Développement d'un noeud minimal en C++ avec ROS2

Dr. Ing. Chiheb Ameur ABID

Contact : chiheb.abid@gmail.com

Janvier 2025

Développement d'un noeud minimal

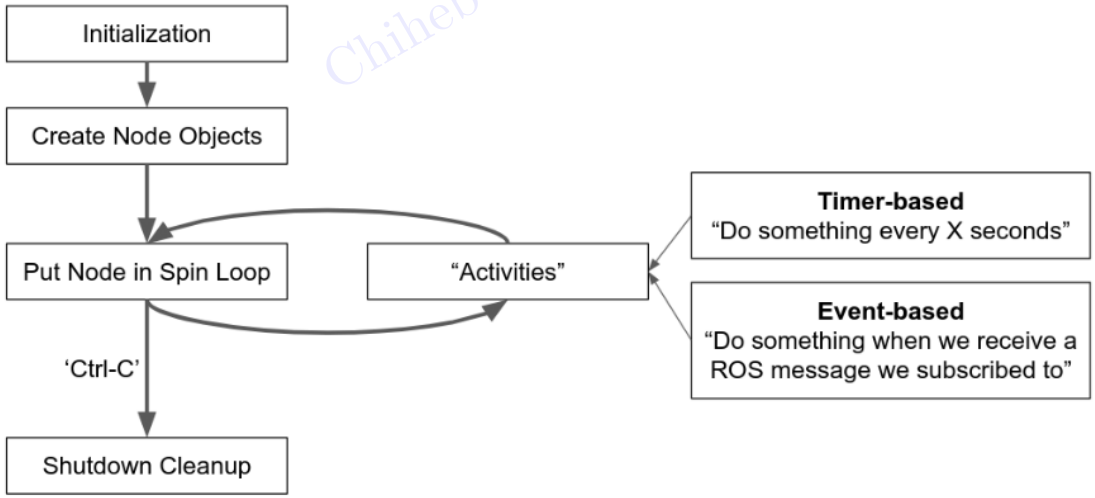
Présentation

- Un noeud 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 noeud est encapsulé par la classe `rclcpp::Node`
- Il est recommandé qu'un noeud possède une responsabilité spécifique
- Le développement d'un noeud peut s'effectuer de deux manières
  1. Par dérivation de la classe `rclcpp::Node`
  - ☞ Implémenter les fonctionnalités dans la classe dérivée
  2. Par utilisation directe de la classe `rclcpp::Node`
  - ☞ Toutes les fonctionnalités sont configurées dans la fonction appelante
- Chaque noeud 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 noeud minimal

Phase de cycle de vie d'un programme ROS2

- La mise en place d'un noeud dans un programme se fait en quatre étapes
  1. Initialiser le système ROS 2
  2. Création d'un ou de plusieurs noeuds
  3. Mettre le (les) noeud(s) en mode **écoute active**
  4. Nettoie les ressources allouées par ROS 2



Développement d'un noeud 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

❷ Création d'un ou de plusieurs noeuds

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

```cpp
// Instanciation d'un noeud directement à partir rclcpp::Node
auto node {std::make_shared<rclcpp::Node>("mon_noeud")};

// Instanciation d'un noeud à partir d'une classe dérivée
class MonNoeud : public rclcpp::Node {
public:
  MonNoeud() : Node("mon_noeud") {
    // Initialisation spécifique du noeud ici
  }
  ...
};

auto node {std::make_shared<MonNoeud>()};
```


- Un noeud instancié n'est pas actif : les évènements ne sont pas traités

❸ Maintien du noeud actif (1/4)

- Maintenir le noeud en écoute active
  - Traiter les événements (messages, timers, services, etc.)
  - À chaque noeud, une file d'évènements est associée
- Le maintien par la famille des fonctions **`rclcpp::spin`**

❶ Tourner un noeud de manière bloquante indéfiniment

```
void rclcpp::spin(std::shard_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);
```

❸ Maintien du noeud actif (2/4)

❷ 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::shard_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();
```

❸ Maintien du noeud actif (3/4)

❸ 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 noeud 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 }
```

❸ Maintien du noeud actif (4/4)

❹ 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 noeud minimal

❹ 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 noeud minimal

Développement d'un noeud minimal

➥ Le code source d'un noeud 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");
6    rclcpp::spin(node);
7    rclcpp::shutdown();
8    return 0;
9  }
```


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

Développement d'un noeud 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 noeud 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 noeud minimal

Exécution

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

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

- Exécuter le noeud développer

```
$ ros2 run hello simple
```

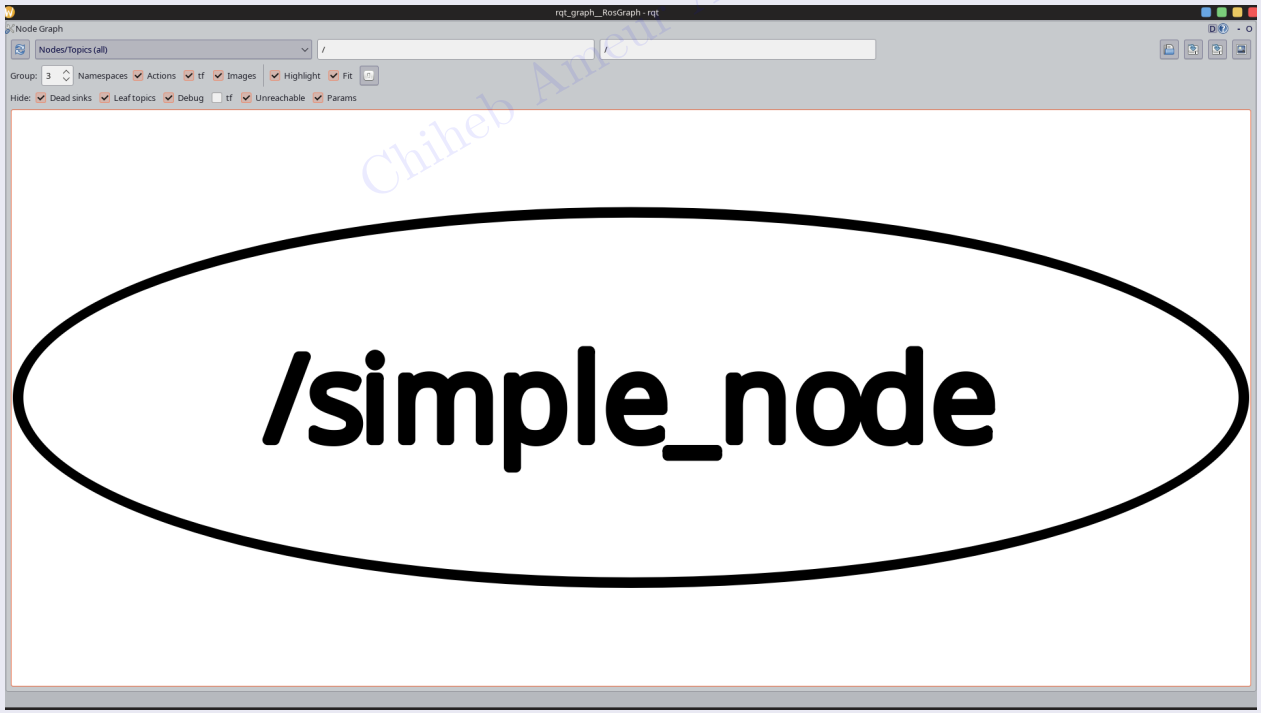
Développement d'un noeud minimal

Exécution

- Visualiser la liste des noeuds depuis la CLI

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

- Visualiser graphiquement les noeuds avec l'outil `rqt_graph`



Merci pour votre attention