ROS 정리 보고서

2023741062 윤형준

목차

1. 로봇 운영체제

가장 먼저 Personal Computer 분업을 정착 -> 하드웨어 모듈 + 운영체제 + 앱 + 유저. 이를 따라 Personal Phone 또한 이 분업을 따랐다(ex : 컴퓨터 운영체제 - 윈도우, 리눅스 등 / 폰 운영체제 - 윈도우 폰, IOS 등). 이 분업에 의한 소프트웨어 플랫폼은 하드웨어, 운영체제, 어플을 서로 분리시켰다. 하드웨어의 인터페이스 통합 및 추상화, 규격화, 모듈화 가능하도록 했고 사용자 수요에 맞는 서비스에 집중할 수 있었다.

로봇 또한 이 분업의 형태를 따르고 있다(ex : 로봇 운영체제 – ROS, RTM, NAOqi 등). 이런 분업에 의한 로봇 소프트웨어 플랫폼의 긍정적 영향은 앞선 컴퓨터, 폰 분업의 긍정적 영향과 비슷하다. 하드웨어 & 소프트웨어 간의 인터페이스를 확립할 수 있고 하드웨어의 규격화/모듈화, 유저에게 서비스를 집중할 수 있다. 무엇보다 로봇 운영체제의 대중화로 인한 소프트웨어 인력들의 로봇분야 진입이다. (그 동안은 로봇 운영체제가 제대로 갖춰지지 않아 로봇 관련 종사자들만의 플랫폼을 이용하는 등의 방식을 취해왔지만 운영체제가 제대로 갖춰진다면 다른 분야의 소프트웨어 지식을 갖춘 인재들이 로봇분야에 쉽게 진입 및 소프트웨어 지식을 로봇분야에 활용하여 로봇분야의 발전 가속 가능)

2. ROS란

ROS(Robot Operating System)란 간단히 로봇 운영체제이다. 하지만 정확히는 Robot Meta-Operating System라고 하며 윈도우, 리눅스, 안드로이드와 같은 전통적인 운영체제는 아니다. ROS는 기존의 운영체제(리눅스, 윈도우즈, 안드로이드)를 이용한다. 메타 운영체제는 아래 자료를 보면 이해하기 쉽다.

ROS를 사용 가능한 운영체제로는 Ubuntu, OS X, Windows, Fedora, Gentoo, OpenSUSE, Debian, Raspbian, Arch, QNX Realtime OS 등이 있지만 기능 제한이 있을 수 있다. 스마트폰 운영체제 Android, iOS의 경운 부분적으로 사용 가능하다. 기본적으로는 Ubuntu, OS X에서 구동을 추천한다.

3. ROS특징

특징1 : 통신 인프라

• 메시지 파싱 기능 : 로봇 개발 시 많이 사용되는 통신 시스템 제공, 노드들 간의 메시지 전달 인터페이스 지원.

• 메시지의 기록 및 재생 : 노드 간 송/수신되는 데이터 메시지를 저장하고 필요시 재사용 가능, 이를 기반으로 반복적인 실험 및 알고리즘 개발에 용이.

• 메시지 사용으로 인한 다양한 프로그래밍 언어 사용 가능 : 노드 간의 데이터 교환이 메시지를 사용하기에 서로 다른 언어로 작성 가능.

• 분산 매개 변수 시스템 : 시스템에 사용되는 변수를 공유 및 수정하여 실시간으로 반영.

특징 2 : 다양한 기능

• 로봇에 대한 표준 메시지 정의 : 로봇 관련 다양한 데이터의 표준 메시지를 정의하여 모듈화, 협업 작업 유도, 효율성 향상.

• 로봇 기하학 라이브러리 : 로봇, 센서 등의 상대적 좌표를 트리화 하는 TF(Transform)제공.

• 로봇 기술 언어 : 로봇의 물리적 득성을 설명하는 XML 문서 기술.

• 진단 시스템 : 로봇의 상태 파악.

• 센싱/인식 : 센싱/인식 레벨의 라이브러리 제공.

• 네비게이션 : 로봇의 포즈, 지도 내에서 자기 위치 추정 제공 (SLAM/Navigation 라이브러리).

• 매니퓰레이션 : 로봇 암에 사용되는 다양한 Manipulation 라이브러리 제공, GUI 형태 Tools 제공.

특징 3 : 다양한 개발 도구

• Command-Line Tools : GUI없이 ROS 제공 명령어로만 로봇 엑세스 및 거의 모든 ROS 기능 소화.

• Rviz : 강력한 3D 시각화툴 제공, 센서 데이터 시각화, 로봇 동작 표현.

• RQT : GUI개발을 위한 Qt 기반 프레임 워크 제공.

• Gazebo : 물리 엔진 탑재 지원, 3차원 시뮬레이터.

4. ROS 용어

• Node : 최소 단위의 실행 가능한 프로세서. (ex : 얼굴 매칭 프로그램을 만든다고 할 때 한 번에 일련의 프로그램을 짜지 않음 -> 카메라 Law 데이터 받음, 데이터 필터1, 필터2, 얼굴매칭, 하드웨어 제어 -> 여기선 총 5개의 노드)

• Package : 하나 이상의 노드, 노드 실행을 위한 정보 등을 묶어 놓은 것. (ex :

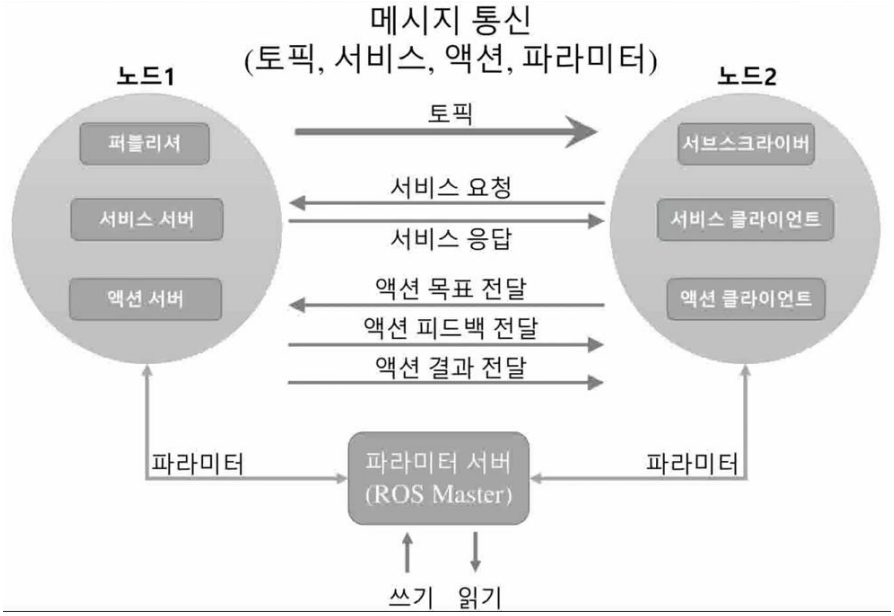
위 Node 예시에서 5개의 노드를 묶어 얼굴 매칭하는 패키지.)

- Message : 이를 통해 노드간의 데이터를 주고 받음. 메시지는 integer, floating point, Boolean 와 같은 변수형태.
- Topic : 단방향, 연속성을 가진 메시지 통신 방법 / 메시지가 전달되는 곳.
- Publisher : 메시지를 보내는 쪽. (Publisher Node)
- Subscriber : 메시지를 받는 쪽. (Subscriber Node)

Publisher -> (Topic) -> Subscriber (Publisher/Subscriber -> 1:1, N:1, 1:N, N:N가능)

- Service : 양방향, 일회성 메시지 통신 방법.
- Service server : client쪽의 요청으로 수행하는 쪽.
- Service client : server쪽에 요청하는 쪽.
- Action : 양방향 메시지 통신 방법 -> 중간중간 피드백(중간 결과) 전달.
- Action server : client쪽의 요청으로 수행하며 액션 피드백을 전달하는 쪽.
- Action client : server쪽에 요청하는 쪽.

5. 메시지 통신 개념



1. 마스터 구동 - XML기반 간단한 서버 구동($rosrun).

2. Subscriber Node 구동 - 마스터로 Topic, IP, Port 등 Subscriber노드 정보 전송 ($rosrun 패키지이름 노드이름).

3. Publisher Node 구동 - 마스터로 Topic, IP, Port 등 Publisher노드 정보 전송 ($rosrun 패키지이름 노드이름).

4. Publisher 정보 알림 - 마스터가 Subscriber Node에게 새로운 Publisher 정보를 알림.

5. Publisher Node에 접속 요청 - 마스터로부터 받은 Publisher 정보를 이용하여 TCPROS접속 요청(마스터없이 서로간 통신).

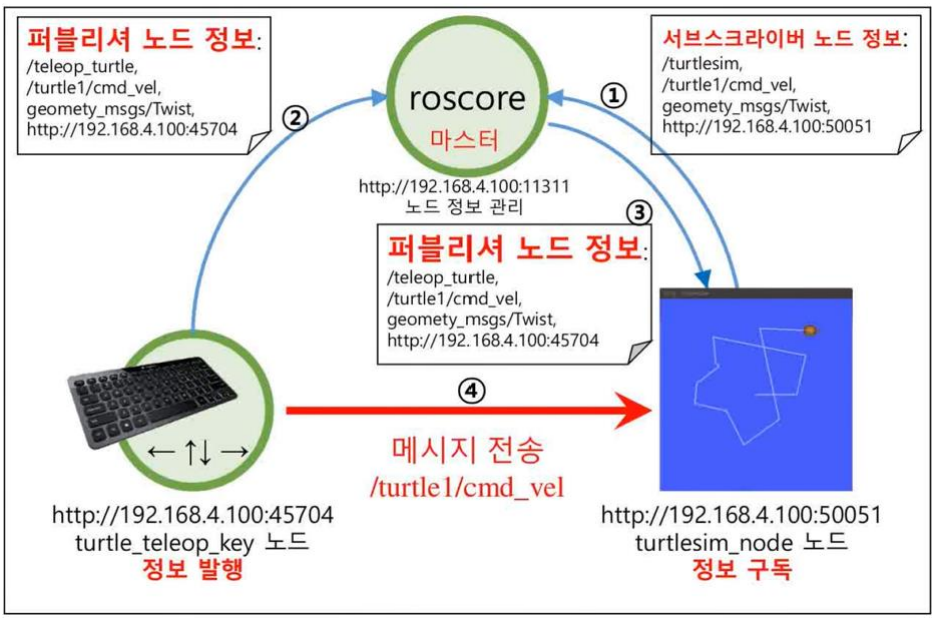
6. Subscriber Node에 접속 응답.

7. TCP접속 - TCPROS를 이용하여 Publisher Node와 직접 연결(XMLRPC -> TPCROS).

8. 메시지 전송 - Publisher Node가 Subscriber Node에게 메시지 전송(Topic).

9. 서비스 요청 및 응답.

Ex) Turtlesim 도식화



+Ex) 이기종 디바이스 간 통신

명령어 eb -> Publisher쪽 IP를 MASTER localhost부분에 작성.

예를 들어, 타기기에서 카메라 정보를 가져오고 싶다면 타기기 IP를 이용해 위와 같이 설정 후 rqt로 이미지 플러그인.

6. 메시지

- 노드 간 데이터를 송/수신할 때 사용하는 데이터의 형태.

  메시지 관련 정보 : https://wiki.ros.org/msg

  자주 쓰는 메시지 정보 : https://wiki.ros.org/common_msgs

- 단순 자료형, 메지시 안에 메시지를 품고 있는 데이터 구조, 메시지들이 나열된 배열과 같은 구조 형태들이 있음.

7. 네임

- 노드, 메시지가 가지는 고유 식별자.
- Global : 문자없이 네임을 바로 쓰거나 네임 앞에 슬래쉬(/)를 붙임.
- Private : 네임 앞에 틸트(~)를 붙임.

8. ROS 명령어

ROS 명령어 정보(사이트 8.~) : https://wiki.ros.org/Tools

8.1. 셸 명령어

명령어	중요도	설명

ros+cd(changes directory)	★★★	지정한 ROS패키지의 디렉터리로 이동
ros+ls(lists files)	★	ROS패키지의 파일 목록 확인
ros+ed(editor)	★	ROS패키지의 파일 편집
ros+cp(copies files)	★	ROS패키지의 파일 복사
ros+puched		ROS디렉터리 인덱스에 디렉터리 추가
ros+directory		ROS디렉터리 인덱스 확인

8.2. 실행 명령어

명령어	중요도	설명
ros+core	★★★	• master – ROS네임 서비스 • rosout – 로그 기록 • parameter server – 파라미터 관리
ros+run	★★★	노드 실행
ros+launch	★★★	노드를 여러 개 실행 및 실행 옵션 설정
ros+clean	★★	ROS 로그 파일을 검사하거나 삭제

8.3. 정보(확인) 명령어

명령어	중요도	설명
ros+topic	★★★	ROS 토픽 정보 확인
ros+service	★★★	ROS 서비스 정보 확인
ros+node	★★★	ROS 노드 정보 확인
ros+param(parameter)	★★★	ROS 파라미터 정보 확인 및 수정
ros+bag	★★★	ROS 메시지 기록 및 재생
ros+msg	★★	ROS 메시지 정보 확인
ros+srv	★★	ROS 서비스 정보 확인
ros+version	★	ROS패키지 및 배포 릴리즈 버전 정보 확인
ros+wtf		ROS 시스템 검사

8.4. catkin 명령어

명령어	중요도	설명
catkin_create_pkg	★★★	패키지 자동 생성
catkin_make	★★★	캐킨 빌드 시스템에 기반을 둔 빌드
catkin_eclipse	★★	캐킨 빌드 시스템으로 생성한 패키지를 이클립스에서 사용할 수 있게 변경
catkin_prepare_release	★★	릴리즈할 때 사용되는 로그 정리 및 버전 태깅
catkin_generate_changelog	★★	릴리즈할 때 CHANGELOG.rst 파일 생성 또는 업데이트
catkin_init_workspace	★★	캐킨 빌드 시스템의 작업 폴더 초기화
catkin_find	★	캐킨 검색

8.5. 패키지 명령어

명령어	중요도	설명
ros+pack(package)	★★★	ROS 패키지 관련 정보 보기
ros+install	★★	ROS 추가 패키지 설치
ros+dep(dependencies)	★★	해당 패키지의 의존성 파일 설치
ros+locate		ROS 패키지 정보 관련 명령어
ros+create-pkg		ROS 패키지 자동 생성
ros+make		ROS 패키지 빌드

9. ROS 도구

9.1. RViz

RViz란 ROS Visualization Tool의 약자로 ROS의 시각화 툴이다.

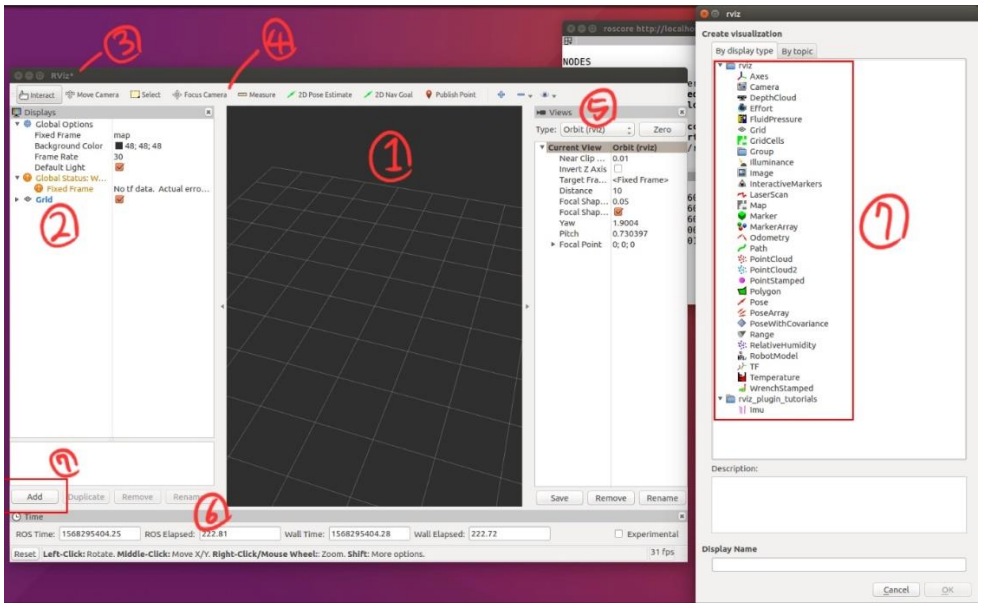
- ROS 3D 시각화툴 – 센서 데이터의 시각화, 레이저 거리 센서(LDS)의 거리 데이터, Depth Camera의 포인트 클라우드 데이터, 영상 데이터, 관성 데이터 등
- 로봇 외형의 표시와 계획된 동작의 표현 – URDF(Unified Robot

Description Format)

- 내비게이션
- 매니퓰레이션
- 원격 제어

RViz 설치 및 실행

- 설치 : $sudo apt-get install ros-kinetic-rviz
- 실행 : $rosrun rviz rviz OR $rviz
- 화면 구성 : 아래 사진



1. 3D 뷰
2. 디스플레이
3. 메뉴
4. 툴
5. 뷰 형태
6. 시간
7. 디스플레이 종류

9.2. RQT

9.2.1. 도입

RQT란 플러그인 방식의 ROS 종합 GUI 툴이다.

RQT 설치 및 실행

- 설치 : $sudo apt-get install ros-kinetic-rqt ros-kinetic-rqt-common-plugins
- 실행 : $rqt

9.2.2. 플러그인 종류

RQT 플러그인 종류

1. 액션(Action)
   - Action Type Browser : Action 타입의 데이터 구조 확인
2. 구성(Configuration)
   - Dynamic Reconfigure : 노드들에서 제공하는 설정값 변경을 위한 GUI 설정값 변경.
   - Launch : roslaunch의 GUI 버전.
3. 내성(Introspection)
   - Node Graph : 구동중인 노드들의 관계도 및 메시지의 흐름을 확인 가능한 그래프 뷰.
   - Package Graph : 노드의 의존 관계를 표시하는 그래프 뷰.
   - Process Monitor : 실행중인 노드들의 CPU사용률, 메모리 사용률 등을 확인.
4. 로깅(Logging)
   - Bag : ROS 데이터 로깅(녹화 및 재생).
   - Console : 노드들에서 발생되는 경고, 에러 등의 메시지를 확인.
   - Logger Level : ROS의 Debug, Info, Error 등 로거 정보를 선택하여 표시.
5. 다양한 툴(Miscellaneous Tools)

- Python Console : 파이썬 콘솔 화면.
- Shell : 셸 구동.
- Web : 웹 브라우저 구동.

6. 로봇(Robot)
   - 사용하는 로봇에 따라 계기판(dashboard0) 등의 플러그인을 추가.
7. 로봇툴(Robot Tools)
   - Controller Manager : 컨트롤러 제어에 필요한 플로그인.
   - Diagnostic Viewer : 로봇 디바이스 및 에러 확인.
   - Moverit! Monitor : 로봇 팔 계획에 사용되는 Moveit! 데이터를 확인.
   - Robot Steering : 로봇 조정 GUI툴, 원격 조정에서 이 툴을 이용해 로봇 조종.
   - Runtime Monitor : 실시간으로 노드들에서 발생되는 에러 및 경고를 확인.
8. 서비스(Service)
   - Service Caller : 구동중인 서비스 서버에 접속해 서비스 요청.
   - Service Type Browser : 서비스 타입의 데이터 구조를 확인.
9. 토픽(Topics)
   - Easy Message Publisher : 토픽을 GUI 환경에서 발행.
   - Topic Publisher : 토픽을 생성하여 발행.
   - Topic Type Browser : 토픽 타입의 데이터 구조 확인.
   - Topic Monitor : 사용자가 선택한 토픽의 정보 확인.
10. 시각화(Visualization)
    - Image View : 카메라의 영상 데이터 확인.

- Navigation Viewer : 로봇 네비게이션의 위치 및 목표지점 확인.
- Plot : 2차원 데이터 플롯 GUI 플러그인, 2차원 데이터의 도식화.
- Pose View : 현재 TF의 위치 및 모델의 위치 표시.
- RViz : 3차원 시각화 툴 RViz 플러그인.
- TF Tree : TF관계를 트리로 나타내는 그래프 뷰.

9.2.2. 간단한 예시들

Ex1) rqt_image_view – 영상 데이터 확인 예시 :

$rosrun uvc_camera uvc_camera_node -> $rqt -> [Plugins] -> [Visualization] -> [Image View].

Ex2) rqt_graph – 노드들의 도식화 예시 :

$rosrun image_view image_view image:=image_raw -> $rqt -> [Plugins] -> [Introspection] -> [Node_Graph].

EX3) rqt_bag – 영상 녹화 및 재생 예시 :

$rosrun uvc_camera uvc_camera_node -> $rosbag record/image_raw -> $rqt -> [Plugins] -> [Logging] -> [Bag].

9.3. Gazebo

Gazebo란 물리 엔진을 탑재하여 실제와 근사한 결과를 얻을 수 있는 3차원 시뮬레이터이다. 로봇 개발시 필요한 시뮬레이션 로봇, 센서, 환경 모델 등을 지원한다.

설치 : $sudo apt-get install ros-melodic-gazebo-ros

10. ROS 프로그래밍

10.1. 규칙

- 표준 단위 - SI단위 사용(전 세계적으로 통용되는 단위 체계로 미터(m), 킬로그램(kg), 초(s), 암페어(A), 켈빈(K), 몰(mol), 칸델라(cd) 7가지).
- 좌표 표현 방식 – x : 앞, y : 왼쪽, z : 위 (오른손 법칙)

양(Quantity)	단위(Unit)
Angle	Radian
Frequency	Hertz
Force	Newton
Power	Watt
Voltage	Volt
Length	Meter
Mass	Kilogram
Time	Second
Current	Ampere
Temperature	Celsius

사이트 자세한 정보 : https://www.ros.org/reps/rep-0103.html

- 프로그래밍 규칙 정보

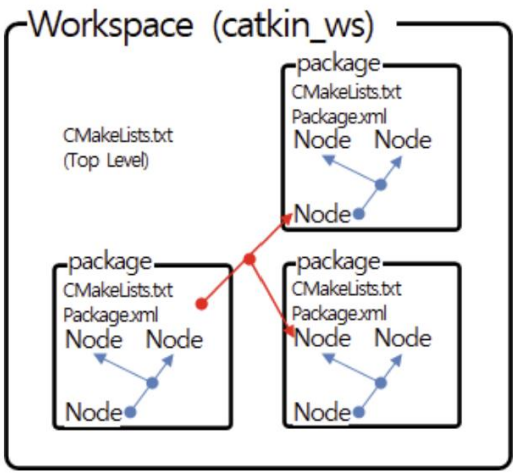
대상	명명 규칙	예시
패키지	under_scored	first_ros_package
토픽, 서비스	under_scored	raw_image
파일	under_scored	turtlebot3_fake.cpp
네임스페이스	under_scored	ros_awesome_package
변수	under_scored	string table_name;

타입	CamelCased	typedef int32_t PropertiesNumber;
클래스	CamelCased	class UrlTable
구조체	CamelCased	struct UrlTableProperties
열거형	CamelCased	enum ChoiceNumber;
함수	CamelCased	addTableEntry();
메소드	CamelCased	void setNumEntries(int32_t num_entries)
상수	ALL_CAPITALS	const uint8_t DAYS_IN_A_WEEK = 7;
매크로	ALL_CAPITALS	#define PI_ROUNDED 3.0

사이트 자세한 정보 : https://wiki.ros.org/StyleGuide

10.2. Catkin

- Catkin이란 공통된 소스를 가지는 ROS패키지들을 하나로 묶어 관리하는 워크스페이스이고 ROS에서 제공하는 다양한 툴을 이용해 쉽게 구축할 수 있다. ROS에서 다양한 프로그램을 사용하려면 그 코드를 읽어 빌드해야 한다. ROS는 빌드한 기능을 불러와 내가 만들려는 프로그램에서 사용하는데 이때 사용하는 빌드 시스템이 캣킨(Catkin)이다.
- 작업공간(Workspace)이란 내가 만드는 프로그램과 관련된 ROS코드를 모아두는 디렉토리이다. 여러 개의 작업공간을 가질 순 있지만 하나의 작업공간에서만 작업 가능하다.
- Carkin workspace의 구성을 간단히 표현하면 다음과 같다고 한다.



Catkin workspace 환경을 최초로 만드는 과정은 다음과 같다.

```
$mkdir -p ~/catkin_ws/src $cd ~/catkin_ws $catkin_make
```

mkdir를 통해 경로를 생성하고 catkin_ws로 진입해 환경을 만들어 준다. catkin_make는 catkin을 이용한 작업과정에서 쓰이는 편리한 도구로 여기서의 초기설정 뿐 아니라 이후 작업 시 package내 변경사항을 수정할 때도 사용된다. catkin_ws디렉토리 밑에는 총 3개의 하위 디렉토리(build, devel, src)가 있다. 3개의 각 하위 디렉토리는 다음과 같은 역할을 한다.

build디렉토리 : c++을 사용하면 라이브러리와 실행 프로그램과 같은 캣킨 작업 결과 중 일부를 저장하는 곳.

devel디렉토리 : 시스템이 현재 작업 공간과 그 안에 포함된 코드를 사용하도록 위치를 알려주는 환경설정 파일이 있는 곳.

src디렉토리 : 로봇 구동시키는 ROS패키지를 저장하는 곳.

10.3. Topic프로그래밍

1. 패키지 생성

```
$cd ~/catkin_ws/src
```

```
$catkin_create_pkg ros_tutorials_topic message_generation std_msgs rospy
```

ros_tutorials_topic : 패키지 이름.

message_generation std_msgs rospy : 패키지가 3개에 대한 의존성을 지님.

```
$cd ros_tutorials_topic/
```

```
$ls
```

cd ros_tutorials_topic/ : 생성한 패키지로 이동

ls : (리스트) 패키지에 있는 파일들 확인

2. 패키지 설정 파일 수정

```
$gedit package.xml
<?xml version="1.0"?>
<package format="2">
<name>ros_tutorials_topic</name>
<version>0.1.0</version>
<description>ROS turtorial package to learn the topic</description>
<license>Apache 2.0</license>
<author email="pyo@robotis.com">Yoonseok Pyo</author>
<maintainer email="pyo@robotis.com">Yoonseok Pyo</maintainer>
<url type="website">http://www.robotis.com</url>
<url type="repository">https://github.com/ROBOTIS-GIT/ros_tutorials.git</url>
<url type="bugtracker">https://github.com/ROBOTIS-GIT/ros_tutorials/issues</url>
<buildtool_depend>catkin</buildtool_depend>
<build_depend>roscpp</build_depend>
<build_depend>std_msgs</build_depend>
<build_depend>message_generation</build_depend>
<run_depend>roscpp</run_depend>
<run_depend>std_msgs</run_depend>
<run_depend>message_runtime</run_depend>
<export> </export>
</package>
```

<description>ROS turtorial package to learn the topic</description> : 패키지에 대한 간단한 설명(1~2 줄).

<license>Apache 2.0</license> : 개발시 적용되는 라이선스.

<author email="pyo@robotis.com">Yoonseok Pyo</author> : 저자 작성.

<maintainer email="pyo@robotis.com">Yoonseok Pyo</maintainer> : 사용자, 관리자 작성.

<url type="website">http://www.robotis.com</url>, <url type="repository">https://github.com/ROBOTIS-GIT/ros_tutorials.git</url>, <url type="bugtracker">https://github.com/ROBOTIS-GIT/ros_tutorials/issues</url> : 추가 URL.

<build_depend> : 빌드 시 의존성 기술.

<run_depend> : 실행 시 의존성 기술.

roscpp : 언어 c++ 사용.

std_msgs : 표준 메시지 형태 사용.

message_generation : 메시지 생성.

3. 빌드 설정 파일

```
$gedit CMakeLists.txt
cmake_minimum_required(VERSION 2.8.3)
project(ros_tutorials_topic)
## 캐킨 빌드를 할 때 요구되는 구성요소 패키지
## 의존성 패키지로 message_generation, std_msgs, roscpp 이며 이 패키지들이 존재하지 않으면 빌드 도중에 에러가 남
find_package(catkin REQUIRED COMPONENTS message_generation std_msgs roscpp)
## 메시지 선언: MsgTutorial.msg
add_message_files(FILES MsgTutorial.msg)
## 의존하는 메시지를 설정하는 옵션
## std_msgs 가 설치되어 있지 않다면 빌드 도중에 에러가 남
generate_messages(DEPENDENCIES std_msgs)
## 캐킨 패키지 옵션으로 라이브러리, 캐킨 빌드 의존성, 시스템 의존 패키지를 기술
catkin_package(
LIBRARIES ros_tutorials_topic
CATKIN_DEPENDS std_msgs roscpp
)
```

```
## 인클루드 디렉터리를 설정
include_directories(${catkin_INCLUDE_DIRS})
## topic_publisher 노드에 대한 빌드 옵션
## 실행 파일, 타깃 링크 라이브러리, 추가 의존성 등을 설정
add_executable(topic_publisher src/topic_publisher.cpp)
add_dependencies(topic_publisher ${${PROJECT_NAME}_EXPORTED_TARGETS}
${catkin_EXPORTED_TARGETS})
target_link_libraries(topic_publisher ${catkin_LIBRARIES})
## topic_subscriber 노드에 대한 빌드 옵션
add_executable(topic_subscriber src/topic_subscriber.cpp)
add_dependencies(topic_subscriber ${${PROJECT_NAME}_EXPORTED_TARGETS}
${catkin_EXPORTED_TARGETS})
target_link_libraries(topic_subscriber ${catkin_LIBRARIES})
```

cmake_minimum_required(VERSION 2.8.3) : 버전이 2.8.3 이상이면 빌드할 때 큰 문제없다는 것을 기술.

add_executable(topic_~ src/topic_~) : src/topic_~ 을 참조해(의존해) topic_~을 생성하라는 것을 기술.

add_dependencies(topic_~ $~) : topic_~ 을 빌드 시 $~을 참조(의존)하라는 것을 기술.

4. 메시지 파일 작성

```
$roscd ros_tutorials_topic
$mkdir msg
$cd msg
$gedit MsgTutorial.msg
time stamp
int32 data
```

roscd ros_tutorials_topic : 패키지 폴더로 이동.

mkdir msg : 패키지에 msg 라는 메시지 폴더 새로 작성.

cd msg : 작성한 msg 폴더로 이동.

gedit MsgTutorial.msg : MsgTutorial.msg 파일 새로 작성 및 내용 수정.

time stamp : 메시지 송신 시 시간 저장을 위한 변수.

int32 data : 데이터 저장을 위한 변수.

5. Publisher Node 작성

```
$roscd ros_tutorials_topic/src
$gedit topic_publisher.cpp
#include "ros/ros.h" // ROS 기본 헤더파일
#include "ros_tutorials_topic/MsgTutorial.h" // MsgTutorial 메시지 파일 헤더(빌드 후 자동 생성됨)
int main(int argc, char **argv) // 노드 메인 함수
{
ros::init(argc, argv, "topic_publisher"); // 노드명 초기화
ros::NodeHandle nh; // ROS 시스템과 통신을 위한 노드 핸들 선언
// 퍼블리셔 선언, ros_tutorials_topic 패키지의 MsgTutorial 메시지 파일을 이용한
// 퍼블리셔 ros_tutorial_pub 를 작성한다. 토픽명은 "ros_tutorial_msg" 이며,
// 퍼블리셔 큐(queue) 사이즈를 100 개로 설정한다는 것
ros::Publisher ros_tutorial_pub = nh.advertise<ros_tutorials_topic::MsgTutorial>("ros_tutorial_msg",
100);
// 루프 주기를 설정한다. "10" 이라는 것은 10Hz 를 말하는 것으로 0.1 초 간격으로 반복
ros::Rate loop_rate(10);
// MsgTutorial 메시지 파일 형식으로 msg 라는 메시지를 선언
ros_tutorials_topic::MsgTutorial msg;
// 메시지에 사용될 변수 선언
int count = 0;
while (ros::ok())
{
```

```
msg.stamp = ros::Time::now(); // 현재 시간을 msg 의 하위 stamp 메시지에 담음
msg.data = count; // count 라는 변수 값을 msg 의 하위 data 메시지에 담음
ROS_INFO("send msg = %d", msg.stamp.sec); // stamp.sec 메시지를 표시
ROS_INFO("send msg = %d", msg.stamp.nsec); // stamp.nsec 메시지를 표시
ROS_INFO("send msg = %d", msg.data); // data 메시지를 표시
ros_tutorial_pub.publish(msg); // 메시지를 발행
loop_rate.sleep(); // 위에서 정한 루프 주기만큼 슬립(쉼)
++count; // count 변수 1 씩 증가
}
return 0;
}
```

roscd ros_tutorials_topic/src : 패키지 소스 폴더인 src 폴더로 이동.

gedit topic_publisher.cpp : 소스 파일 새로 작성 및 내용 수정.

ros::Publisher ros_tutorial_pub = nh.advertise<ros_tutorials_topic::MsgTutorial>("ros_tutorial_msg", 100); : 여기서의 노드가 Publisher 의 역할을 한다고 기술. 큐(queue)사이즈가 100 이라는 것은 네트워크의 지연 등으로 데이터 지연 시 가지고 있을 저장하고 있을 데이터의 수를 의미.

ROS_INFO("",); : 간단히 목적에 따라 나뉘어진 printf 중 하나.

6. Subscriber Node 작성

```
$roscd ros_tutorials_topic/src
$gedit topic_subscriber.cpp
#include "ros/ros.h" // ROS 기본 헤더파일
#include "ros_tutorials_topic/MsgTutorial.h" // MsgTutorial 메시지 파일 헤더 (빌드 후 자동 생성됨)
// 메시지 콜백 함수로써, 밑에서 설정한 ros_tutorial_msg 라는 이름의
// 토픽 메시지를 수신하였을 때 동작하는 함수
```

```
// 입력 메시지로는 ros_tutorials_topic 패키지의 MsgTutorial 메시지를 받도록 되어 있음
void msgCallback(const ros_tutorials_topic::MsgTutorial::ConstPtr& msg)
{
ROS_INFO("recieve msg = %d", msg->stamp.sec); // stamp.sec 메시지를 표시
ROS_INFO("recieve msg = %d", msg->stamp.nsec); // stamp.nsec 메시지를 표시
ROS_INFO("recieve msg = %d", msg->data); // data 메시지를 표시
}
int main(int argc, char **argv) // 노드 메인 함수
{
ros::init(argc, argv, "topic_subscriber"); // 노드명 초기화
ros::NodeHandle nh; // ROS 시스템과 통신을 위한 노드 핸들 선언
// 서브스크라이버 선언, ros_tutorials_topic 패키지의 MsgTutorial 메시지 파일을 이용한
// 서브스크라이버 ros_tutorial_sub 를 작성한다. 토픽명은 "ros_tutorial_msg" 이며,
// 서브스크라이버 큐(queue) 사이즈를 100 개로 설정한다는 것
ros::Subscriber ros_tutorial_sub = nh.subscribe("ros_tutorial_msg", 100, msgCallback);
// 콜백함수 호출을 위한 함수로써, 메시지가 수신되기를 대기,
// 수신되었을 경우 콜백함수를 실행
ros::spin();
return 0;
}
```

7. ROS 노드 빌드

```
$cd ~/catkin_ws
$catkin_make
```

cd ~/catkin_ws : catkin 폴더로 이동.

catkin_make : catkin 빌드 실행.

8. Publisher 실행

[노드 실행에 앞서 roscore 실행.]

```
$rosrun ros_tutorials_topic topic_publisher
```

- 참고 – rostopic : 현재 ROS 네트워크에서 사용 중인 토픽 목록, 주기, 내용 확인 등이 가능.

```
$rostopic list
/ros_tutorial_msg
/rosout
/rosout_agg
$rostopic info /ros_tutorial_msg
```

9. Publisher 실행

```
$rosrun ros_tutorials_topic topic_subscriber
```

10. 노드들의 통신 상태 확인

$rqt_graph -> $rqt -> [Plugins] -> [Inreospection] -> [Node Graph]

- 참고 : 복잡한 과정 없이 github 에 의한 소스코드로 쉽게 설정 가능.

```
$cd ~/catkin_ws/src
$git clone https://github.com/ROBOTIS-GIT/ros_tutorials.git
$cd ~/catkin_ws
$catkin_make
```

10.4. Service 프로그래밍

- Topic 프로그래밍과 대부분 비슷하기에 차이점 위주로 기술함.

1. 패키지 생성

패키지 이름 변화 -> ros_tutorials_service

2. 패키지 설정 파일 수정

이름 변화 주의

3. 빌드 설정 파일 수정

```
##서비스 선언 : SrvTutorial.srv
add_service_files(FILES Srv Tutorial.srv)
```

4. 서비스 파일 작성

```
$roscd ros_tutorials_service
$mkdir srv
$cd srv
$gedit SrvTutorial.srv
int64 a
int64 b
int64 result
```

int64 a, int64 b / int64 result : int64 a, b 는 서비스 요청 변수이고 int64 result 는 서비스 응답 변수.

5. Service server Node 작성

```
#include “ros_tutorials_service/SrvTutorial.h” // SrvTutorial 서비스 파일 헤더(빌드 후 자동 생성됨)

// 서비스 요청이 있을 경우, 아래의 처리를 수행
// 서비스 요청은 req, 서비스 응답은 res 로 설정
bool calculation(ros_tutorials_service::SrvTutorial::Request &req,
ros_tutorials_service::SrvTutorial::Response &res)
{
// 서비스 요청시 받은 a 와 b 값을 더하여 서비스 응답 값에 저장
res.result = req.a + req.b;
// 서비스 요청에 사용된 a, b 값의 표시 및 서비스 응답에 해당되는 result 값을 출력
```

```
ROS_INFO("request: x=%ld, y=%ld", (long int)req.a, (long int)req.b);
ROS_INFO("sending back response: %ld", (long int)res.result);
return true;
}
int main(int argc, char **argv) // 노드 메인 함수
{ ros::init(argc, argv, "service_server"); // 노드명 초기화
ros::NodeHandle nh; // 노드 핸들 선언
// 서비스 서버 선언, ros_tutorials_service 패키지의 SrvTutorial 서비스 파일을 이용한
// 서비스 서버 ros_tutorials_service_server 를 선언
// 서비스명은 ros_tutorial_srv 이며 서비스 요청이 있을 때,
// calculation 라는 함수를 실행하라는 설정
ros::ServiceServer ros_tutorials_service_server = nh.advertiseService("ros_tutorial_srv", calculation);
ROS_INFO("ready srv server!");
ros::spin(); // 서비스 요청을 대기
return 0;
}
```

6. Service client Node 작성

```
#include "ros_tutorials_service/SrvTutorial.h" // SrvTutorial 서비스 파일 헤더 (빌드 후 자동
생성됨)
#include<cstdlib> //atoll 함수 사용을 위한 라이브러리

int main(int argc, char **argv) // 노드 메인 함수
{
ros::init(argc, argv, "service_client"); // 노드명 초기화
if (argc != 3) // 입력값 오류 처리
{
ROS_INFO("cmd : rosrun ros_tutorials_service service_client arg0 arg1");
ROS_INFO("arg0: double number, arg1: double number");
```

```
return 1;
}
ros::NodeHandle nh; // ROS 시스템과 통신을 위한 노드 핸들 선언
// 서비스 클라이언트 선언, ros_tutorials_service 패키지의 SrvTutorial 서비스 파일을
// 이용한 서비스 클라이언트 ros_tutorials_service_client 를 선언
// 서비스명은 "ros_tutorial_srv"
ros::ServiceClient ros_tutorials_service_client =
nh.serviceClient<ros_tutorials_service::SrvTutorial>("ros_tutorial_srv");
// srv 라는 이름으로 SrvTutorial 서비스 파일을 이용하는 서비스를 선언
ros_tutorials_service::SrvTutorial srv;
// 서비스 요청 값으로 노드가 실행될 때 입력으로 사용된 매개변수를 각각의 a, b 에 저장
srv.request.a = atoll(argv[1]);
srv.request.b = atoll(argv[2]);
// 서비스를 요청하고, 요청이 받아들여졌을 경우, 응답 값을 표시
if (ros_tutorials_service_client.call(srv))
{
ROS_INFO("send srv, srv.Request.a and b: %ld, %ld", (long int)srv.request.a, (long int)srv.request.b);
ROS_INFO("receive srv, srv.Response.result: %ld", (long int)srv.response.result);
}
else
{
ROS_ERROR("Failed to call service ros_tutorial_srv");
return 1;
}
return 0;
}
```

7. ROS 노드 빌드

```
$cd ~/catkin_ws && catkin_make
```

-> catkin 폴더로 이동 후 catkin 빌드 실행.

8. Service server 실행

```
$rosrun ros_tutorials_service service_server
```

-> 정상 작동하면 [INFO] [1495726541.268629564]: ready srv server! 이 출력되고 요청을 기다림.

9. Service client 실행

```
$rosrun ros_tutorials_service service_client 2 3
```

-> 정상 작동한다면 매개변수 2 와 3 을 요청받았으니 응답값으로 2+3 인 5 를 전송받아 다음과 같이 출력된다.

[INFO] [1495726543.277216401]: send srv, srv.Request.a and b: 2, 3

[INFO] [1495726543.277258018]: receive srv, srv.Response.result: 5

- 참고 1 – rosservice call 명령어 : 일회성이 아니고 응답값을 계속해서 부를 수 있음. `$rosservice call /ros_tutorial_srv 10 2`
- 참고 2 – GUI 도구 Service Caller : rqt 로 매개변수를 바꾸며 일회성이 아닌 계속해서 응답값을 부를 수 있음. $rqt -> [Plugins] -> [Service] -> [Service Caller] -> service 에 /ros_tutorial_srv 설정 -> 매개변수 입력 후 Call
- 참고 3 - 복잡한 과정 없이 github 에 의한 소스코드로 쉽게 설정 가능.

```
$cd ~/catkin_ws/src
$git clone https://github.com/ROBOTIS-GIT/ros_tutorials.git
$cd ~/catkin_ws
$catkin_make
```

10.5. 파라미터 활용

파라미터(parameter)는 사용자가 노드들 간의 주고 받는 데이터를 저장하거나 이를 변경해 조종 가능하도록 해준다. 그렇다면 왜 파라미터를 사용하는가에 대한 의문이 들 것이다. 앞서 확인한 토픽이나 서비스를 보았을 때 시리얼 창에 입력을 하면 데이터가 변해서 출력이 되거나 하는데 왜 사용하는가. 그 이유는 바로 시스템 내 모든 변수들 또는 사용자가 접근하기 원하는 변수들에 쉽게

접근하고, 무엇보다 노드들 간의 통신이 계속해서 이루어지고 있는 상황에 실시간으로 값들을 바꿔줄 수 있기 때문이다.

- 10.3 에서의 내용에서 service_server.cpp 소스 파일 부분 수정.

1. 파라미터 노드 작성

```
#include “ros/ros.h” // ROS 기본 헤더파일
#include “ros_tutorials_service/SrvTutorial.h” // SrvTutorial 서비스 파일 헤더 (빌드 후 자동 생성됨)
#define PLUS 1 // 덧셈
#define MINUS 2 // 빼기
#define MULTIPLICATION 3 // 곱하기
#define DIVISION 4 // 나누기
int g_operator = PLUS;
// 서비스 요청이 있을 경우, 아래의 처리를 수행
// 서비스 요청은 req, 서비스 응답은 res 로 설정
bool calculation(ros_tutorials_service::SrvTutorial::Request &req,
ros_tutorials_service::SrvTutorial::Response &res)
{ // 서비스 요청시 받은 a 와 b 값을 파라미터 값에 따라 연산자를 달리함
// 계산한 후 서비스 응답 값에 저장
switch(g_operator)
{
case PLUS: res.result = req.a + req.b; break;
case MINUS: res.result = req.a - req.b; break;
case MULTIPLICATION: res.result = req.a * req.b; break;
case DIVISION:
if(req.b == 0){ res.result = 0; break; }
else{ res.result = req.a / req.b; break; }
default: res.result = req.a + req.b; break; }
// 서비스 요청에 사용된 a, b 값의 표시 및 서비스 응답에 해당되는 result 값을 출력
ROS_INFO(“request: x=%ld, y=%ld”, (long int)req.a, (long int)req.b);
```

```
ROS_INFO("sending back response: [%ld]", (long int)res.result);
return true;
}
int main(int argc, char **argv) // 노드 메인 함수
{ ros::init(argc, argv, "service_server"); // 노드명 초기화
ros::NodeHandle nh; // ROS 시스템과 통신을 위한 노드 핸들 선언
nh.setParam("calculation_method", PLUS); // 매개변수 초기설정
// 서비스 서버 선언, ros_tutorials_service 패키지의 SrvTutorial 서비스 파일을 이용한
// 서비스 서버 service_server 를 작성. 서비스명은 "ros_tutorial_srv"이며,
// 서비스 요청이 있을 때, calculation 라는 함수를 실행하라는 설정
ros::ServiceServer ros_tutorial_service_server = nh.advertiseService("ros_tutorial_srv", calculation);
ROS_INFO("ready srv server!");
ros::Rate r(10); // 10 hz
while (1)
{ nh.getParam("calculation_method, g_operator); // 연산자를 매개변수로부터 받은 값으로 변경
ros::spinOnce(); // 콜백함수 처리루틴
r.sleep(); } // 루틴 반복을 위한 sleep 처리
return 0;
}
```

nh.setParam("calculation_method", Plus); : 파라미터 PLUS 로 초기설정.

nh.getParam("calculation_method", g_operator); : 변경값 받아 파라미터 변경.

2. 노드 빌드 및 실행

빌드 : `$cd ~/catkin_ws && catkin_make`

실행 : `$rosrun ros_tutorials_service service_server` -> 정상 작동 시 [INFO] [1495767130.149512649]: ready srv server! 출력.

3. 파라미터 목록 확인

```
$rosparam list
```

4. 파라미터 사용

```
$rosparam set /calculation_method 2
$rosservice call /ros_tutorial_srv 10 5
```

-> 정상 작동 시 뺄셈으로 result : 5

```
$rosparam set /calculation_method 3
$rosservice call /ros_tutorial_srv 10 5
```

-> 정상 작동 시 곱셈으로 result : 50

10.6. roslaunch 사용

rosrun 은 하나의 노드를 실행하는 명령어이다. 때문에 매우 많은 노드를 사용할 때 번거러움이 있다. 이를 해결할 수 있는 명령어가 바로 roslaunch 이다. roslaunch 는 하나 이상의 정해진 노드를 실행시킬 수 있다. roslaunch 는 '*.launch'라는 파일을 사용해 실행 노드를 설정하는데 이는 XML 기반이다. 실행 명령어는 "roslaunch [패키지명] [roslaunch 파일]"이다.

아래는 .launch 파일 생성 후 그 안에 내용의 몇 형식이다.

```
<!-- example -->
<launch>
  <arg name=""  default=""/>
  <rosparam command="" file=""/>

  <node pkg="" type="" name"" output="screen"/>

  <!--
  ...
  -->

</launch>
```

Arg 는 지역변수처럼 사용되는 값이다. Name 에 인자값 이름 지정, default 에 기본 인자값, 추가적으로 value 를 통해 인자값을 설정한다.

Rosparam 은 ROS 의 파라미터를 설정하는 것이다. Command 에는 rosparam 에서 사용하는 커맨드를 입력한다. 기본값은 load 이며 그 외 dump, delete 입력한다. file 에는 파일의 이름을 입력한다. 추가적으로 param 에는 파라미터의 이름을 지정한다.

Node 는 노드를 실행시키는 것이다. Pkg 는 node 가 들어있는 패키지, type 에는 노드 실행 파일 이름, name 은 노드의 이름을 지정한다(노드 내부의 이름보다 우선적으로 사용). Output 에는 screen 인 경우 노드의 stdout/stderr 를 화면에 출력하며, log 인 경우에는 stderr 를 화면에 출력한다.

11. ROS QT PACKAGE

ROS QT PACKAGE 란 ROS 와 QT 를 연결시켜 ROS 에서 QT 에 접근 가능하도록 지원하는 패키지이다. 패키지 생성을 위해선 설치를 먼저 해야 한다. 다음 GITHUB 사이트에서 README 를 읽으며 따라하며 설치한다.

https://github.com/mjlee111/catkin_create_qt_pkg

GITHUB 사이트에도 나와 있듯이 설치 후 package 생성 명령어는 catkin_create_qt_pkg [원하는 패키지 이름] [의존성] 이다. 이렇게 생성된 ROS QT 패키지 파일에는 기존 catkin_create_pkg 로 생성된 파일과 다른 점이 많을 것이다. ROS QT 패키지에서 우리가 주로 보는 것은 include 파일에 main_window.hpp 와 qnode.hpp, src 파일에서 main_window.cpp 와 qnode.cpp, ui 파일에서 main_window.ui 파일이다. Main_window.ui 에서 UI 를 디자인할 수 있고, main_window.cpp 에서 주로 UI 와 관련한 코드를 작성하며 main_window.hpp 에서 cpp 에 작성한 함수나 변수들에 대한 선언을 해준다. Qnode.cpp 에서는 메인 코드를 작성한다(주로 Topic pub, sub). hpp 에서는 cpp 에서 작성한 함수나 변수들을 선언해준다. 물론 앞서 설명한 것과 같이 각 파일에 코드 작성 위치가 정해진 것이 아닌 유동적으로 바뀔 수 있다.

아래 사진들을 보며 간단한 설명들을 해보겠다. 먼저 QNode.cpp 부분이다.

```cpp

/*****************************************************************************
** Includes
*****************************************************************************

#include <ros/ros.h>
#include <ros/network.h>
#include <string>
#include <std_msgs/String.h>
#include <sstream>
#include "../include/receiveudp/qnode.hpp"

/*****************************************************************************
** Namespaces
*****************************************************************************

namespace receiveudp
{
  /*****************************************************************************
  ** Implementation
  *****************************************************************************

  QNode::QNode(int argc, char** argv) : init_argc(argc), init_argv(argv)
  {
  }

  QNode::~QNode()
  {
    if (ros::isStarted())
    {
      ros::shutdown();  // explicitly needed since we use ros::start();
      ros::waitForShutdown();
    }
    wait();
  }
```

여기서 Qnod 에 대한 클래스들이 작동하는 형식이고 생성자 후 소멸자가 존재한다. 소멸자 진입 후 ROS 가 셧다운한다.

```
44  bool QNode::init()
45  {
46    ros::init(init_argc, init_argv, "receiveudp");
47    if (!ros::master::check())
48    {
49      return false;
50    }
51    ros::start();  // explicitly needed since our nodehandle is going out of 
52    ros::NodeHandle n;
53    // Add your ros communications here.
54    start();
55    return true;
56  }
57
58  void QNode::run()
59  {
60    ros::Rate loop_rate(33);
61    while (ros::ok())
62    {
63      ros::spinOnce();
64      loop_rate.sleep();
65    }
66    std::cout << "Ros shutdown, proceeding to close the gui." << std::endl;
67    Q_EMIT rosShutdown();  // used to signal the gui for a shutdown (useful t
68  }
69
70  }  // namespace s
```

그 다음으로 init 에선 말 그대로 Qnode 진입/시작을 알린다. 만약 여기서 true 로 함수를 무사히 빠져 나오면 아래 run 부분으로 들어간다. Rate loop_rate(33)은 반복 주기를 의미한다. 그리고 ros 상태가 ok 라면 while 문에 들어간다. 그렇게 들어간 후 사용자가 작성한 함수를 실행하거나 한 다음 spinOnce()부분 – 한 번만 반복 에 걸려 다시 run 과정을 반복하게 된다. 그러다가 사용자가 ros 를 끊는다면 while 문을 지나쳐 종료 문자를 보내며 rosshutdown SIGNAL(신호)를 발생한다. 이 신호로 인해 mainwindow.cpp 부분으로 가보자.

```
12  #include <QtGui>
13  #include <QMessageBox>
14  #include <iostream>
15  #include "../include/receiveudp/main_window.hpp"
16
17  /*****************************************************************************
18  ** Namespaces
19  *****************************************************************************/
20
21  namespace receiveudp
22  {
23    using namespace Qt;
24
25    /*****************************************************************************
26    ** Implementation [MainWindow]
27    *****************************************************************************/
28
29    MainWindow::MainWindow(int argc, char** argv, QWidget* parent) : QMainWindow(parent), qnode(argc, argv)
30    {
31      ui.setupUi(this);  // Calling this incidentally connects all ui's triggers to on_...() callbacks in this class.
32
33      setWindowIcon(QIcon(":/images/icon.png"));
34
35      qnode.init();
36
37      QObject::connect(&qnode, SIGNAL(rosShutdown()), this, SLOT(close()));
38    }
39
40    MainWindow::~MainWindow()
41    {
42    }
43
```

종료 신호를 받으면 mainwindow 클래스 내 QObject::connect(~)부분에서 SIGNAL 을 받은 뒤 SLOT(close())을 실행한다는 의미이다. 즉 close()함수로 Qt UI 까지 같이 꺼지게 되는 것이다. 이런 원리들로 ROS QT 가 실행되는 모습이다.