

Créer des applications ROS2

Dr. Eng. Chiheb Ameer ABID

[/in/chiheb-ameur-abid](https://www.linkedin.com/in/chiheb-ameur-abid)
 chiheb.abid@gmail.com

Janvier 2026

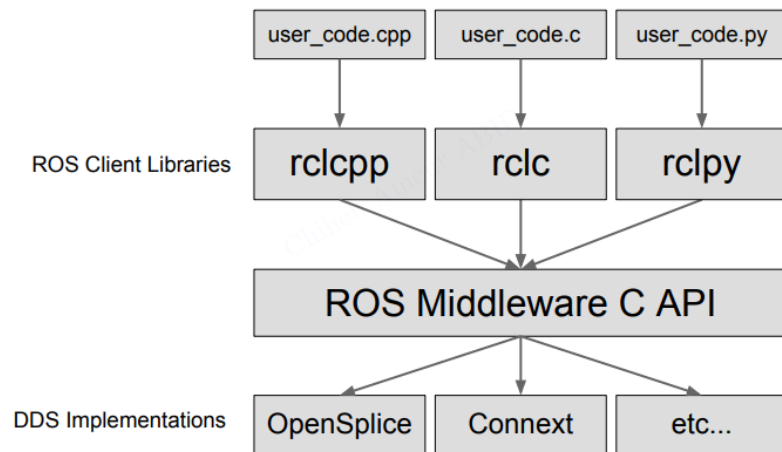
Chiheb Ameer ABID

3 / 86

Partie 2

Développement avec ROS

Architecture de ROS2



Plan

1 Développement avec ROS

- Aperçu sur l'outil CMake
- Création d'un nouveau package
- Développement d'un éditeur et d'un abonné
- Exécution des noeuds
- Messages de journalisation
- Interfaces de types

2 Les paramètres

3 Fichiers de lancement

Partie 2

Développement avec ROS

Chiheb Ameer ABID

4 / 86

ROS Client Library for C++ : rclcpp

Bibliothèque rclcpp (ROS Client Library for C++)

- rclcpp fournit l'API C++ canonique pour interagir avec ROS
- Développée en C++14
- Site de documentation officielle
<https://docs.ros2.org/humble/api/rclcpp/index.html>
- Toutes les fonctions et les classes sont définies dans l'espace de noms rclcpp
- Il se compose de ces composants principaux :
 - rclcpp::Node est le point d'entrée unique pour la création d'éditeurs et d'abonnés
 - rclcpp::Publisher modélise un éditeur
 - rclcpp::Subscription implémente la notion d'abonnement
 - rclcpp::Client modélise un client (un node) qui envoie une requête à un service et attend une réponse
 - rclcpp::Service est une classe template qui implémente la notion de service
 - rclcpp::Parameter de stocker un paramètre arbitraire avec des méthodes get/set templatisées. Un paramètre est une donnée qui peut être utilisée pour configurer ou modifier le comportement d'un node
 - rclcpp::WallTimer permet de créer un objet pour exécuter une fonction de rappel à intervalles réguliers ou après un délai spécifié.

Plan

1 Développement avec ROS

• Aperçu sur l'outil CMake

- Création d'un nouveau package
- Développement d'un éditeur et d'un abonné
- Exécution des noeuds
- Messages de journalisation
- Interfaces de types

Chiheb Ameur ABID

2 Les paramètres

3 Fichiers de lancement

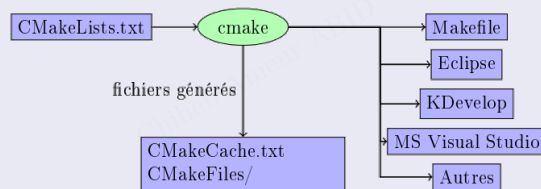
Partie 2
Chiheb Ameur ABID
7/ 86

- └ Développement avec ROS
- └ Aperçu sur l'outil CMake

CMake

Présentation

- ↳ Le fichier `CMakeLists.txt` définit les paramètres du projet et la démarche à suivre pour construire un projet



- ↳ Le fichier `CMakeCache.txt` définit un ensemble de variables de configuration et d'informations sur le système
- ↳ Le répertoire `CMakeFiles` contient les fichiers intermédiaires de compilation, des fichiers temporaires et autres fichiers de configuration qui ne regardent que CMake

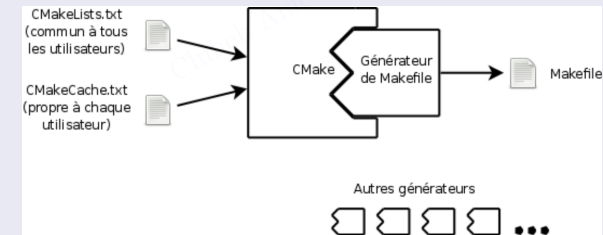
Partie 2
Chiheb Ameur ABID
7/ 86

- └ Développement avec ROS
- └ Aperçu sur l'outil CMake

Présentation de CMake

- ↳ CMake (Cross platform Make) est un outil qui permet de gérer la construction de logiciels sur différentes plateformes

- ↳ Développé par Kitware (VTK - Visualisation ToolKit) depuis 2001
- ↳ Un générateur de scripts de compilation, qui crée des fichiers adaptés au système et à la chaîne de compilation utilisés, à partir d'un fichier de configuration générique
- ↳ CMake peut produire des fichiers Makefile, des fichiers de projet Visual Studio, des fichiers de projet Code::Blocks, etc.



- ↳ Possède de nombreuses commandes permettant de localiser les dépendances

- ↳ Recherche de bibliothèques
- ↳ Facilite le portage ou la gestion de différents compilateurs

Partie 2
Chiheb Ameur ABID
8/ 86

- └ Développement avec ROS
- └ Aperçu sur l'outil CMake

Commandes de CMake

- ↳ CMake possède son propre langage

```

1 nom_de_la_commande(argument1 argument2 argument3 ... argumentN)
2 # Ceci est un commentaire
3 nom_d_une_autre_commande(argument " argument avec des espaces ")
  
```

- ↳ Nommer un projet

```
project(<projectname> [languageName1 languageName2 ... ])
```

- ↳ `projectname` : nom du projet
- ↳ `languageName1` : nom du langage de programmation utilisé dans le projet

- ↳ Ajouter un exécutable dans le projet

```
add_executable(<name> source1 source2 ... sourceN)
```

- ↳ `name` : nom de l'exécutable (cible)
- ↳ `source1...sourceN` : les fichiers sources permettant de construire l'exécutable

- ↳ Afficher des messages

```
message([STATUS|WARNING|AUTHOR_WARNING|FATAL_ERROR|SEND_ERROR ] "message" ...)
```

- ↳ Définir une valeur ou une liste de valeurs pour une variable

```
set(<variable> <value1> [<value2> ...])
```

Partie 2
Chiheb Ameur ABID
7/ 86

- └ Développement avec ROS
- └ Aperçu sur l'outil CMake

Partie 2
Chiheb Ameur ABID
8/ 86

- └ Développement avec ROS
- └ Aperçu sur l'outil CMake

Commandes de CMake

- Créer une bibliothèque statique ou partagée

```
add_library(<name> [STATIC | SHARED | MODULE] [EXCLUDE_FROM_ALL])
```

- ☞ Prend au moins deux arguments : le nom de la bibliothèque à créer et le type de bibliothèque

- Spécifier les bibliothèques ou les options de lien à utiliser lors de l'édition des liens

```
target_link_libraries(<target> <PRIVATE|PUBLIC|INTERFACE> <item1> ... <itemN>)
```

- ☞ Les dépendances PUBLIC sont visibles pour la cible actuelle et toutes les cibles qui dépendent de cette cible
- ☞ Les dépendances PRIVATE sont uniquement visibles pour la cible actuelle
- ☞ Les dépendances INTERFACE sont uniquement visibles pour les cibles qui dépendent de la cible actuelle, mais pas pour la cible elle-même

- Inclure un sous-répertoire dans le processus de construction

```
add_subdirectory(source_dir [binary_dir] [EXCLUDE_FROM_ALL] [SYSTEM])
```

- ☞ `binary_dir` : spécifie le répertoire où seront placés les fichiers de sortie
- ☞ `EXCLUDE_FROM_ALL` : le sous-répertoire sera exclu de la construction par défaut
- ☞ `SYSTEM` : les en-têtes du sous-répertoire seront traités comme des en-têtes système lors de la compilation

##ROS2

CMake

Commandes conditionnelles et boucles

- CMake offre des commandes pour écrire des instructions conditionnelles, des boucles ou d'itérer sur des listes

1 Commandes conditionnelles

- ☞ `if, endif`
- ☞ `elif, endif`

2 Boucles

- ☞ `while, endwhile`
- ☞ `foreach, endforeach`

##ROS2

CMake

Principaux types de variables

- CMake offre plusieurs types de variables

1 Variables d'environnement

- ☞ Définies dans l'environnement système avant l'exécution de CMake
- ☞ Peuvent être utilisées dans un script CMake via la commande `$ENV{VAR_NAME}`

2 Variables CMake

- ☞ Des variables définies directement dans le fichier `CMakeLists.txt` ou via la ligne de commande
- ☞ Locales au projet et ne sont pas accessibles en dehors de la session CMake

3 Variables prédéfinies

- ☞ Contiennent des informations sur l'environnement de build ou la configuration du projet

```
1 CMAKE_BINARY_DIR # Chemin de la racine de l'arborescence de construction
2 CMAKE_SOURCE_DIR # Chemin de la racine de l'arborescence des sources
3 CMAKE_INCLUDE_PATH # Chemins (séparés par ;) pour la recherche avec les commandes find_file() et find_path()
```

##ROS2

CMake

Installation

- CMake génère des fichiers de build qui contiennent les instructions pour installer un projet
- Pour contrôler quels fichiers sont installés, on utilise la commande `install` dans le fichier `CMakeLists.txt`

```
1 # Fichiers exécutables
2 install(TARGETS ...)
3 # Bibliothèques
4 install(TARGETS ... LIBRARY)
5 # Autres fichiers:
6 install(FILEs ...)
```

- Le contrôle de l'emplacement des fichiers installés

- ☞ `CMAKE_INSTALL_PREFIX` spécifie le préfixe d'installation pour tous les fichiers du projet
- ☞ `CMAKE_INSTALL_BINDIR` le répertoire d'installation par défaut pour fichiers exécutables (binaires)
- ☞ `CMAKE_INSTALL_LIBDIR` est le répertoire d'installation par défaut pour les fichiers de bibliothèque

##ROS2

CMake

Utilisation de CMake dans ROS2

- ↳ `ament_cmake` est un package qui fournit un ensemble de fonctions et de macros CMake pour la construction et la gestion de packages ROS2
- ↳ `ament_cmake` fournit les fonctionnalités suivantes :
 - Configuration des packages tels que le nom du package, la version du package, la description du package et les dépendances du package.
 - Construction des packages ROS2 tels que la création d'exécutables, de bibliothèques et de ressources
 - Gestion des packages ROS2 tels que l'installation des packages, la suppression des packages et la vérification des packages.

Plan

- 1 Développement avec ROS
 - Aperçu sur l'outil CMake
 - Création d'un nouveau package
 - Développement d'un éditeur et d'un abonné
 - Exécution des noeuds
 - Messages de journalisation
 - Interfaces de types
- 2 Les paramètres
- 3 Fichiers de lancement

Structure d'un fichier CMake pour un package ROS2

↳ La structure d'un fichier CMake pour un package ROS2

```

1 cmake_minimum_required(VERSION 3.5)
2 project(<project_name>)
3 find_package(ament_cmake REQUIRED)
4
5 ...
6
7 ament_package()

```

Principales commandes fournies par ament_cmake

↳ Définir les dépendances d'une cible

```
ament_target_dependencies(<target_name> <dependencies>)
```

- `target_name` est le nom de la cible
- `dependencies` est une liste de dépendances

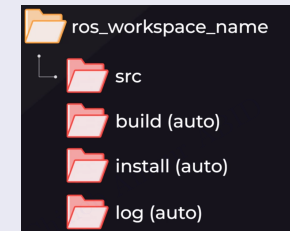
↳ Exporter les dépendances d'un paquet vers d'autres paquets

```
ament_export_dependencies(<package_name> <dependencies>)
```

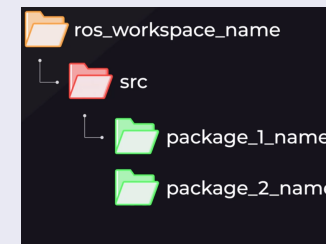
Structure d'un workspace

↳ Un **workspace** est un répertoire structuré qui permet d'organiser, compiler et gérer les paquets et dépendances d'un projet ROS 2

↳ Structure d'un workspace ROS2



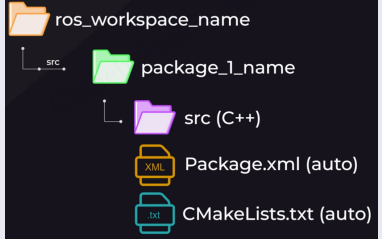
↳ L'emplacement des packages



Création d'un nouveau package

Structure d'un paquet

Structure d'un paquet dans un workspace ROS2



Création d'un nouveau package

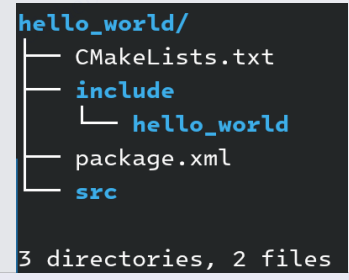
Créer un nouveau paquet

```
ros2 pkg create [package name] --dependencies [depend1] [depend2] [depend3] ...
```

- Par convention, le nom d'un package commence par une minuscule
- Il faut se placer dans le répertoire workspace/src

```
ros2 pkg create hello_world --dependencies rclcpp std_msgs
```

L'arborescence suivante est créée



Création d'un nouveau package

Un dossier est reconnu comme étant un package grâce à la présence du fichier package.xml

- C'est un fichier XML qui fournit les métadonnées sur le package, telles que le nom, la version, les auteurs, les mainteneurs et les dépendances sur d'autres packages ROS.

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd" schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>hello_world</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="chiheb.abid@fst.utm.tn">chiheb</maintainer>
  <license>TODO: License declaration</license>

  <buildtool_depend>ament_cmake</buildtool_depend>

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

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

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

Création d'un nouveau package

Création d'un nouveau package

Pour qu'il soit complet, le fichier package.xml doit contenir au moins les 5 champs suivants :

- <name> : le nom du paquet.
- <version> : le numéro de version du paquet.
- <description> : une description du contenu du paquet
- <maintainer> : le nom de la personne qui s'occupe de la maintenance du paquet ROS
- <license> : la license sous laquelle le code est publié (souvent la license BSD)

Développement d'un nœud minimal

Présentation

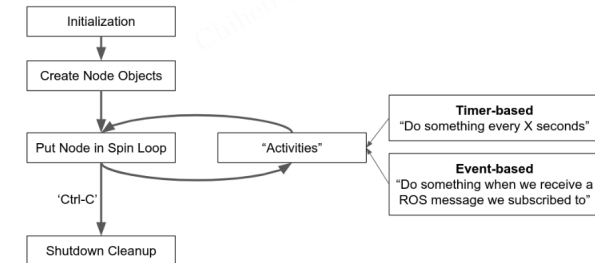
- Un nœud est une entité qui effectue des tâches spécifiques, comme publier ou souscrire à des topics, fournir ou appeler des services, ou gérer des paramètres
- Un nœud est encapsulé par la classe `rclcpp::Node`
- Il est recommandé qu'un nœud possède une responsabilité spécifique
- Le développement d'un nœud peut s'effectuer de deux manières
 - ❶ Par dérivation de la classe `rclcpp::Node`
 - ☛ Implémenter les fonctionnalités dans la classe dérivée
 - ❷ Par utilisation directe de la classe `rclcpp::Node`
 - ☛ Toutes les fonctionnalités sont configurées dans la fonction appelante
- Chaque nœud a sa propre file d'événements (event loop) pour gérer les callbacks et les événements associés : les messages reçus, les timers, les services, etc.

Développement d'un nœud minimal

Phase de cycle de vie d'un programme ROS2

- La mise en place d'un nœud dans un programme se fait en quatre étapes

- ❶ Initialiser le système ROS 2
- ❷ Création d'un ou de plusieurs nœuds
- ❸ Mettre le(s) nœud(s) en mode **écoute active**
- ❹ Nettoie les ressources allouées par ROS 2



Développement d'un nœud minimal

❶ Initialiser le système ROS 2

- Configurer le contexte ROS 2

- ☛ Préparer le système pour utiliser les fonctionnalités ROS 2

```
rclcpp::init(argc, argv);
```

- ☛ Prend en charge les arguments en ligne de commande (`argc`, `argv`)
- ☛ Doit être appelée avant toute autre fonction ROS 2

❷ Création d'un ou de plusieurs nœuds

- Les nœuds sont instanciés à partir de la classe `rclcpp::Node` ou d'une classe dérivée de `rclcpp::Node`
 - ☛ Il est recommandé d'utiliser des pointeurs intelligents `std::shared_ptr<>` pour gérer automatiquement la durée de vie du nœud et éviter les fuites de mémoire
 - ☛ Le nom du nœud doit être passé au constructeur pour identifier le nœud dans ROS 2

```

1 // Instanciation d'un nœud directement à partir rclcpp::Node
2 auto node {std::make_shared<rclcpp::Node>("mon_nœud")};
3
4 // Instanciation d'un nœud à partir d'une classe dérivée
5 class MonNoeud : public rclcpp::Node {
6 public:
7   MonNoeud() : Node("mon_nœud") {
8     // Initialisation spécifique du nœud ici
9   }
10  ...
11 };
12
13 auto node {std::make_shared<MonNoeud>()};
  
```



➤ Un nœud instancié n'est pas actif : les événements ne sont pas traités

③ Maintien du nœud actif (1/4)

↳ Maintenir le nœud en écoute active

- ☛ Traiter les événements (messages, timers, services, etc.)
- ☛ À chaque nœud, une file d'événements est associée

↳ Le maintien par la famille des fonctions `rclcpp::spin`

① Tourner un nœud de manière bloquante indéfiniment

```
void rclcpp::spin(std::shared_ptr<rclcpp::Node> node_ptr)
```

- ☛ Peut être interrompu par `Ctrl+\` ou `rclcpp::shutdown`

```
1 auto node {std::make_shared<rclcpp::Node>("mon_noeud")};
2 rclcpp::spin(node);
```

③ Maintien du nœud actif (2/4)

② Traite tous les événements disponibles dans la file d'attente au moment de l'appel, mais ne bloque pas pour attendre de nouveaux événements

```
void rclcpp::spin_some(std::shared_ptr<rclcpp::Node> node_ptr);
```

```
1 auto node {std::make_shared<rclcpp::Node>("mon_noeud")};
2 while (rclcpp::ok()) {
3   rclcpp::spin_some(node); // Traite les événements disponibles
4   // Autre logique ici
5   std::this_thread::sleep_for(std::chrono::milliseconds(100));
6 }
```



↳ `rclcpp::ok()` vérifie si le contexte de ROS2 est actif, et qui renvoie :

- ☛ `true` si le programme peut continuer à s'exécuter
- ☛ `false` si le contexte ROS 2 est fermé et le programme doit s'arrêter

```
bool rclcpp::ok();
```

③ Maintien du nœud actif (3/4)

③ Traiter un seul événement de la file d'attente et retourne immédiatement, avec une option de temporisation

- ☛ Si aucun événement n'est disponible dans le délai spécifié, elle ne bloque pas indéfiniment

```
void rclcpp::spin_once(std::shared_ptr<rclcpp::Node> node, std::chrono::duration
  sleep_timeout = std::chrono::nanoseconds(-1));
```

- ☛ `timeout` : Si négatif (par défaut), bloque jusqu'à ce qu'un événement soit disponible ou que le nœud soit arrêté. La valeur 0 rend la fonction non bloquante

```
1 auto node {std::make_shared<rclcpp::Node>("mon_noeud")};
2 while (rclcpp::ok()) {
3   rclcpp::spin_once(node, std::chrono::milliseconds(50));
4   // Autre logique ici
5   std::this_thread::sleep_for(std::chrono::milliseconds(100));
6 }
```

③ Maintien du nœud actif (4/4)

④ Traite tous les événements disponibles dans la file d'attente jusqu'à un timeout spécifié, ou jusqu'à épuisement des événements si aucun timeout n'est donné

```
void rclcpp::spin_all(rclcpp::Node::SharedPtr node, std::chrono::nanoseconds
  max_duration);
```

- ☛ Si `max_duration` est défini à 0, `spin_all()` continuera à traiter les événements jusqu'à ce qu'il n'y ait plus aucun événement prêt dans la file d'attente.

```
1 auto node {std::make_shared<rclcpp::Node>("mon_noeud")};
2 while (rclcpp::ok()) {
3   rclcpp::spin_all(node, std::chrono::seconds(1));
4   // Autre logique ici
5   std::this_thread::sleep_for(std::chrono::milliseconds(100));
6 }
```

Développement d'un nœud minimal

4 Arrêt de ROS 2

↳ Libérer les ressources allouées par ROS 2

```
rclcpp::shutdown()
```

- Après `rclcpp::shutdown()`, aucune fonction ROS ne peut être utilisée
- Éviter les fuites de mémoire et les comportements indéfinis



↳ Aucune fonctionnalité de ROS2 ne peut être exécutée après l'appel de `rclcpp::shutdown()`

Développement d'un nœud minimal

Développement d'un nœud minimal

↳ Le code source d'un nœud minimal

```
1 #include "rclcpp/rclcpp.hpp"
2
3 int main(int argc, char * argv[] {
4   rclcpp::init(argc, argv);
5   auto node {rclcpp::Node::make_shared("simple_node")}; // std::make_shared<rclcpp::
6     Node>("simple_node")
7   rclcpp::spin(node);
8   rclcpp::shutdown();
9   return 0;
10 }
```

- `rclcpp::init()` permet de transmettre des arguments depuis la ligne de commandes au nœud
- `rclcpp::Node::make_shared()` crée un nœud et retourne un pointeur intelligent sur une instance de type `rclcpp::Node`
- `spin()` lancer la fonction callback du nœud
- `shutdown()` libère tous les ressources associées au nœud

Développement d'un nœud minimal

Compilation

↳ Pour effectuer la compilation, il est nécessaire d'indiquer dans le fichier CMakefile les dépendances et l'exécutable

```
....
# find dependencies
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)

#####
set(dependencies rclcpp)
add_executable(simple src/simple.cpp)
ament_target_dependencies(simple ${dependencies})

install(TARGETS simple
  DESTINATION lib/${PROJECT_NAME})

#####
if(BUILD_TESTING)
...

```

Développement d'un nœud minimal

Compilation

- ↳ La compilation des packages d'un workspace s'effectue en utilisant l'outil `colcon`
- ↳ On doit se placer dans le répertoire du workspace
 - Re-compiler tous les packages du workspace

```
colcon build --symlink-install
```

- ↳ Compiler uniquement un package

```
colcon build --packages-select hello --symlink-install
```



↳ L'option `--symlink-install` a pour but de créer des liens symboliques vers les fichiers construits dans `build/` plutôt que de les copier directement dans `install/`

- ✓ Pendant le développement : facilite les tests et modifications
- ⊗ Sur un système de production : mieux vaut éviter pour garantir l'intégrité des fichiers installés

Développement d'un nœud minimal

Exécution

- ↳ On ajoute le chemin du workspace dans les variables d'environnement

```
source install/local_setup.bash
```

- ↳ Exécuter le nœud développer

```
ros2 run hello simple
```

Plan

- 1 Développement avec ROS
 - Aperçu sur l'outil CMake
 - Création d'un nouveau package
 - Développement d'un éditeur et d'un abonné
 - Exécution des noeuds
 - Messages de journalisation
 - Interfaces de types

2 Les paramètres

3 Fichiers de lancement

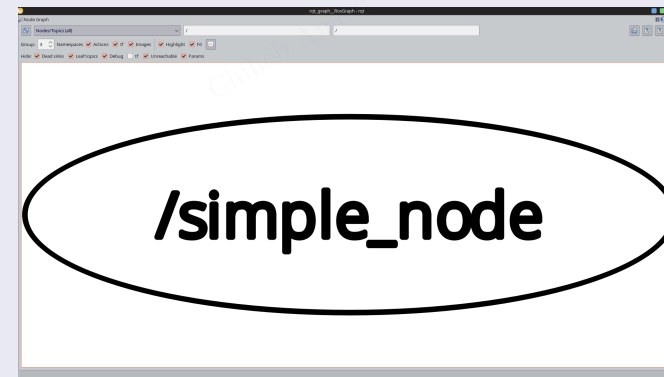
Développement d'un nœud minimal

Exécution

- ↳ Visualiser la liste des nœuds depuis la CLI

```
1 $ ros2 node list
2 /simple_node
```

- ↳ Visualiser graphiquement les nœuds avec l'outil rqt_graph



Principales méthodes de la classe `rclcpp::Node`

- ↳ Constructeur

```
Node(const std::string &node_name, const NodeOptions &options = NodeOptions())
```

- `node_name` : nom du nœud à créer
- `options` : les options telles que les arguments de ligne de commande, les paramètres initiaux, les QoS, les groupes de rappel et la communication intra-processus

- ↳ Retourner le nom du module

```
const char* get_name() const
```

- ↳ Obtenir le logger du nœud, qui peut être utilisé pour enregistrer des messages de différents niveaux de sévérité

```
rclcpp::Logger get_logger() const
```

- ↳ Créer et retourner un timer qui peut exécuter une fonction de rappel à intervalles réguliers

```
rclcpp::WallTimer<CallbackT>::SharedPtr create_wall_timer(std::chrono::duration<
DurationRepT, DurationT> period, CallbackT callback, rclcpp::CallbackGroup::
SharedPtr group=nullptr)
```

La classe `rclcpp::QoS`

Présentation de la classe `rclcpp::QoS`

- La classe `rclcpp::QoS` est une classe qui permet de configurer les paramètres de qualité de service (QoS) des éditeurs et des abonnés
 - Les QoS sont des politiques qui définissent la qualité de la communication entre les entités ROS2, telles que la fiabilité, la durée de vie, la vivacité et le délai des messages
- Les QoS supportés par `rclcpp::QoS`
 - La politique d'historique, qui détermine le nombre de messages que le middleware ROS2 peut stocker pour les éditeurs et les abonnés. Elle peut être de type "keep last", qui ne conserve que les N derniers messages, ou de type "keep all", qui conserve tous les messages, sous réserve des limites de ressources du middleware
 - La politique de profondeur, qui spécifie la taille de la file d'attente des messages. Elle n'est prise en compte que si la politique d'historique est de type "keep last"
 - La politique de fiabilité, qui définit le niveau de garantie de livraison des messages aux abonnés. Elle peut être de type "best effort", qui n'assure pas que tous les messages sont livrés et peut les abandonner en cas de perte ou de congestion, ou de type "reliable", qui assure que tous les messages sont livrés et peut les retransmettre en cas de perte

##ROS2

La classe `rclcpp::QoS`

Principales méthodes de la classe `rclcpp::QoS` (1/3)

- Construction en spécifiant


```
QoS(size_t history_depth);
```

 - `history_depth` spécifie le nombre de messages que le middleware ROS2 peut stocker pour les éditeurs et les abonnés
- Renvoie le profil QoS sous-jacent utilisé par le middleware ROS2. Il s'agit d'une structure qui contient les valeurs des différentes politiques QoS, comme la fiabilité, la durabilité, le délai, la vivacité et la profondeur d'historique


```
const rmw_qos_profile_t& get_rmw_qos_profile() const
```

##ROS2

La classe `rclcpp::QoS`

Présentation de la classe `rclcpp::QoS`

- Les QoS supportés par `rclcpp::QoS` (SUITE)
 - La politique de durabilité, qui détermine si le middleware ROS2 conserve les messages publiés par les éditeurs, même s'ils n'ont pas encore d'abonnés, et s'il les envoie aux nouveaux abonnés lorsqu'ils se connectent. Elle peut être de type "volatile", qui ne conserve pas les messages, ou de type "transient local", qui conserve les messages
 - La politique de délai, qui spécifie le délai maximal entre la publication d'un message et sa livraison aux abonnés. Elle peut être utilisée pour garantir que les messages sont livrés dans un temps
 - La politique de vivacité, qui définit le niveau d'activité attendu des éditeurs et des abonnés. Elle peut être utilisée pour détecter les entités inactives ou déconnectées
 - La politique de durée de vie, qui détermine la durée pendant laquelle un message reste valide. Elle peut être utilisée pour éviter que les abonnés reçoivent des messages obsolètes

##ROS2

La classe `rclcpp::QoS`

Principales méthodes de la classe `rclcpp::QoS` (2/3)

- Modifier la politique de fiabilité
 - Mode fiable
 - ROS2 garantit que tous les messages sont livrés aux abonnés, quitte à les retransmettre en cas de perte
 - Politique restrictive

```
QoS& reliable()
```
 - Mode meilleur effort
 - Le middleware ROS2 ne garantit pas que tous les messages sont livrés aux abonnés, et qu'il peut les abandonner en cas de congestion ou de perte
 - Mode est la politique la moins restrictive

```
QoS& best_effort()
```

##ROS2

La classe `rclcpp::QoS`

Principales méthodes de la classe `rclcpp::QoS` (3/3)

- ↳ Modifier la politique de durabilité
- ① Mode transitoire local
 - ▣ ROS2 conserve les messages, même s'ils ne sont pas encore abonnés, et qu'il les envoie aux nouveaux abonnés lorsqu'ils se connectent
 - ▣ Politique restrictive

```
QoS& transient_local()
```
- ② Mode volatile
 - ▣ ROS2 ne conserve pas les messages, et qu'il ne les envoie qu'aux abonnés existants au moment de la publication
 - ▣ Politique la moins restrictive

```
QoS& durability_volatile()
```

La classe `rclcpp::QoS`

Exemple de création d'une configuration QoS

```
auto qos {rclcpp::QoS(100).transient_local().best_effort()};
```

- ↳ La configuration QoS créée possède les caractéristiques suivantes :
 - ▣ Profondeur de l'historique = 100
 - ▣ Durabilité : mode transitoire local
 - ▣ Fiabilité : mode meilleur effort

Compatibilité des configuration QoS entre un éditeur et un abonné

Compatibilité des configuration QoS entre un éditeur et un abonné

- ↳ Chaque éditeur spécifie sa qualité de service, et chaque éditeur peut également spécifier sa qualité de service
 - ▣ Il existe des QoS qui ne sont pas compatibles, ce qui empêchera l'abonné de recevoir des messages
 - ▣ Compatibilité des profils de durabilité

Compatibility of QoS durability profiles		Subscriber	
		Volatile	Transient Local
Publisher	Volatile	Volatile	No Connection
	Transient Local	Volatile	Transient Local

- ▣ Compatibilité des profils de fiabilité

Compatibility of QoS reliability profiles		Subscriber	
		Best Effort	Reliable
Publisher	Best Effort	Best Effort	No Connection
	Reliable	Best Effort	Reliable

Compatibilité des configuration QoS entre un éditeur et un abonné

Règle pour assurer la compatibilité des configuration QoS entre un éditeur et un abonné



↳ L'éditeur doit spécifier la politique la moins restrictive

Implémentation d'un noeud éditeur

Méthode de création d'un éditeur

- Créer un éditeur (publisher) qui publie des messages sur un topic

```
std::shared_ptr<rclcpp::Publisher<MessageT>> create_publisher(const std::string&
    topic_name, const rclcpp::QoS & qos)
```

- Retourne un pointeur intelligent sur l'éditeur créé `rclcpp::Publisher<MessageT>` où `MessageT` est le type de message pour le topic
- `topic_name` spécifie le nom du topic sur lequel publier
- `qos` spécifie la qualité de service définissant la politique de transmission du message

Implémentation d'un noeud éditeur

La classe `rclcpp::Publisher<T>`

- `rclcpp::Publisher<T>` est une classe modèle modélisant un éditeur défini pour un topic
 - Le type `T` définit le type des messages pour le topic
- Principales méthode de `rclcpp::Publisher<T>`

```
1 void publish(const MessageT &msg); // Publier le message msg
2 const char *get_topic_name () const; // Renvoie le topic sur lequel publier
3 size_t get_subscription_count () const; // Renvoie le nombre d'abonnés au topic
4 rclcpp::QoS get_actual_qos () const; // Renvoie l'actuelle QoS
5 size_t get_queue_size () const; // Renvoie la taille de la file associée à l'
    éditeur
```

Implémentation d'un noeud éditeur

Exemple d'implémentation d'un noeud éditeur

- On procède par dérivation
 - Dans le constructeur, on crée l'éditeur
 - On crée un timer pour exécuter une fonction de rappel
 - On définit la fonction de rappel pour l'envoi des messages

```
1 class MinimalPublisher : public rclcpp::Node {
2 public:
3     MinimalPublisher():Node("minimal_publisher"), count_(0) {
4         publisher_=create_publisher<std_msgs::msg::String>("topic",10);
5         timer_=create_wall_timer(500ms, std::bind(&MinimalPublisher::timer_callback, this));
6     }
7
8 private:
9     void timer_callback() {
10         auto message = std_msgs::msg::String();
11         message.data = "Hello, world! " + std::to_string(count_++);
12         RCLCPP_INFO(this->get_logger(), "Publishing: '%s'", message.data.c_str());
13         publisher_->publish(message);
14     }
15     rclcpp::TimerBase::SharedPtr timer_;
16     rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
17     size_t count_;
18 };
```

Implémentation d'un noeud abonné

Implémentation d'un noeud abonné

- Création d'un abonné

```
1 std::shared_ptr<SubscriptionT> rclcpp::Node::create_subscription(
2 const std::string &topic_name,
3 const rclcpp::QoS & qos,
4 CallbackT&& callback,
5 const SubscriptionOptionsWithAllocator<AllocatorT>& options
6 =SubscriptionOptionsWithAllocator<AllocatorT>(),
7 typename MessageMemoryStrategyT::SharedPtr msg_mem_strat
8 =(MessageMemoryStrategyT::create_default())
9 )
```

- `topic_name` est le topic
- `qos` est le profil QoS
- `callback` est la fonction de rappel

```
void callback(std::shared_ptr<SubscriptionT> msg)
```

Implémentation d'un noeud abonné

Exemple d'un noeud abonné

```

1 #include <memory>
2 #include "rclcpp/rclcpp.hpp"
3 #include "std_msgs/msg/string.hpp"
4 using std::placeholders::_1;
5
6 class MinimalSubscriber : public rclcpp::Node
7 {
8 public:
9     MinimalSubscriber():Node("minimal_subscriber") {
10         subscription_ = create_subscription<std_msgs::msg::String>(
11             "topic", 10, std::bind(&MinimalSubscriber::topic_callback, this, _1));
12     }
13
14 private:
15     void topic_callback(const std_msgs::msg::String::SharedPtr msg) const {
16         RCLCPP_INFO(get_logger(), "I heard: '%s'", msg->data.c_str());
17     }
18     rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription_;
19 };
20
21 int main(int argc, char * argv[]) {
22     rclcpp::init(argc, argv);
23     rclcpp::spin(std::make_shared<MinimalSubscriber>());
24     rclcpp::shutdown();
25     return 0;
26 }

```

##ROS2

Exécution des noeuds

- la bibliothèque rclcpp (ROS Client Library for C++) fournit plusieurs variantes de la fonction `rclcpp::spin()` pour gérer l'exécution des noeuds et le traitement des callbacks
- Tourner un noeud de manière bloquante indéfiniment

```
void rclcpp::spin(std_shared_ptr<rclcpp::Node> node);
```

```

1 auto node {std::make_shared<rclcpp::Node>("mon_noeud")};
2 rclcpp::spin(node);
3 rclcpp::shutdown();

```

##ROS2

Plan

1 Développement avec ROS

- Aperçu sur l'outil CMake
- Création d'un nouveau package
- Développement d'un éditeur et d'un abonné
- Exécution des noeuds
- Messages de journalisation
- Interfaces de types

2 Les paramètres

3 Fichiers de lancement

##ROS2

Exécution des noeuds

- Traite tous les événements disponibles dans la file d'attente au moment de l'appel, mais ne bloque pas pour attendre de nouveaux événements

```
void rclcpp::spin_some(std::shared_ptr<rclcpp::Node> node_ptr);
```

```

1 auto node {std::make_shared<rclcpp::Node>("mon_noeud")};
2 while (rclcpp::ok()) {
3     rclcpp::spin_some(node); // Traite les événements disponibles
4     // Autre logique ici
5     std::this_thread::sleep_for(std::chrono::milliseconds(100));
6 }

```

##ROS2

Exécution des noeuds

- Traite un seul événement de la file d'attente (le prochain disponible) et retourne immédiatement, avec une option de temporisation. Si aucun événement n'est disponible dans le délai spécifié, elle ne bloque pas indéfiniment.

```
void rclcpp::spin_once(std::shared_ptr<rclcpp::Node> node, std::chrono::nanoseconds timeout = std::chrono::nanoseconds(-1));
```

- **timeout** : Si négatif (par défaut), bloque jusqu'à ce qu'un événement soit disponible ou que le noeud soit arrêté.

```
1 auto node {std::make_shared<rclcpp::Node>("mon_noeud")};
2 while (rclcpp::ok()) {
3   rclcpp::spin_once(node, std::chrono::milliseconds(50));
4   // Autre logique ici
5   std::this_thread::sleep_for(std::chrono::milliseconds(100));
6 }
```

Exécution des noeuds

- Traite tous les événements disponibles dans la file d'attente jusqu'à un timeout spécifié, ou jusqu'à épuisement des événements si aucun timeout n'est donné. Elle est bloquante pendant cette période.

```
void rclcpp::spin_all(rclcpp::Node::SharedPtr node, std::chrono::nanoseconds max_duration);
```

```
1 auto node = std::make_shared<rclcpp::Node>("mon_noeud");
2 while (rclcpp::ok()) {
3   rclcpp::spin_all(node, std::chrono::seconds(1));
4   // Autre logique ici
5   std::this_thread::sleep_for(std::chrono::milliseconds(100));
6 }
```

#ROS2
Partie 2
Chiheb Ameur ABID
55 / 86

- └ Développement avec ROS
- └ Messages de journalisation

Plan

1 Développement avec ROS

- Aperçu sur l'outil CMake
- Création d'un nouveau package
- Développement d'un éditeur et d'un abonné
- Exécution des noeuds
- Messages de journalisation
- Interfaces de types

2 Les paramètres

3 Fichiers de lancement

#ROS2
Partie 2
Chiheb Ameur ABID
56 / 86

- └ Développement avec ROS
- └ Messages de journalisation

Messages de journalisation (logs)

Présentation

- Le journalisation est un mécanisme permettant aux noeuds d'enregistrer des messages à des fins de diagnostic ou d'analyse
 - ☛ Par défaut, ROS2 enregistre les fichiers logs dans le dossier `./ros/log`
- les logs sont organisés en niveaux de sévérité. Les niveaux de sévérité sont les suivants :
 - ☛ **DEBUG** : Informations détaillées qui peuvent être utiles pour le débogage.
 - ☛ **INFO** : Informations générales sur l'exécution du système.
 - ☛ **WARN** : Informations sur des événements qui peuvent être problématiques.
 - ☛ **ERROR** : Informations sur des événements qui indiquent un problème.
 - ☛ **FATAL** : Informations sur des événements qui indiquent une erreur critique.

Messages de journalisation (logs)

Présentation

- Par défaut, les messages de log dans les noeuds ROS 2 seront envoyés à :
 - La console (sur stderr)
 - Aux fichiers de log sur disque
 - Au topic `/rosout` sur le réseau ROS 2
- Toutes les cibles peuvent être activées ou désactivées individuellement pour chaque noeud.



- Les logs peuvent consommer des ressources, telles que la mémoire et le CPU
- Il est important de configurer les logs correctement pour éviter d'encombrer le système de logs

Messages de journalisation (logs)

Contrôle par les variables d'environnement

- Certains aspects de la journalisation sont contrôlés à travers les variables d'environnement
 - Les valeurs d'environnement s'appliquent à tous les noeuds d'un processus
- `ROS_LOG_DIR` spécifier l'emplacement de la journalisation
- `RCUTILS_LOGGING_USE_STDOUT` : la valeur 0 pour utiliser `stderr`, et 1 pour utiliser `stdout`

Messages de journalisation (logs)

Contrôle par le paramétrage d'un noeud

- Lors de lancement d'un noeud, il est possible de contrôler certains aspects de la journalisation
- `log_level` spécifie le niveau de journalisation


```
ros2 run demo_nodes_cpp talker --ros-args --log-level talker:=DEBUG
```
- `log_config_file` définit le fichier de journalisation


```
ros2 run demo_nodes_cpp talker --ros-args --log-config-file log-config.txt
```
- `log_stdout_disabled` désactiver la journalisation dans la console


```
ros2 run demo_nodes_cpp talker --ros-args --disable-stdout-logs
```
- `log_rosout_disabled` : désactiver l'envoi des messages de journalisation au topic `/rosout`

```
ros2 run demo_nodes_cpp talker --ros-args --disable-rosout-logs
```

L'outil `rqt_console`

- `rqt_console` est un outil graphique utilisé pour l'inspection des messages de journalisation
 - Affichage des messages de log par niveau de sévérité, par logger, ou par sujet.
 - Filtrage des messages de log en fonction de leur niveau de sévérité, de leur logger, ou de leur sujet.
 - Recherche de messages de log spécifiques.
 - Exportation des messages de log vers un fichier.
- Lancer l'outil `rqt_console`

```
ros2 run rqt_console rqt_console
```



Messages de journalisation (logs)

Générer des messages de journalisation depuis un noeud

- Obtenir un logger en spécifiant un préfixe pour les messages

```
rclcpp::Logger rclcpp::get_logger(const string &name)
```

- Obtenir un logger associé à une instance de `rclcpp::Node`

```
rclcpp::Logger rclcpp::get_logger() const
```

- Les macros pour générer différents types de messages

- `RCLCPP_INFO(logger, message)` génère un message de log d'information.
- `RCLCPP_DEBUG(logger, message)` génère un message de log de débogage.
- `RCLCPP_WARN(logger, message)` génère un message de log d'avertissement.
- `RCLCPP_ERROR(logger, message)` génère un message de log d'erreur.
- `RCLCPP_FATAL(logger, message)` génère un message de log fatal, qui entraîne l'arrêt du programme.

Partie 2 Chiheb Ameur ABID 63 / 86

Interface de types

Interfaces de types

Présentation

- Les interfaces de types sont des descriptions de structures des données échangées entre les noeuds
 - Elles permettent de garantir la compatibilité entre les noeuds qui communiquent entre eux.
- Il existe trois types d'interfaces de types :
 - Les **messages** sont utilisés pour échanger des données simples, telles que des chaînes de caractères ou des nombres.
 - Les **services** sont utilisés pour effectuer des tâches simples, telles que lire ou écrire un fichier
 - Les **actions** sont utilisées pour effectuer des tâches complexes, telles que la navigation d'un robot.
- Les interfaces de types sont définies dans un langage de description de langage appelé IDL (Interface Definition Language)
 - ROS 2 fournit un outil appelé `rosidl` qui peut être utilisé pour générer du code à partir des définitions d'interface de types

Plan

- Développement avec ROS
 - Aperçu sur l'outil CMake
 - Création d'un nouveau package
 - Développement d'un éditeur et d'un abonné
 - Exécution des noeuds
 - Messages de journalisation
 - Interfaces de types
- Les paramètres
- Fichiers de lancement

Partie 2 Chiheb Ameur ABID 64 / 86

Interface de types

Interfaces de types

Manipuler les interfaces depuis la CLI

- La commande CLI pour examiner les interfaces de types disponibles

```
ros2 interface <command>
```

- Lister toutes les interfaces de types disponibles

```
ros2 interface list
```

- Lister les interfaces de types d'un package

```
ros2 interface package <package_name>
```

- Lister les packages qui définissent des interfaces de types

```
ros2 interface packages
```

- Affiche le prototype d'une interface

```
ros2 interface proto <type>
```

- Afficher la définition IDL d'une interface

```
ros2 interface show <type>
```

Interfaces de types

Les interfaces prédéfinies dans ROS2

- ROS2 offre plusieurs packages dans les quelles des interfaces sont prédéfinies https://github.com/ros2/common_interfaces
 - `std_msgs` contient une collection de types de messages courants pour représenter des types de données primitifs et d'autres constructions de messages de base
 - `sensor_msgs` contient des message types pour représenter les données de capteurs, telles que des images, des scans laser, des données de localisation, etc.
 - `geometry_msgs` fournit des message types pour représenter des données géométriques, telles que des points, des vecteurs, des quaternions, des matrices de transformation, etc.
 - `tf2_msgs` fournit des message types pour représenter les données de transformation, qui sont utilisées pour convertir les coordonnées entre différents systèmes de coordonnées.

Interfaces de types

Création d'une interface personnalisée

- Une interface personnalisée pour un message est définie dans un fichier avec l'extension `.msg` dans le dossier `msg/` du package
- La syntaxe de déclaration de l'interface dans un fichier `.msg`

```
1 fieldtype1 fieldname1
2 fieldtype2 fieldname2
3 fieldtype3 fieldname3
```

Exemple

```
1 int32 my_int
2 string my_string
```

Type name	C++
bool	bool
byte	uint8_t
char	char
float32	float
float64	double
int8	int8_t
uint8	uint8_t
int16	int16_t
uint16	uint16_t
int32	int32_t
uint32	uint32_t
int64	int64_t
uint64	uint64_t
string	std::string
wstring	std::wstring

Interfaces de types

Interfaces de types

Compilation d'une interface personnalisée

- Dans le fichier `package.xml` du package

```
1 <buildtool_depend>rosidl_default_generators</buildtool_depend>
2 <exec_depend>rosidl_default_runtime</exec_depend>
3 <member_of_group>rosidl_interface_packages</member_of_group>
```

- Dans le fichier `CMakeLists.txt` du package

```
1 #Trouver le package pour générer le code
2 find_package(rosidl_default_generators REQUIRED)
3
4 #Inclure le fichier de l'interface personnalisée
5 set(msg_files
6 "msg/My_msg_file.msg"
7 )
8
9 #Générer le code avec rosidl
10 rosidl_generate_interfaces(${PROJECT_NAME}
11 ${msg_files}
12 )
13
14 #Exporter la dépendance
15 ament_export_dependencies(rosidl_default_runtime)
```

Utiliser une interface personnalisée

- Une fois le code de l'interface est généré, il devient possible de l'utiliser dans un programme C++

```
1 #include "rclcpp/rclcpp.hpp"
2 #include "my_package/msg/my_msg_file.hpp" // .hpp commence par une minuscule
3
4
5 class MyNode : public rclcpp::Node {
6     ...
7     void myFunc() {
8         // Le nom de la classe de l'interface commence par une majuscule
9         auto instance_msg = my_package::msg::My_msg_file();
10    }
11    ...
12};
```

- Il faut intégrer la dépendance à travers `CMakeLists.txt`

```
1 rosidl_get_typesupport_target(cpp_typesupport_target
2 ${PROJECT_NAME} rosidl_typesupport_cpp)
3
4 target_link_libraries(my_package "${cpp_typesupport_target}")
```

Interfaces de types

Exemple illustratif (1/4)

↳ Développer un noeud éditeur envoyant dans le sujet `mon_sujet` un message de type personnalisé

❶ Créer le package `custom_interfaces`

```
ros2 pkg create custom_interfaces --dependencies rclcpp std_msgs
```

❷ Créer le fichier `Customer.msg`, dans le dossier `custom_interfaces/msg` contenant la description IDL suivante :

```
1 string name
2 uint8 age
```

❸ Ouvrir le fichier `package.xml` et ajouter les lignes suivantes :

```
1 <buildtool_depend>rosidl_default_generators</buildtool_depend>
2 <exec_depend>rosidl_default_runtime</exec_depend>
3 <member_of_group>rosidl_interface_packages</member_of_group>
```

Interfaces de types

Exemple illustratif (2/4)

❹ Ouvrir le fichier `CMakefile.txt` et ajouter les lignes suivantes :

```
1 find_package(rosidl_default_generators REQUIRED)
2
3 set(msg_files "msg/Custom.msg")
4
5 rosidl_generate_interfaces(${PROJECT_NAME}
6   ${msg_files}
7 )
8
9 ament_export_dependencies(rosidl_default_runtime)
```

❺ Compiler le package pour générer le code C++ pour les interfaces

```
~/workspace_ros2$ colcon build --symlink-install
```

Partie 2 Chiheb Ameur ABID 71/ 86

- ↳ Développement avec ROS
- ↳ Interfaces de types

Interfaces de types

Exemple illustratif (3/4)

❻ Implémenter le noeud dans le fichier `CustomPub.cpp`

```
1 #include <chrono>
2 #include <memory>
3 #include "rclcpp/rclcpp.hpp"
4 #include "custom_interfaces/msg/customer.hpp"
5 using namespace std::chrono_literals;
6
7 class CustomPub : public rclcpp::Node {
8 public:
9   CustomPub(): Node("customer_publisher") {
10     c_publisher=create_publisher<custom_interfaces::msg::Customer>("mon_sujet",10);
11     auto publish_msg = [this]() -> void {
12       auto message = custom_interfaces::msg::Customer();
13       message.name = "John";
14       message.age = 25;
15       std::cout << "Publishing Contact\nFirst:" << message.name <<
16         " Age:" << message.age << std::endl;
17       c_publisher->publish(message);
18     };
19     timer_ = create_wall_timer(1s, publish_msg);
20   }
21 private:
22   rclcpp::Publisher<custom_interfaces::msg::Customer>::SharedPtr c_publisher_;
23   rclcpp::TimerBase::SharedPtr timer_;
24 };
```

Partie 2 Chiheb Ameur ABID 72/ 86

- ↳ Développement avec ROS
- ↳ Interfaces de types

Interfaces de types

Exemple illustratif (4/4)

❻ Implémenter le noeud dans le fichier `CustomPub.cpp` (Suite)

```
1 int main(int argc, char * argv[]) {
2   rclcpp::init(argc, argv);
3   rclcpp::spin(std::make_shared<CustomPub>());
4   rclcpp::shutdown();
5   return 0;
6 }
```

❼ Modifier le fichier `CMakefile.txt` pour la compilation finale

```
1 set(dependencies rclcpp)
2 rosidl_get_typesupport_target(cpp_typesupport_target "${PROJECT_NAME}"
3   "rosidl_typesupport_cpp")
4
5 add_executable(custompub src/custompub.cpp)
6 ament_target_dependencies(custompub ${dependencies})
7
8
9 target_link_libraries(custompub "${cpp_typesupport_target}")
10
11 install(TARGETS
12   custompub
13   DESTINATION lib/${PROJECT_NAME}
14 )
```

Plan

- 1 Développement avec ROS
- 2 Les paramètres
- 3 Fichiers de lancement

Chiheb Ameur ABID

Les paramètres

Présentation

- Les paramètres sont des valeurs que les noeuds peuvent utiliser pour configurer leur comportement.
- Ils sont associés à un noeud spécifique et leur durée de vie est liée à la durée de vie du noeud
- Les paramètres peuvent être utilisés pour configurer une grande variété d'aspects d'un noeud, tels que :
 - ☞ Les valeurs par défaut des variables
 - ☞ Les ports de communication
 - ☞ Les paramètres de performance
 - ☞ Les paramètres de sécurité
- Les paramètres peuvent être définis de deux façons :
 - ☞ En déclarant des paramètres dans le fichier de configuration du noeud
 - ☞ En passant des paramètres en ligne de commande au noeud au démarrage

Les paramètres

Les paramètres

Les commandes CLI pour gérer les paramètres (1/2)

- Syntaxe de la commande pour gérer les paramètres

```
ros2 param <command>
```

- ☞ Consulter la liste des paramètres

```
ros2 param list [/node_name]
```

- ☞ Modifier la valeur d'un paramètre

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

- ☞ Renvoyer la valeur d'un paramètre

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

- ☞ Supprimer un paramètre

```
ros2 param delete </node_name> <parameter_name>
```

Les commandes CLI pour gérer les paramètres (2/2)

- Syntaxe de la commande pour gérer les paramètres

```
ros2 param <command>
```

- ☞ Afficher la description d'un paramètre

```
ros2 param describe </node_name> <parameter_name>
```

- ☞ Enregistrer la liste des paramètres d'un noeud dans un fichier au format YAML. Le fichier peut être utilisé pour charger les paramètres pour re-exécuter le noeud

```
ros2 param dump </node_name>
```

- ☞ Charger les valeurs des paramètres à partir d'un fichier au format YAML pour un noeud donné

```
ros2 param load </node_name> <parameter_file>
```

Les paramètres

La classe `rclcpp::Parameter`

➤ À chaque paramètre un objet de type `rclcpp::Parameter` est associé

➤ Principales méthodes

☞ Constructeur

```
1 rclcpp::Parameter::Parameter(const std::string& name,
2 const ParameterValue &value)
```

☞ Récupérer le type du paramètre

```
1 ParameterType get_type() const
```

☞ Récupérer le nom du paramètre enregistré dans l'objet

```
1 const std::string& get_name() const
```

☞ Récupérer la valeur du paramètre

```
1 template<ParameterType ParamT>
2 decltype(auto) get_value() const
```

Les paramètres

Méthodes pour gérer les paramètres

➤ Déclarer un paramètre

```
1 template<typename ParameterT>
2 auto rclcpp::Node::declare_parameter(const std::string& name,
3 const ParameterT& default_value)
```

☞ `name` est le nom du paramètre

☞ `default_value` spécifie la valeur par défaut

☞ Renvoie une copie de la valeur spécifique au paramètre

➤ Récupérer la valeur d'un paramètre

```
1 bool rclcpp::Node::get_parameter(const std::string& name, ParameterT& parameter)
   const
```

☞ `name` est le nom du paramètre

☞ `parameter` contiendra la valeur courante du paramètre

☞ Renvoie `true` si OK, sinon `false` et peut générer une exception

Les paramètres

Les paramètres

Méthodes pour gérer les paramètres

➤ Récupérer une instance, de la classe `Parameter`, associée à un paramètre

```
1 bool rclcpp::Node::get_parameter(const std::string& name, rclcpp::Parameter&
   parameter) const
```

☞ `name` est le nom du paramètre

☞ `parameter` est une référence sur l'instance représentant le paramètre

☞ Renvoie `true` si OK, sinon `false` et peut générer un exception

➤ Récupérer une instance, de la classe `Parameter`, associée à un paramètre

```
1 rclcpp::Parameter rclcpp::Node::get_parameter(const std::string& name ) const
```

☞ `name` est le nom du paramètre

☞ Renvoie un objet de type `Parameter` associé au paramètre

☞ Peut générer une exception

Exemple de gestion des paramètres

```
1 class MyNode : public rclcpp::Node {
2 public:
3   MyNode():Node("my_node") {
4     declare_parameter<double>("my_param", 1.0);
5     ...
6   }
7   ...
8 private:
9   void my_func() {
10    double param_value;
11    get_parameter("my_param", param_value);
12    ...
13  }
14};
```

Plan

- 1 Développement avec ROS
- 2 Les paramètres
- 3 Fichiers de lancement

Chiheb Ameur ABID

Fichiers de lancement

Présentation

- ↳ Les fichiers de lancement (launch files) sont des fichiers de configuration qui permettent de démarrer et de configurer plusieurs noeuds simultanément
 - ↳ Ils sont écrits en Python, et utilisent des fichiers XML ou YAML, pour décrire les actions à effectuer
- ↳ Créer des configurations personnalisées
- ↳ Ajouter de la logique au démarrage d'un système
 - ↳ Vérification des dépendances entre les noeuds
 - ↳ Attente de l'achèvement d'une tâche avant de démarrer un autre noeud
- ↳ Exemples d'utilisation des fichiers de lancement :
 - ↳ Démarrer un ensemble complet de noeuds pour une application
 - ↳ Configurer les paramètres d'un noeud
 - ↳ Ajouter de la logique au démarrage d'un système
 - ↳ Automatiser le déploiement

Partie 2
Chiheb Ameur ABID
83 / 86

Fichiers de lancement

Fichiers de lancement

Partie 2
Chiheb Ameur ABID
84 / 86

Fichiers de lancement

Fichiers de lancement

Création d'un script de lancement en Python

- ↳ Le fichier de lancement (.py) doit être enregistré dans le répertoire launch/ du package
 - ↳ Par convention le fichier est suffixé par .launch.py
- ↳ La squelette d'un fichier de lancement Python

```

1 from launch import LaunchDescription
2 from launch_ros.actions import Node
3
4 def generate_launch_description():
5     return LaunchDescription([
6         Node(
7             package="my_package",
8             executable="mynode_name",
9             parameters=[{"param1":value1},...]
10        ),
11        Node(
12            ...
13        ),
14        ...
15    ])

```

Création d'un script de lancement en Python

- ↳ Exécuter une commande shell depuis un fichier de lancement

```

1 from launch.actions import ExecuteProcess
2
3 ExecuteProcess(cmd=["command", "arg1", "arg2", ...], output="screen")

```

Fichiers de lancement

Exécution d'un fichier de lancement

↳ Intégrer le fichier du lancement dans CMakefile.txt

```
1 install(DIRECTORY launch
2   DESTINATION share/${PROJECT_NAME}/
3 )
```

↳ Lancer un fichier de lancement depuis la CLI

☛ Aller dans le dossier launch/

```
ros2 launch <launch_file_name>
```

☛ Depuis la racine de l'espace de travail

```
ros2 launch <package_name> <launch_file_name>
```

Merci pour votre attention



Questions ?