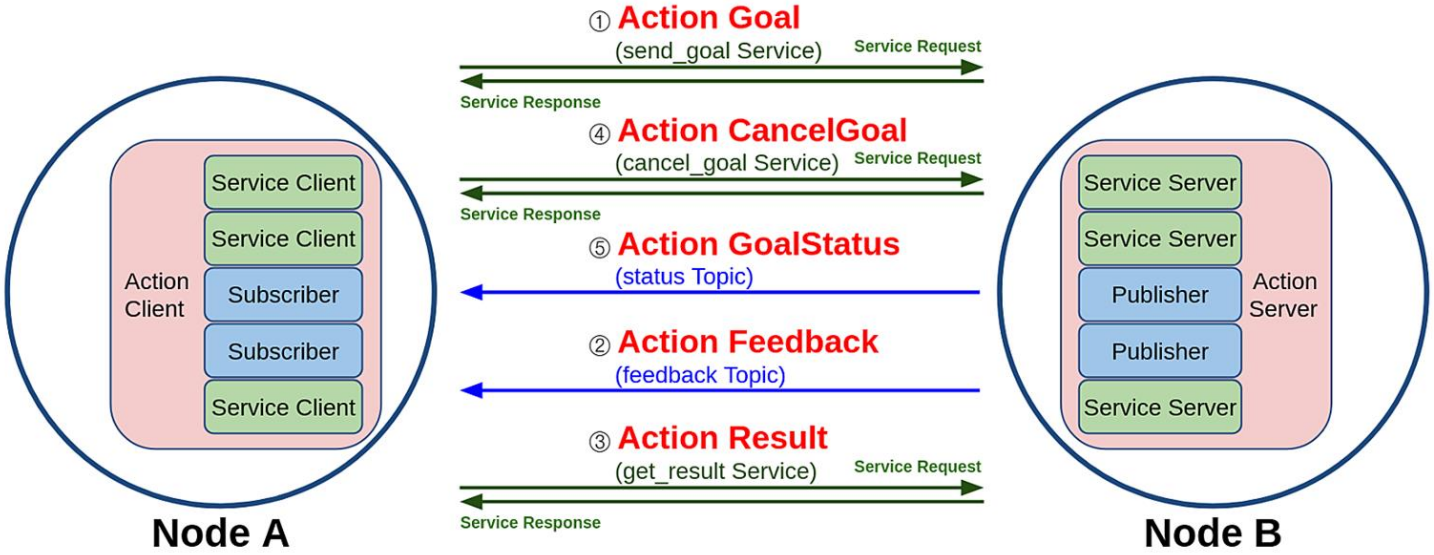
액션 (Action)


Node A
Action Client
① Action Goal
(Goal Service)
② Action Feedback
(Feedback Topic)
③ Action Result
(Result Service)
Node B
Action Server

액션 (Action)

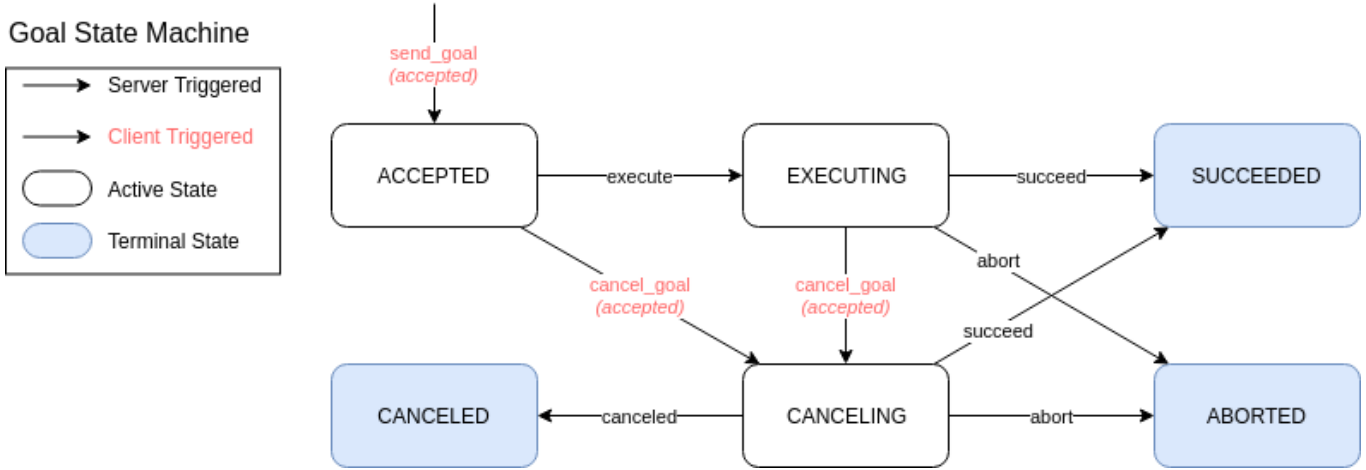
액션의 구현 방식을 더 자세히 살펴보면 다음 그림과 같이 토픽과 서비스의 혼합이라고 볼 수 있는데 ROS 1이 토픽만을 사용하였다면 ROS 2에서는 액션 목표, 액션 결과, 액션 피드백은 토픽과 서비스가 혼합되어 있다.

즉, 다음 그림과 같이 **Action Client는 Service Client 3개와 Topic Subscriber 2개로 구성**되어있으며, **Action Server는 Service Server 3개와 Topic Publisher 2개로 구성**된다. **액션 목표/피드백/결과(goal/feedback/result) 데이터**는 msg 및 srv 인터페이스의 변형으로 **action 인터페이스**라고 한다.



액션 (Action)

ROS 1에서의 액션은 목표, 피드백, 결과 값을 토픽으로만 주고 받았는데 **ROS 2에서는 토픽과 서비스 방식을 혼합하여 사용**하였다. 그 이유로 토픽으로만 액션을 구성하였을 때 토픽의 특징인 비동기식 방식을 사용하게 되어 ROS 2 액션에서 새롭게 선보이는 **목표 전달(send_goal), 목표 취소 (cancel_goal), 결과 받기 (get_result)를 동기식인 서비스를 사용하기 위해서**이다. 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 프로스세를 쫓는 것**이다. 여기서 말하는 상태머신은 **Goal State Machine**으로 다음 그림과 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.



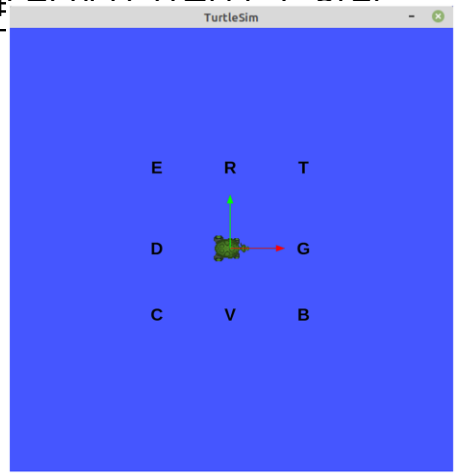
액션 서버 및 클라이언트

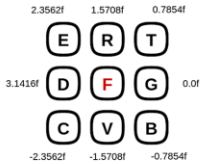
```
$ ros2 run turtlesim turtlesim_node
```

```
$ 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.
```

액션 서버 및 클라이언트

지금까지는 teleop_turtle 노드가 실행된 터미널 창에서 ← ↑ ↓ → 키와 같이 화살표 키를 눌러 turtlesim의 거북이를 움직여봤는데 이번에는 G, B, V, C, D, E, R, T 키를 사용해보겠다. 이 키는 각 거북이들의 rotate_absolute 액션을 수행함에 있어서 액션의 목표 값을 전달하는 목적으로 사용된다. F키를 중심으로 주위의 8개의 버튼을 사용하는 것으로 다음 그림과 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 3시를 가리키는 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 12기 방향으로 회전하게 된다. 각 키에 할당된 라디안 값은 코드를 확인하면





119 액션 서버 및 클라이언트

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_key가 실행된 터미널 창에서 F 키를 눌러 액션 목표를 취소하게 되면 turtlesim_node이 실행된 터미널 창에 아래와 같이 목표가 취소 되었음을 알리면서 거북이는 그 자리에서 멈추게된다.

```
[INFO]: Rotation goal canceled
```

노드 정보 (ros2 node info)

```
$ 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
```

액션 목록 (ros2 action list -t)

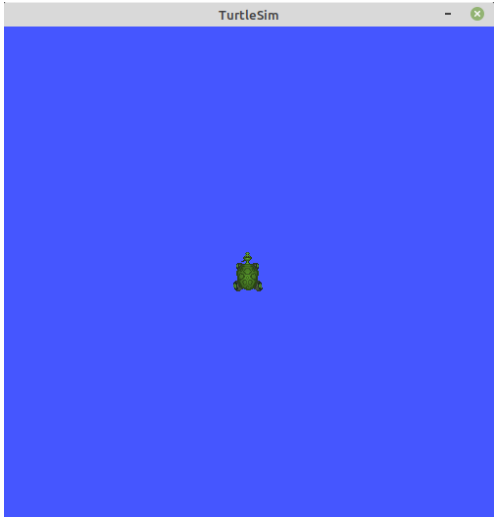
```
$ ros2 action list -t
/turtle1/rotate_absolute [turtlesim/action/RotateAbsolute]
```

액션 정보 (ros2 action info)

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

액션 목표(action goal) 전달

```
$ 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
```



액션 목표(action goal) 전달

```
$ 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
```

125 액션 인터페이스 (action interface, action)

여기서 다룬 액션 또한 토픽, 서비스와 마찬가지로 별도의 인터페이스를 가지고 있는데 이를 **액션 인터페이스**라 부르며, 파일로는 **action 파일**을 가르킨다. 액션 인터페이스는 메시지 및 서비스 인터페이스의 확장형이라고 볼 수 있는데 위에서 액션 목표를 전달할 때 실습으로 사용하였던 /turtle1/rotate_absolute 서비스를 예를 들어 설명하겠다.

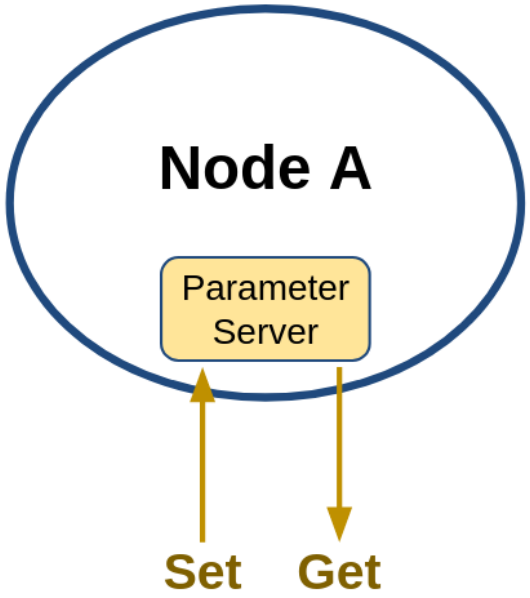
/turtle1/rotate_absolute 액션에 사용된 RotateAbsolute.action 인터페이스를 알아보기 위해서는 원본 파일을 참고해도 되고, `**ros2 interface show**` 명령어를 이용하여 확인할 수 있다. 이 명령어를 이용하면 아래의 결과 값과 같이 `turtlesim/action/RotateAbsolute.action`은 float32 형태의 theta, delta, remaining 라는 세개의 데이터가 있음을 알 수 있다. 여기서 서비스와 마찬가지로 **`---` 이라는 구분자를 사용하여 액션 목표(goal)), 액션 결과(result), 액션 피드백(feedback)으로 나누어 사용**하게 된다. 즉 theta는 액션 목표, delta는 액션 결과, remaining는 액션 피드백에 해당된다. 참고로 각 데이터는 각도의 SI 단위인 라디안(radian)을 사용한다.

```
$ ros2 interface show turtlesim/action/RotateAbsolute.action
# The desired heading in radians
float32 theta
---
# The angular displacement in radians to the starting position
float32 delta
---
# The remaining rotation in radians
float32 remaining
```

ROBOTIS

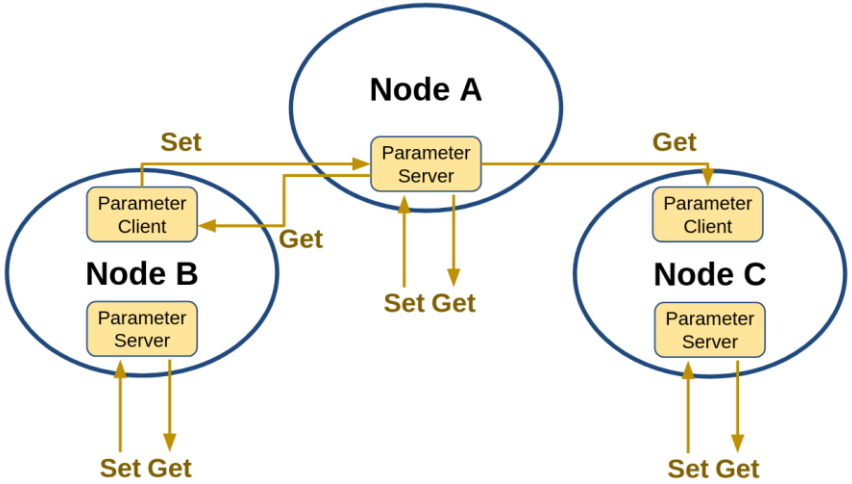
ROS 2 파라미터 (parameter)

파라미터 (parameter)



파라미터 (parameter)

파라미터 관련 기능은 **RCL(ROS Client Libraries)의 기본 기능**으로 다음 그림과 같이 모든 노드가 자신만의 **Parameter server**를 가지고 있고, 그림과 같이 각 노드는 **Parameter client**도 포함시킬 수 있어서 자기 자신의 파라미터 및 다른 노드의 파라미터를 읽고 쓸수 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.



파라미터 목록 확인 (ros2 param list)

```
$ ros2 run turtlesim turtlesim_node
```

```
$ ros2 run turtlesim turtle_teleop_key
```

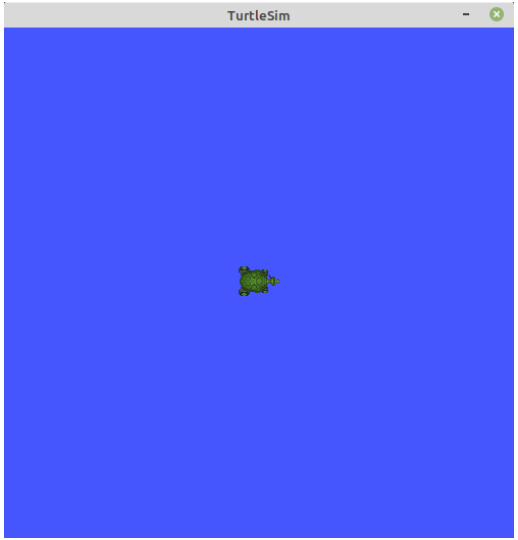
```
$ ros2 param list
/teleop_turtle:
  scale_angular
  scale_linear
  use_sim_time
/turtlesim:
  background_b
  background_g
  background_r
  use_sim_time
```

파라미터 내용 확인 (ros2 param describe)

```
$ 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
```

파라미터 읽기 (ros2 param get)

```
$ 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
```



파라미터 쓰기 (ros2 param set)

```
$ 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
```

133 파라미터 저장 (ros2 param dump) 및 재사용

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 초기 값으로 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
```

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

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

파라미터 삭제 (ros2 param delete)

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

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

ROBOTIS

ROS 2 토픽, 서비스, 액션 정리 및 비교

136 토픽, 서비스, 액션 비교

지금까지 강좌에서 다루었던 토픽, 서비스, 액션은 ROS의 중요 컨셉이자 앞으로 강좌에서 다룰 ROS 프로그래밍에 있어서 매우 중요한 부분이기에 다시 한번 비교를 해보도록 하겠다. 여기서 비교한 연속성, 방향성, 동기성, 다자간 연결, 노드 역할, 동작 트리거, 인터페이스를 각 토픽, 서비스, 액션의 서로 다른 특징이라고 볼 수 있고 노드간의 데이터 전송에 있어서 특성에 맞게 선택하여 ROS 프로그래밍을 하게 된다.

	토픽 (topic)	서비스 (service)	액션 (action)
연속성	연속성	일회성	복합 (토픽+서비스)
방향성	단방향	양방향	양방향
동기성	비동기	동기	동기 + 비동기
다자간 연결	1:1, 1:N, N:1, N:N (publisher:subscriber)	1:1 (server:client)	1:1 (server:client)
노드 역할	발행자 (publisher) 구독자 (subscriber)	서버 (server) 클라언트 (client)	서버 (server) 클라언트 (client)
동작 트리거	발행자	클라언트	클라언트
인터페이스	msg 인터페이스	srv 인터페이스	action 인터페이스
CLI 명령어	ros2 topic ros2 interface	ros2 service ros2 interface	ros2 action ros2 interface
사용 예	센서 데이터, 로봇 상태, 로봇 좌표, 로봇 속도 명령 등	LED 제어, 모터 토크 On/Off, IK/FK 계산, 이동 경로 계산 등	목적지로 이동, 물건 파지, 복합 태스크 등

137 토픽, 서비스, 액션 비교

아래의 표 2는 토픽, 서비스, 액션에서 사용되는 msg, srv, action 인터페이스를 비교한 내용이다. 모든 인터페이스는 msg 인터페이스의 확장형이라고 생각하면 되고 `---` 구분자가 0, 1, 2개로 몇개를 넣어 데이터를 구분했냐의 차이이다.

	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	**[geometry_msgs/msg/Twist]**	**[turtlesim/srv/Spawn.srv]**	**[turtlesim/action/RotateAbsolute.action]**
	Vector3 linear Vector3 angular	float32 x float32 y float32 theta string name --- string name	float32 theta --- float32 delta --- float32 remaining

ROS 2 도입을 위한 다음 단계는? (프로그래밍에 왕도는 없어요.)

ROS 2 소개

> 000 로봇 운영체제 ROS 강좌 목차
> 001 ROS 2 개발 환경 구축
> 002 ROS 2 기반 로봇 개발에 필요한 정보 링크 모음
> 003 왜? 'ROS 2'로 가야하는가?
> 004 ROS 2의 중요 컨셉과 특징
> 005 ROS 2의 특징
> 006 ROS 2와 DDS (Data Distribution Service)
> 007 패키지 설치와 노드 실행
> 008 ROS 2 노드와 메시지 통신
> 009 ROS 2 토픽 (Topic)
> 010 ROS 2 서비스 (Service)
> 011 ROS 2 액션 (Action)
> 012 ROS 2 토픽/서비스/액션 정리 및 비교
> 013 ROS 2 파라미터 (Parameter)
> 014 ROS 2 도구와 CLI 명령어
> 015 ROS의 종합 GUI 툴 RQt
> 016 ROS 2 인터페이스 (interface)
> 017 ROS 2의 물리량 표준 단위
> 018 ROS 2의 좌표 표현
> 019 DDS의 QoS (Quality of Service)
> 020 ROS 2의 파일 시스템
> 021 ROS 2의 빌드 시스템과 빌드 툴
> 022 패키지 파일 (환경 설정, 빌드 설정)
> 041 시간(Time, Duration, Clock, Rate)

ROS 2 기본 프로그래밍

> 023 ROS 프로그래밍 규칙 (코드 스타일)
> 024 ROS 프로그래밍 기초 (Python)
> 025 ROS 프로그래밍 기초 (C++)
> 027 토픽, 서비스, 액션 인터페이스
> 028 ROS 2 패키지 설계 (Python)
> 029 토픽 프로그래밍 (Python)
> 030 서비스 프로그래밍 (Python)
> 031 액션 프로그래밍 (Python)
> 032 파라미터 프로그래밍 (Python)
> 033 실행 인자 프로그래밍 (Python)
> 034 ROS 2 패키지 설계 (C++)
> 035 토픽 프로그래밍 (C++)
> 036 서비스 프로그래밍 (C++)
> 037 액션 프로그래밍 (C++)
> 038 파라미터 프로그래밍 (C++)
> 039 실행 인자 프로그래밍 (C++)
> 040 런치 프로그래밍 (Python, C++)

ROS 2 심화 프로그래밍

> 042 Logging
> 043 ROS2CLI
> 044 Intra-process communication
> 045 QoS
> 046 Component
> 047 RQt plugin
> 048 Life cycle
> 049 Security
> 050 Real-time

기타

> 026 ROS 2 Tips
> 051 TF
> 052 URDF
> 053 CI & Lint
> 054 Unit tests
> 055 Buildfarm, Binary Release
> 056 ROS 2 추천 패키지

ROBOTIS

The END