

ROS 2

Een uitgebreide introductie

Jorn Bunk



HAN UNIVERSITY OF APPLIED SCIENCES

DIT DOCUMENT IS GEMAAKT VOOR STUDENTEN VAN AIM VAN DE HOGESCHOOL ARNHEM EN NIJMEGEN. DIT DOCUMENT IS GELICENTIEERD ONDER DE *CREATIVE COMMONS ATTRIBUTION-SHAREALIKE 4.0 INTERNATIONAL (CC BY-SA 4.0)* LICENTIE.

www.han.nl

Dit document is tot stand gekomen door bijdrages en feedback van:

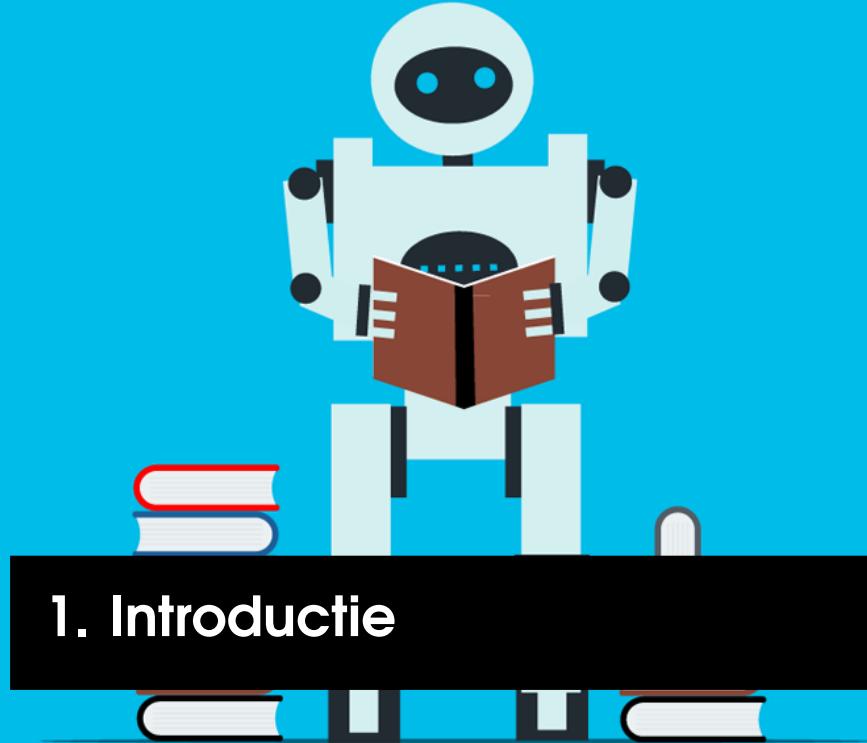
Jorn Bunk	Hogeschooldocent ESD	Auteur
Wilrick Bakker	Student HAN	Feedback
Erik Brilleman	Student HAN	Feedback
Sandra ter Maat	Student HAN	Feedback
Daan Smienk	Student HU	Feedback
Sua Wilhelm	Student HU	Feedback
Huib Aldewereld		LAT _E X-template



Inhoudsopgave

1	Introductie	5
1.1	Installatie en gebruik	6
2	Nodes	7
2.1	RCLCPP_INFO	9
2.2	rclcpp::TimerBase::SharedPtr	9
2.3	HelloNode op jouw computer	10
2.4	ROS communicatie	10
3	Topics	11
3.1	Publisher code	11
3.2	Subscriber code	13
3.3	Message types	15
3.4	Publisher revisited	16
4	Services	19
4.1	Request and response structure	20
4.2	Server code	20
4.3	Client code	23
4.4	Meer bronnen	26
5	Actions	27
5.1	Action message structure	27

5.2	Action Server	29
5.3	Action client	34
5.4	Meer bronnen:	38
6	ROS tools en etc.	41
6.1	Handige ROS commando's	41
6.2	Achtergrond	42
7	ROS packages	45
7.1	Tf2	45
7.2	Rviz	51
8	Practicum	53
	Appendices	59
A	HelloNode package	61
B	Frequently Asked Questions	65
B.1	ROS?!	65
B.2	Arduino?!	66



1. Introductie

Met deze reader verkrijgt de lezer een stevige fundatie voor het gebruik van ROS 2 aan de hand van concrete codevoorbeelden. ROS is een open source software framework en komt met veel tools voor het programmeren, het compileren van code, het testen en het simuleren van hardware. Een belangrijke eigenschap van ROS is dat het de communicatie verzorgt tussen de verschillende processen (“robot onderdelen”). ROS maakt het hiermee mogelijk software te schrijven voor één onderdeel zonder dat men zich zorgen hoeft te maken over elk klein detail van de andere onderdelen. De flexibiliteit en herbruikbaarheid van code en hardware wordt hiermee vergroot. Een in ROS geprogrammeerde ultra-sound sensor kan zowel werken op de Pepper als op de Terminator¹. Het is dan ook niet gek dat de meeste robot(onderdelen)fabrikanten software leveren die werkt met ROS. Hiermee kan hun hardware flexibel worden gebruikt in de meest uiteenlopende projecten. Daarnaast zorgt de open-source-community van ROS dat vele libraries beschikbaar zijn.

Figuur 1.1
De robot Pepper
(Foto door Tokumei-gakarinoaoshima)



De voordelen van ROS zijn goed zichtbaar bij een robot zoals Pepper (zie figuur 1.1). Deze heeft verschillende sensoren (microfoon, touchscreen, camera's, LiDar) en meerdere actuatoren (wielen, robotarmen, scherm, etc.) die met elkaar moeten samenwerken en

¹De aannname is dat Arnold Schwarzenegger draait met ROS (<https://www.imdb.com/title/tt0088247/>).

communiceren. Elke sensor en actuator heeft zijn eigen (programmeer)uitdagingen die we graag, voor zo ver mogelijk, per onderdeel aanpakken. Verder willen we graag flexibel zijn in welke sensoren en actuatoren de Pepper gebruikt. ROS helpt ons hierbij. Het doel van ROS is het verzorgen van een standaard voor robot software development dat resulteert in software dat werkt op elke robot.

De reader start bij de basis, een simpele ROS-node, en gaat vervolgens in op de drie manieren van communicatie in ROS 2. Naast de codevoorbeelden in de reader is er nog meer codevoorbeelden beschikbaar op de git. Voordat men zijn/haar eigen code met ROS gaat schrijven raden wij sterk aan om hoofdstuk 2 en 3 van deze reader te lezen en de stappen van appendix A te doorlopen.

Deze reader is ontwikkeld ter ondersteuning van de cursus *World* van de Hogeschool Arnhem en Nijmegen. De reader bevat de belangrijkste functionaliteiten van ROS 2, maar heeft een focus op de C++-code. Meer informatie over ROS 2 kan men vinden in de documentatie van ROS 2:

<https://docs.ros.org/en/foxy/>

Voordat de lezer vol enthousiasme de reader in duikt raden wij aan om eerst ROS 2 te installeren.

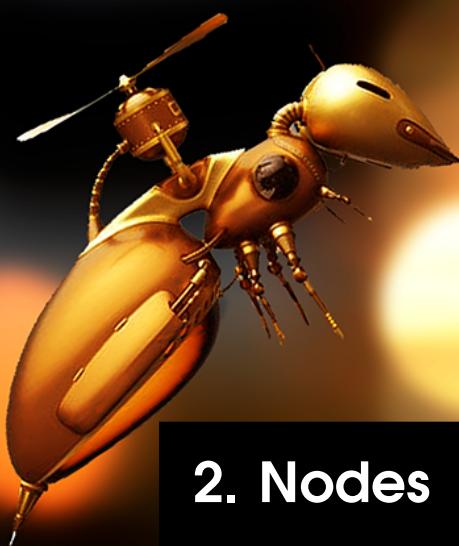
1.1 Installatie en gebruik

Voor de installatie van ROS verwijzen we naar de tutorial van ROS. De auteur heeft gebruik gemaakt van ROS 2 Foxy Fitzroy op Linux.

<https://docs.ros.org/en/foxy/Installation.html>

Met ROS 2 maken we gebruik van de library *rclcpp*. De documentatie hiervan is te vinden op:

<http://docs.ros2.org/latest/api/rclcpp/>



2. Nodes

Dit hoofdstuk gaat in op C++code voor het maken van een ROS-node. De stappen voor het maken van een ROS-package en het runnen van de node uit dit hoofdstuk zijn te vinden in appendix A.

De basis van een ROS-programma zijn de nodes. Elke node is een proces en vaak verantwoordelijke voor één taak of onderdeel. Dit kunnen dus processen zijn die hardware aansturen, maar kunnen ook processen zijn die sensordata verwerken (bijvoorbeeld beeldherkenning), de business logic afhandelen (de “AI”) of berekeningen doen die nodig zijn voor het aansturen van andere processen. In dit hoofdstuk bekijken wat, qua code, een node precies is en hoe we het gebruiken.

In codevoorbeeld 2.1 zien we het maken van de meest simpele node en het starten van deze node. Deze node doet niks, maar we zien hier wel dat we de node een shared pointer moeten maken, zodat vervolgens ROS het met de commando `rclcpp::spin()` er een thread van kan maken.

```
0 // import the ROS2 core lib:
1 #include "rclcpp/rclcpp.hpp"
2
3 int main(int argc, char **argv)
4 {
5     // start ROS2:
6     rclcpp::init(argc, argv);
7     // create a node with the name "my_node_name" and
8     // make it a shared pointer
9     auto node = std::make_shared<rclcpp::Node>("my_node_name");
10    // run the node (until you kill it (ctrl-c)):
11    rclcpp::spin(node);
12    // stop ROS2:
13    rclcpp::shutdown();
14    return 0;
15 }
```

Codevoorbeeld 2.1

Het maken en runnen van een minimalistische ROS-Node.

Als we onze eigen ROSnode willen definiëren dan doen we dat door een subclass te maken van de `rclcpp::Node`. De initialisatie van deze subclass moet natuurlijk ook in een

sharedpointer, zodat we er een thread binnen ROS van kunnen maken met het commando `rclcpp::spin()`. In codevoorbeeld 2.2, 2.3 en 2.4 zien we dit uitgewerkt. De bestanden zijn netjes gesplitst in een een header-file, source file en de main die de node opstart. Buiten bestaan en zijn superclass aanmaken doet de node van de codevoorbeelden nog niks. Let op: De constructor van de superclass `rclcpp::Node` heeft als parameter de naam van de node. Elke node moet een unieke naam hebben.

```

0 #include "rclcpp/rclcpp.hpp"
1
2 // create our own Node as a subclass of Node:
3 class EmptyNode : public rclcpp::Node{
4 public:
5     // constructor is the only mandatory function:
6     EmptyNode();
7 };

```

Codevoorbeeld 2.2
EmptyNode.hpp

```

0 #include "rclcpp/rclcpp.hpp"
1 #include "EmptyNode.hpp"
2
3 // the constructor initialized the super class Node:
4 EmptyNode::EmptyNode() : Node("EmptyNode"){ }

```

Codevoorbeeld 2.3
EmptyNode.cpp

```

0 #include "rclcpp/rclcpp.hpp"
1 #include "EmptyNode.hpp"
2
3 int main(int argc, char **argv){
4     rclcpp::init(argc, argv);
5
6     // create a EmptyNode and
7     // make it a shared pointer:
8     auto node = std::make_shared<EmptyNode>();
9
10    rclcpp::spin(node);
11    rclcpp::shutdown();
12    return 0;
13 }

```

Codevoorbeeld 2.4
main.cpp

In codevoorbeelden 2.5, 2.6 en 2.7 zien we een node die wel wat doet. De node `HelloNode` print achter elkaar “Hello World!”. Hiervoor gebruiken we de functie `timerCallback()` die “Hello World!” print en een timer die deze functie aanroept. De class maakt hiervoor gebruik van een ROS2 timer die we kunnen krijgen van de functie `create_wall_timer()` van de superclass `rclcpp::Node`. We ontleden de code onder de codevoorbeelden.

```

0 #include "rclcpp/rclcpp.hpp"
1
2 class HelloNode : public rclcpp::Node
3 {
4 public:
5     HelloNode();
6
7 private:
8     // the function to be called by the timer:
9     void timerCallback();

```

```

10
11     // the timer:
12     rclcpp::TimerBase::SharedPtr timer_;
13 }

```

Codevoorbeeld 2.5
HelloNode.hpp

```

0 #include "rclcpp/rclcpp.hpp"
1 #include "HelloNode.hpp"
2
3 HelloNode::HelloNode() : Node("HelloNode"){
4     //initialisation of the timer:
5     timer_ = this->create_wall_timer(
6         std::chrono::milliseconds(200),
7         std::bind(&HelloNode::timerCallback, this));
8 }
9
10 void HelloNode::timerCallback(){
11     // print "Hello World!" in the logger:
12     RCLCPP_INFO(this->get_logger(), "Hello World!");
13 }

```

Codevoorbeeld 2.6
HelloNode.cpp

```

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

```

Codevoorbeeld 2.7
main.cpp

2.1 RCLCPP_INFO

De functie *RCLCPP_INFO()* maakt het mogelijk om berichten te loggen. De functie verwacht een logger en een string om te loggen. In codevoorbeeld 2.6 gebruiken we de logger die we erven van de superclass en loggen we de string "Hello World!". De output van de logger zien we op het moment dat we de node starten.

2.2 rclcpp::TimerBase::SharedPtr

Met *rclcpp::TimerBase::SharedPtr* definiëren we een ROS2 timer object. Deze gebruiken we om de node op bepaalde intervallen een functie uit te laten voeren. In codevoorbeeld 2.6 is de member *timer_* een ROS2 timer object. In de constructor initialiseren we *timer_* met de geïnfde functie *create_wall_timer()*. Deze functie verwacht twee argumenten: de tijd tussen het aanroepen van de functie en de functie die moet worden aangeroepen. Voor de tijd gebruiken we *std::chrono::milliseconds()*² met de waarde 200. De functie gaat dus elke 200ms aangeroepen worden. Om de class methode mee te kunnen geven moeten we gebruik maken van *std::bind*³.

²Voor meer informatie zie: <https://en.cppreference.com/w/cpp/chrono>

³Een uitleg over *std::bind* kan men vinden op: <https://www.youtube.com/watch?v=JtUZmkvRoKg>

2.3 HelloNode op jouw computer

In Appendix A staan de stappen die moeten worden doorlopen om van HelloNode.cpp een werkend voorbeeld te maken.

2.4 ROS communicatie

De communicatie tussen de verschillende nodes/processen⁴ kan in ROS op drie manieren: via *services*, via *topics* of via *actions*. Een node neemt deel aan de communicatie door één of meerdere communicatierollen aan te nemen. Bij topics kennen we de communicatierollen *publishers* en *subscribers*, bij services hebben we *servers* en *clients* en bij actions hebben we *action servers* en *action clients*. Een node neemt een communicatierol aan door binnen haar class een member variabele te declareren en te initialiseren van het type dat hoort bij de communicatierol. Een class mag meerdere members hebben en de programmeur is dus ook vrij om meerdere communicatierollen te geven aan een node. Zo kan een node zowel een publisher en action client zijn of een subscriber zijn op twee verschillende topics en publisher op een ander topic.

In de hoofdstukken 3, 4 en 5 leggen we uit wat de verschillende manieren van communiceren en de bijbehorende rollen omvatten. Per communicatiemethode en rol geven we een voorbeeld in C++⁵.

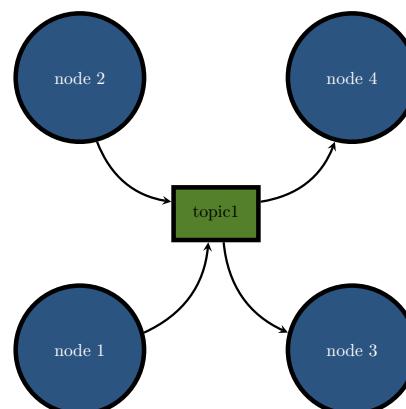
⁴Dit kunnen processen zijn die hardware aansturen, maar kunnen ook processen zijn die sensordata verwerken (bijvoorbeeld beeldherkenning), de Business logic afhandelen (de “AI”) of berekeningen doen die nodig zijn voor het aansturen van andere processen.

⁵Deze voorbeelden (en meer) zijn ook te vinden op OnderwijsOnline. De ROS documentatie geeft ons ook een uitleg op: <https://index.ros.org/doc/ros2/Tutorials/Writing-A-Simple-Cpp-Publisher-And-Subscriber/>

3. Topics

Het communiceren via topics in ROS gaat via een *publisher/subscriber*-systeem. Elke node heeft de mogelijkheid om een *topic* aan te maken. Nodes die informatie willen delen kunnen op een topic hun berichten *publiceren*. Een node die berichten publiceert op een topic noemen we een *publisher*. Nodes die deze informatie willen *subscribe*n zichzelf op de topic en krijgen zo de berichten binnen. Een node die berichten ontvangt van een topic noemen we een *subscriber*. Het kan zijn dat een node zowel een publisher is als een subscriber. Meestal publiceert de node dan wel op een ander topic dan waarop hij is gesubscribed. Nodes kunnen ook op meerdere topics publisheren of op meerdere topics gesubscribed zijn.

Met de topic-communicatiemethode ligt het initiatief van de communicatie bij de publisher. Elke bericht dat op de topic wordt gepubliceerd wordt naar alle subscribers gestuurd. De subscribers koppelen een functie aan een topic. Bij elk bericht wordt deze functie aangeroepen om het bericht te verwerken. In figuur 3.1 zien we dit uitgebeeld. Meerdere nodes kunnen tegelijkertijd gesubscribed zijn op een topic en meerdere nodes kunnen tegelijkertijd publishen op een topic.



Figuur 3.1
Een ROS topic.
Node 1 en node 2
publiceren naar de
topic. Node 3 en 4
zijn gesubscribed
tot het topic en
ontvangen
daarmee de
berichten van
node 1 en node 2.

3.1 Publisher code

Een node noemen we een publisher als het een membervariable heeft van het type `rclcpp::Publisher<T>`. Een `rclcpp::Publisher<T>` is template-class die het message-type

mee moet krijgen. ROS komt met een aantal standaard messages-types⁶, maar het is ook mogelijk om je eigen message-type aan te maken. We gaan dieper in op de verschillende message-types in sectie 3.3. Het declareren van een `rclcpp::Publisher<T>` doen we binnen een node met `rclcpp::Publisher<message_type>::SharedPtr publisher_name`. Bijvoorbeeld de publisher `publisher_` van het standaard message type String:

```
0 rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
```

Code voorbeeld 3.1

Een publisher met het message type `std_msgs::msg::String`.

Een `rclcpp::Publisher` kunnen we initialiseren door middel van de functie `create_publisher<T>()` die nodes erven van de superclass `rclcpp::Node`. Deze functie heeft nodig: het message type, de naam van het topic en de lengte van de messages queue. Bijvoorbeeld de `rclcpp::Publisher<T> publisher_` die Strings publischt op de topic met de naam `topic` en een queue van 10:

```
0 publisher_ = this->create_publisher<std_msgs::msg::String>("topic", 10);
```

Voordat we een bericht kunnen publiceren, moeten we een message maken. Dit doen we door eerst een message te declareren en te initialiseren en vervolgens daar data aan toe te voegen. In het geval van onze standard string message gaat dat als volgt:

```
0 auto message = std_msgs::msg::String();
1 message.data = "Hello, world!";
```

Het sturen van een message gaat vervolgens met de `publish()`-functie van de publisher:

```
0 publisher_->publish(message)
```

Met de boven genoemde elementen kunnen we een publisher maken. In codevoorbeelden 3.2, 3.3 en 3.4 gebruiken we deze elementen om een publisher te maken die met behulp van een timer⁷ elke 500 ms een bericht publiceert.

```
0 #include "rclcpp/rclcpp.hpp"
1 #include "std_msgs/msg/string.hpp"
2
3 class PublisherNode : public rclcpp::Node
4 {
5 public:
6     PublisherNode();
7
8 private:
9     // function to be called by the timer.
10    void timer_callback();
11
12    // the timer:
13    rclcpp::TimerBase::SharedPtr timer_;
14
15    // publisher with message type std_msgs::msg::String :
16    rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
17    // variabele to count the number of messages we have sent:
18    size_t count_;
19};
```

Code voorbeeld 3.2

PublisherNode.hpp

⁶Deze moet je wel includeren!

⁷Voor meer informatie over de timer zie hoofdstuk 2.

```

0 #include "rclcpp/rclcpp.hpp"
1 #include "std_msgs/msg/string.hpp"
2 #include "PublisherNode.h"
3
4 PublisherNode::PublisherNode(): Node("PublisherNode"), count_(0){
5     // initializing the publisher using the create_publisher function of the parent:
6     // "topic" the name of the topic
7     // 10: the size of the queue
8     publisher_ = this->create_publisher<std_msgs::msg::String>("topic", 10);
9
10    // initializing the timer and binding it with the function timer_callback():
11    timer_ = this->create_wall_timer(
12        std::chrono::milliseconds(500),
13        std::bind(&PublisherNode::timer_callback, this)
14    );
15}
16
17 void PublisherNode::timer_callback(){
18     // a new message:
19     auto message = std_msgs::msg::String();
20     // adding data:
21     count_++;
22     message.data = "Hello, world! " + std::to_string(count_);
23     // publishing the message on the topic:
24     publisher_->publish(message);
25}

```

Code voorbeeld 3.3
PublisherNode.cpp

```

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

```

Code voorbeeld 3.4
mainPublisher.cpp

3.2 Subscriber code

Een node noemen we een *subscriber* als het een membervariabele van het type *rclcpp::Subscription*<*T*> heeft. Een *rclcpp::Subscription*<*T*> is template-class die het message-type van de topic waarop we willen subscriben mee moet krijgen. Over message types vertellen we meer in sectie 3.3. Het declareren van een publisher-variabele doen we binnen de node met *rclcpp::Subscription*<*message_type*>::*SharedPtr subscription_name*; Bijvoorbeeld de subscription *subscription_* van het standaard message type String:

```
0 rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription_;
```

Een subscriber reageert altijd op elk bericht dat geplaatst wordt op het topic waartoe hij gesubscribed is. We definiëren de actie die de subscriber moet uitvoeren door bij het initialiseren van de *rclcpp::Subscription*<*T*> een functie aan de subscription te binden.

Deze functie wordt vervolgens bij elk gepubliceerd bericht aangeroepen⁸. Het initialiseren van een subscription doen we met de functie `create_subscription<T>()` die een node erft van de superclass `rclcpp::Node`. Deze functie heeft nodig: het message type, de naam van het topic, lengte van de messages queue en de functie die hij moet aanroepen als er een bericht wordt gepubliceerd. In het geval van `subscription_` in de node `MyNode` met het standaard message type `String`, de topic met de naam `topic`, een queue van 10 en het aanroepen van de functie `topic_callback()` doen we dat met:

```
0 using std::placeholders::_1;
1 // [...]
2 subscription_ = this->create_subscription<std_msgs::msg::String>(
3     "topic", 10, std::bind(&MyNode::topic_callback, this, _1));
```

De `std::placeholders` moeten we in de `std::bind()` gebruiken omdat we een placeholder moeten geven voor de parameter (het bericht van de topic) die de functie meekrijgt⁹.

Als er een bericht wordt gepubliceerd dan wordt de functie aangeroepen met als parameter de message. Hieronder een voorbeeld van functie die een message ontvangt en deze print via de logger:

```
0 void topic_callback(const std_msgs::msg::String::SharedPtr msg) const{
1     RCLCPP_INFO(this->get_logger(), "I heard: '%s'", msg->data.c_str());
2 }
```

Met de boven genoemde elementen kunnen we een subscriber maken. In codevoorbeelden 3.5, 3.6 en 3.7 gebruiken we deze elementen om een subscriber te maken die elke keer als er een bericht wordt gepubliceerd op de topic met de naam `topic` de functie `topic_callback()` aanroept.

```
0 #include "rclcpp/rclcpp.hpp"
1 #include "std_msgs/msg/string.hpp"
2
3 class SubscriberNode : public rclcpp::Node
4 {
5 public:
6     SubscriberNode();
7
8 private:
9     // the function called everytime we receive a message from the topic:
10    void topic_callback(const std_msgs::msg::String::SharedPtr msg) const;
11
12    // the subscription:
13    rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription_;
14};
```

Codevoorbeeld 3.5
Subscriber.hpp

```
0 #include "rclcpp/rclcpp.hpp"
1 #include "std_msgs/msg/string.hpp"
2 #include "SubscriberNode.h"
3
4 using std::placeholders::_1;
5
6 SubscriberNode::SubscriberNode(): Node("minimal_subscriber"){
7     // initialize the subscription with:
```

⁸ Als een publisher een bericht publiceert roept hij eigenlijk indirect van alle subscribers een functie aan. We zeggen daarom dat het initiatief van de communicatie bij de publisher ligt. Immers de subscriber bepaalt niet zelf wanneer het een bericht van een topic afhaalt.

⁹Een uitleg over `std::bind` en `std::placeholders` kan men vinden op: <https://www.youtube.com/watch?v=JtUZmkvRoKg>

```

8 // "topic" name of the topic
9 // 10: size of the queue buffer for backup
10 // binding the function topic_callback()
11 // with std::placeholders::_1 we place a placeholder for the
12 // function parameter (the message from the topic)
13 subscription_ = this->create_subscription<std_msgs::msg::String>(
14   "topic", 10, std::bind(&SubscriberNode::topic_callback, this, _1));
15 }
16
17
18 void SubscriberNode::topic_callback(const std_msgs::msg::String::SharedPtr msg) const
19 {
20   RCLCPP_INFO(this->get_logger(), "I heard: '%s'", msg->data.c_str());
21 }
```

Codevoorbeeld 3.6
Subscriber.cpp

```

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

Codevoorbeeld 3.7
mainSubscriber.cpp

3.3 Message types

ROS komt met een groot aantal build-in message types¹⁰. Je code krijgt toegang hier toe door `std_msgs`¹¹ te includen en toe te voegen aan je dependencies in package.xml¹² en CMakeLists.txt¹³. Het is ook mogelijk om zelf message types te maken. Dit doe je door een `.msg`-bestand te maken. `.msg`-bestanden zijn te gebruiken als ze binnen je package staan, maar het is ook mogelijk om gebruik te maken van een `.msg`-bestand uit een andere package. Een conventie is om een aparte package te maken met alle message-types. Andere conventies geven juist weer voorkeur aan het plaatsen van de message-types binnen de packages die ze gebruiken. Waar de message-types staan is dus onderhevig aan de voorkeur van de programmeur/het team. In de codevoorbeelden die komen bij deze cursus staan voorbeelden met custom messages binnen het buiten de package. Meer informatie over hoe je zelf een message type maakt kan je vinden op:

<https://index.ros.org/doc/ros2/Tutorials/Custom-ROS2-Interfaces/>
en extra verdieping:

<https://index.ros.org/doc/ros2/Tutorials/Single-Package-Define-And-Use-Interface/>

Let op: Bij het maken van een custom message type is het verplicht in de naamgeving PascalCase¹⁴ te gebruiken. ROS zet de custom message om naar een `.hpp`-bestand

¹⁰Voor de complete lijst: https://index.ros.org/p/std_msgs/

¹¹Of specifieker als je een string-message wil: `std_msgs/msg/string.hpp`

¹²`<depend>std_msgs</depend>`

¹³`find_package(std_msgs REQUIRED)` plus toevoegen aan `ament_target_dependencies()`

¹⁴zie <https://nl.wikipedia.org/wiki/CamelCase>

met de naam in snake_case¹⁵, waarbij tussen elke kleine letter en hoofdletter van de PascalCase naamgeving nu een onderstrepingssteken (_) ¹⁶ staat. Bijvoorbeeld de custom message *MyCustomMessage.msg* wordt geïncludeerd met `#include "path/to/folder/my_custom_message.hpp"`.

3.4 Publisher revisited

In sectie 3.1 hebben we een publisher gemaakt die om gezette intervallen een bericht stuurt. Hiervoor hebben we gebruik gemaakt van een *wall timer* die steeds een callback¹⁷ doet naar een functie van de node. Dit is een logische implementatie voor bijvoorbeeld een sensornode die elke 500 miliseconden de waarde van de sensor uitleest en deze publiceert.

Er zijn echter ook systemen die niet elk interval iets moeten publiceren, maar constant aan het werk zijn en direct moeten publiceren als ze iets detecteren. Het is daarom belangrijk te realiseren dat het spinnen¹⁸ alleen nodig is als de node callback's moet kunnen ontvangen. Voor het publiceren is het niet nodig dat de node aan het spinnen is. Codevoorbeeld 3.8 geeft ons een voorbeeld hiervan. De main-functie in het codevoorbeeld maakt door middel van een node een publisher en gebruikt deze om 10 berichten te sturen.

```

0 #include "rclcpp/rclcpp.hpp"
1 #include "std_msgs::msg::String>("
2
3 int main(int argc, char * argv[])
4 {
5     rclcpp::init(argc, argv);
6     auto node = std::make_shared<rclcpp::Node>("my_node");
7     auto pub = node->create_publisher<std_msgs::msg::String>(>("topic", 10);
8
9     auto message = std_msgs::msg::String>();
10    int count = 0;
11
12    for(int i=0; i<10; i++){
13        int count++;
14        message.data = "Hello, world! " + std::to_string(count);
15        pub->publish(message);
16
17        for(int j=0; j<1000000000; j++){} // wait some time
18    }
19
20    return 0;
21 }
```

Codevoorbeeld 3.8

publishingMain.cpp; zonder node-subclass en zonder *rclcpp::spin()*

Het is natuurlijk ook mogelijk om een node-class te maken die publiceert zonder dat hij spint. In codevoorbekelen 3.9, 3.10 en 3.11 zien we een custom publishing node die wel publieert, maar niet spint.

¹⁵zie: https://en.wikipedia.org/wiki/Snake_case

¹⁶ook wel bekend als *underscore*. Zie <https://nl.wikipedia.org/wiki/Underscore>

¹⁷Een callback is een bericht/functieaanroep die een node moet afhandelen. Bijvoorbeeld in de publisher van sectie 3.1 is dit de wall timer die afloopt en *timer_callback()* aanroeft en bij de subscriber in dit hoofdstuk is dat het ontvangen van een bericht van de topic en daarmee de aanroep van de functie *topic_callback()*. Om callbacks te ontvangen/af te handelen moet een node aan het spinnen zijn.

¹⁸De functie *rclcpp::spin();* al zijn er ook andere opties hiervoor. Zie: https://docs.ros2.org/foxy/api/rclpy/api/init_shutdown.html en https://docs.ros2.org/foxy/api/rclpy/api/execution_and_callbacks.html.

```

0 #include "rclcpp/rclcpp.hpp"
1 #include "std_msgs/msg/string.hpp"
2
3 class PublisherNode : public rclcpp::Node
4 {
5 public:
6     PublisherNode();
7
8 private:
9     void publishHelloWorld(const int & n);
10
11    // publisher with message type std_msgs::msg::String :
12    rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
13 };

```

Codevoorbeeld 3.9

PublisherNode.hpp; zonder een wall timer.

```

0 #include "rclcpp/rclcpp.hpp"
1 #include "std_msgs/msg/string.hpp"
2 #include "PublisherNode.h"
3
4 PublisherNode::PublisherNode(): Node("PublisherNode"){
5     publisher_ = this->create_publisher<std_msgs::msg::String>("topic", 10);
6 }
7
8 void PublisherNode::publishHelloWorld(const int & n){
9
10    auto message = std_msgs::msg::String();
11    int count = 0;
12
13    for(int i=0; i<n; i++){
14        int count++;
15        message.data = "Hello, world! " + std::to_string(count);
16        pub->publish(message);
17
18        for(int j=0; j<1000000000; j++){} // wait some time
19    }
20 }

```

Codevoorbeeld 3.10

PublisherNode.cpp; zonder een wall timer.

```

0 #include "rclcpp/rclcpp.hpp"
1 #include "PublisherNode.h"
2
3 int main(int argc, char * argv[])
4 {
5     rclcpp::init(argc, argv);
6     auto node = PublisherNode();
7     node.publishHelloWorld(10);
8     return 0;
9 }

```

Codevoorbeeld 3.11

mainPublisher.cpp; zonder rclcpp::spin()

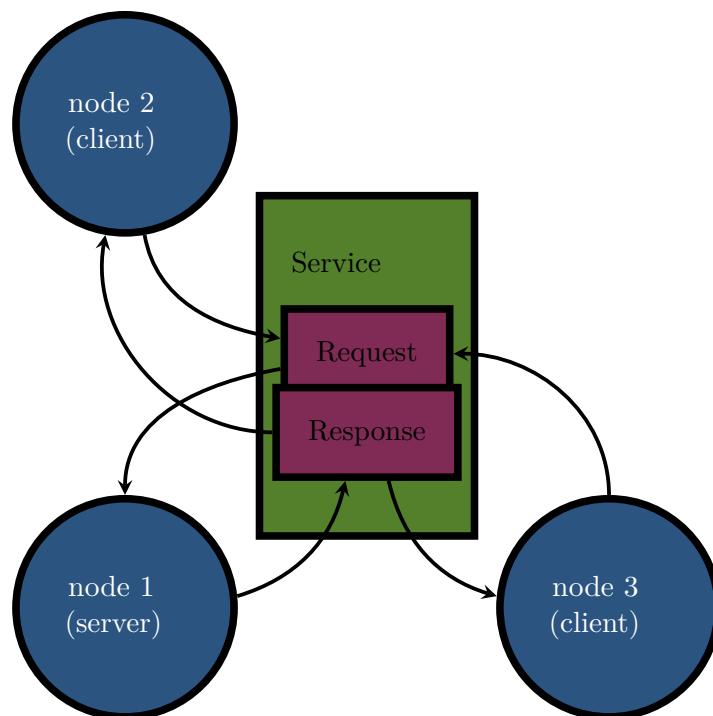


4. Services

De tweede manier van communicatie tussen nodes in ROS zijn *services*. Services werken met een *call-and-response*-model. Dit betekent dat er alleen een bericht/informatie/data wordt gegeven als er om wordt gevraagd. Dit is een groot verschil met topics. Bij topics worden er constant updates gegeven op de topic ongeacht of iemand er om vraagt. Het initiatief bij een topic ligt bij de publishers.

Bij services hebben we *clients*, die om informatie kunnen vragen, en een *server* die na aanvraag van een client de informatie levert. Het initiatief van de communicatie ligt bij de clients. Een service kan meerdere clients hebben, maar een service kent altijd maar één server. In figuur 4.1 zien we een voorbeeld van een services met één server en twee clients. Node 1 zal pas een *response* geven, nadat het een *request* heeft ontvangen van node 2 of 3.

Figuur 4.1
Een ROS service.
Met node 1 als
server en node 2
en 3 als client.



4.1 Request and response structure

Een service heeft altijd een request and response structure beschreven in bestand van het type *.srv*. In deze structuur staan de parameters van het request en het de response van de service. De parameter(s) van de request staan boven een streep van drie gedachtestreepjes (—)¹⁹. De parameter(s) van de response staan onder de drie gedachtestreepjes. In codevoorbeeld 4.1 zien we een voorbeeld van een request and response structure. De request bestaat in dit voorbeeld uit de drie parameters *a*, *b* en *c* van het type int64 en de response bestaat uit de parameter *sum* van het type int64.

```
0 int64 a
1 int64 b
2 int64 c
3 ---
4 int64 sum
```

Codevoorbeeld 4.1
AddThreeInts.srv

De mogelijke types die kunnen worden gebruikt in de request and response structure zijn te vinden op:

<https://index.ros.org/doc/ros2/Concepts/About-ROS-Interfaces/>

Net zoals het message type van de publishers/subscribers (zie sectie 3.3) is het mogelijk de request and response structure binnen de package te plaatsen, maar ook om deze buiten de package te plaatsen.

Let op: Bij het maken van een *.srv*-bestand is het verplicht in de naamgeving PascalCase²⁰ te gebruiken. ROS zet de *.srv* om naar een *.hpp*-bestand met de naam in snake_case²¹, waarbij tussen elke kleine letter en hoofdletter van de PascalCase naamgeving nu een onderstrepingssteken (_)²² staat. Bijvoorbeeld de request and response structure beschreven in *MyCustomService.srv* wordt geïncludeerd met:

```
0 #include "path/to/folder/my_custom_service.hpp".
```

Zie sectie 4.4 voor verwijzingen naar bronnen die met meer diepgang messages en request and response structures van ROS behandelen. In deze bronnen staat ook hoe je *.srv*-bestanden toevoegd aan je package en hoe je ze vanuit een andere package includeert.

4.2 Server code

Een node noemen we een *service*, *server* of, om extra onderscheid te maken met een *action server* (zie hoofdstuk 5), een *service server* als het een membervariabele van het type *rclcpp::Service<T>* heeft. Een *rclcpp::Service<T>* is een template-class die een request and response structure nodig heeft. Het declareren van een *rclcpp::Service<T>* variabele doen we binnen de node met *rclcpp::Service<request_response_struct>::SharedPtr service_name;*. Bijvoorbeeld de server *service_* met de request and response structure *myPackages/srv/MyStruct.srv*²³:

```
0 #include "myPackages/srv/my_struct.hpp"
1 // [...]
```

¹⁹Het gedachtestreepje of aandachtsstreepje is een leesteken dat de vorm heeft van een liggend streepje. Zie ook: <https://nl.wikipedia.org/wiki/Gedachtestreepje>

²⁰zie <https://nl.wikipedia.org/wiki/CamelCase>

²¹zie: https://en.wikipedia.org/wiki/Snake_case

²²ook wel bekent als *underscore*. Zie <https://nl.wikipedia.org/wiki/Underscore>

²³Zie sectie 4.1

```
2 rclcpp::Service<myPackages::srv::MyStruct>::SharedPtr service_;
```

Codevoorbeeld 4.2

Een voorbeeld van het declareren van een service server.

Een service reageert op elk bericht. Bij het initialiseren van de service moeten we daarom net als bij de subscriber een functie toekennen die het bericht afhandelt. Deze functie heeft verplicht twee parameters: een shared pointer naar de request en een shared pointer naar de response. Hiermee kan de functie de data van de request binnen krijgen en met de response kan de service data terugsturen naar client. Het toekennen van de functie aan de service doen we met *create_service<T>()* die we erven van de superclass rclcpp::Node. De functie *create_service<T>()* verwacht, naast een binding met een functie, de naam van de service en een request and response structure. In het geval van de bovenstaand gedeclareerde *service_* met de service naam “*my_service*” en de member functie *provideService()* die de service-request afhandelt doen we dat met:

```
0 #include "myPackages/srv/my_struct.hpp"
1
2 using std::placeholders::_1;
3 using std::placeholders::_2;
4 // [...]
5 service_ = this->create_service<myPackages::srv::MyStruct>(
6     "my_service",
7     std::bind(&myNode::provideService, this, _1, _2));
```

Codevoorbeeld 4.3

Het initialiseren van een service server. Meestal gebeurt dit in de constructor.

De *std::placeholders* moeten we in de *std::bind()* gebruiken omdat we een placeholder moeten geven voor de twee parameters die de gekoppelde functie verplicht moet hebben²⁴. De functie die gekoppeld is aan de service heeft verplicht twee parameters: request en response. De types van de parameters zijn afhankelijk van het .srv-bestand, maar zijn altijd een shared pointer. De gekoppelde functie heeft als taak een response te bepalen, eventueel aan de hand van het request, en deze toe te kennen aan de response parameter. Codevoorbeeld 4.4 geeft een voorbeeld van een dergelijke functie. Het voorbeeld gebruikt de request and response van codevoorbeeld 4.1.

```
0 void myNode::provideService(
1     const std::shared_ptr<srvccli_custom_srv_in_pkg::srv::AddThreeInts::Request> request,
2     std::shared_ptr<srvccli_custom_srv_in_pkg::srv::AddThreeInts::Response> response )
3 {
4     // processing the request (calculating the sum) and assigning it to the response:
5     response->sum = request->a + request->b + request->c;
6 }
```

Codevoorbeeld 4.4

Een voorbeeld van een service-functie.

In codevoorbeelden 4.5, 4.6 en 4.7 zien we alle serveronderdelen samengevoegd tot de server node *ServiceNode*. ServiceNode maakt gebruikt de request and response structure gedefinieerd in codevoorbeeld 4.1. De node is ook te vinden de package-voorbeelden die komen met deze reader.

```
0 #include "rclcpp/rclcpp.hpp"
1 // srv/AddThreeInts.srv becomes srv/add_three_ints.hpp:
2 #include "srvccli_custom_srv_in_pkg/srv/add_three_ints.hpp"
3
4 class ServiceNode : public rclcpp::Node
```

²⁴Een uitleg over *std::bind* en *std::placeholders* kan men vinden op: <https://www.youtube.com/watch?v=JtUZmkvroKg>

```

5  {
6  public:
7      ServiceNode();
8
9  private:
10     // function to be called by the service.
11     void add(
12         const std::shared_ptr<srvcli_custom_srv_in_pkg::srv::AddThreeInts::Request> request,
13         std::shared_ptr<srvcli_custom_srv_in_pkg::srv::AddThreeInts::Response> response
14     );
15
16     rclcpp::Service<srvcli_custom_srv_in_pkg::srv::AddThreeInts>::SharedPtr service_;
17 }

```

Codevoorbeeld 4.5
ServiceNode.hpp

```

0 #include "rclcpp/rclcpp.hpp"
1 // srv/AddThreeInts.srv becomes srv/add_three_ints.hpp:
2 #include "srvcli_custom_srv_in_pkg/srv/add_three_ints.hpp"
3 #include "ServiceNode.hpp"
4
5 // placeholders for the arguments
6 using std::placeholders::_1;
7 using std::placeholders::_2;
8
9 ServiceNode::ServiceNode(): Node("ServiceNode"){
10     // initialise the service
11     // <srv-message-type> in this case custom_interfaces::srv::AddTwoInts
12     // "add_three_ints" the name of the topic
13     // and a pointer to the function to be called when a service message is received.
14     service_ = this->create_service<srvcli_custom_srv_in_pkg::srv::AddThreeInts>(
15         "add_three_ints",
16         std::bind(&ServiceNode::add, this, _1, _2));
17 }
18
19 void ServiceNode::add(
20     const std::shared_ptr<srvcli_custom_srv_in_pkg::srv::AddThreeInts::Request> request,
21     std::shared_ptr<srvcli_custom_srv_in_pkg::srv::AddThreeInts::Response> response
22 ){
23     // processing the request (calculating the sum) and assigning it to the response:
24     response->sum = request->a + request->b + request->c;
25
26     // for example purposes, printing to the logging:
27     RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Incoming request\na: %ld" " b: %ld" "c: %ld",
28                 request->a, request->b, request->c);
29     RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "sending back response: [%ld]",
30                 (long int)response->sum);
31 }

```

Codevoorbeeld 4.6
ServiceNode.cpp

```

0 #include "rclcpp/rclcpp.hpp"
1 #include "ServiceNode.hpp"
2
3 int main(int argc, char **argv)
4 {
5     rclcpp::init(argc, argv);
6     rclcpp::spin(std::make_shared<ServiceNode>());
7     rclcpp::shutdown();
8     return 0;

```

9 }

Codevoorbeeld 4.7
mainServer.cpp

4.3 Client code

Om een request in te dienen bij een ROS-service moeten we de communicatierol *client*²⁵ aannemen. Dit doen we door een member variabele te declareren van het type *rclcpp::Client<T>*. Een *rclcpp::Client<T>* is een template-class die een request and response structure (zie sectie 4.1) nodig heeft. Het declareren van een *rclcpp::Client<T>* variabele doen we binnen de node met *rclcpp::Client<request_response_structure>::SharedPtr client_name;*. Bijvoorbeeld de client *client_* met de request and response structure *myPackages/srv/MyStruct.srv*:

```
0 #include "myPackages/srv/my_struct.hpp"
1 // [...]
2 rclcpp::Client<myPackages::srv::MyStruct>::SharedPtr client_;
```

Codevoorbeeld 4.8

Een voorbeeld van het declareren van een service client.

Voor het initialiseren van de *rclcpp::Client<T>* gebruiken we de functie *create_client<T>()* die een node erft van *rclcpp::Node*. Deze functie verreist enkel de naam van de service als argument. Het versturen van de request en het koppelen van een functie aan het ontvangen van de response handelen we later af. De initialisatie is dus relatief simpel. In het geval van de bovenstaande gedeclareerde *client_* met de service naam “*my_service*” doen we dat met:

```
0 #include "myPackages/srv/my_struct.hpp"
1 // [...]
2 client_ = this->create_client<myPackages::srv::MyStruct>("my_service");
```

Codevoorbeeld 4.9

Het initialiseren van een service client. Meestal gebeurt dit in de constructor.

Met de geïnitialiseerde service client kunnen we nu een request sturen naar de service server. Hiervoor maken we eerst een request. Dit doen we door een shared pointer te maken van de request van de juiste request and response structure. Hiervoor gebruiken we de functie *Request()* die samen met de request and response structure komt. In het geval van de request and response structure *AddThreeInts.srv* (zie codevoorbeeld 4.1) doen we dat als volgt:

```
0 auto request = std::make_shared<srvccli_custom_srv_in_pkg::srv::AddThreeInts::Request>();
```

Voordat we de request versturen moet hij nog gevuld worden met data. De request heeft publieke member-variabelen met de namen zoals aangegeven in de request and response structure. Hieraan kunnen we direct waardes toekennen. Bij de bovenstaand gemaakte *request* vullen we de drie member *a*, *b* en *c* op de volgende manier:

```
0 request->a = 5;
1 request->b = 7;
2 request->c = 9;
```

Nu we een request gevuld met data hebben, kan deze worden verstuurd. Voordat we de request versturen is het verstandig om te kijken of de service server node wel aan het draaien is. Dit kunnen we doen met behulp van de functie *wait_for_service()* van de client die we hebben gemaakt. Het volgende stuk code controleert elke seconde of de service server draait, totdat hij de server heeft gevonden.

²⁵Soms ook *service client* genoemd om duidelijker onderscheid te maken met *action clients*

```

0 while (!client_->wait_for_service(1s)) {
1     if (!rclcpp::ok()) {
2         RCLCPP_ERROR(
3             rclcpp::get_logger("rclcpp"),
4             "Interrupted while waiting for the service. Exiting."
5         );
6     }
7     RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "service not available, waiting again...");
8 }
```

Gegeven dat de service server bestaat kan er een request worden gestuurd door middel van de functie *async_send_request()*. Deze functie van onze `rclcpp::Client<T>` heeft als argumenten de gemaakte request en binding met een functie die response van de server afhandelt. De functie die de response afhandelt heeft als parameter de response van de server. Met de client *client_*, de request *request* en de functie *handleResponse()* (met als argument de response, maar die binden we tijdelijk met een *std::placeholder*), die het response van de server afhandelt, versturen we de request op de onderstaande manier:

```

0 using std::placeholders::_
1 // [...]
2 auto future = client_->async_send_request(request, std::bind(&ClientNode::handleResponse,
3                                              this, _1));
```

Codevoorbeeld 4.10

Een voorbeeld van het versturen van een request door een `rclcpp::Client`.

de `rclcpp::Client<T>` biedt ook nog andere mogelijkheden rond het versturen van requests. De return value van *async_send_request()*, in codevoorbeeld 4.10 *future* genoemd, biedt ook nog mogelijkheden die interessant zijn om te verkennen, maar waar we in deze reader niet op in gaan.

Met deze onderdelen kunnen we een service client node maken die een request stuurt naar de eerder genoemde service server node. Dit is uitgewerkt in codevoorbeelden 4.11, 4.12 en 4.13. Dit voorbeeld staat ook in de voorbeelden-packages die komen bij deze reader.

```

0 #include "rclcpp/rclcpp.hpp"
1 // srv/AddThreeInts.srv becomes srv/add_three_ints.hpp:
2 #include "srvcli_custom_srv_in_pkg/srv/add_three_ints.hpp"
3
4 class ClientNode : public rclcpp::Node
5 {
6 public:
7     ClientNode();
8
9 private:
10    // function that makes the request:
11    void makeRequest();
12
13    // function that handles the response:
14    void handleResponse(rclcpp::Client<srvcli_custom_srv_in_pkg::srv::AddThreeInts>::
15                         SharedFuture future);
16
17    rclcpp::Client<srvcli_custom_srv_in_pkg::srv::AddThreeInts>::SharedPtr client_;
};
```

Codevoorbeeld 4.11

`ClientNode.hpp`

```

0 #include "rclcpp/rclcpp.hpp"
1 // srv/AddThreeInts.srv becomes srv/add_three_ints.hpp:
2 #include "srvcli_custom_srv_in_pkg/srv/add_three_ints.hpp"
3 #include "ClientNode.hpp"
```

```

4 // needed for the use of the 's' notation in wait_for_service()
5 using namespace std::chrono_literals;
6
7 // placeholder for the arguments
8 using std::placeholders::_1;
9
10
11 ClientNode::ClientNode(): Node("ClientNode"){
12     // initialise the client.
13     // "add_three_ints" the name of the service
14     client_ = this->create_client<srvcli_custom_srv_in_pkg::srv::AddThreeInts>(
15         "add_three_ints");
16
17     // call the service:
18     makeRequest();
19 }
20
21 void ClientNode::makeRequest(){
22     // make a request of the right srv-type:
23     auto request = std::make_shared<srvcli_custom_srv_in_pkg::srv::AddThreeInts::Request
24     >();
25
26     // add the data to the request
27     request->a = 5;
28     request->b = 7;
29     request->c = 9;
30
31     // wait for the service to exist.
32     // if the service is never started this is a endless loop
33     while (!client_->wait_for_service(1s)) {
34         if (!rclcpp::ok()) {
35             RCLCPP_ERROR(
36                 rclcpp::get_logger("rclcpp"),
37                 "Interrupted while waiting for the service. Exiting.");
38         }
39         RCLCPP_INFO(
40             rclcpp::get_logger("rclcpp"),
41             "service not available, waiting again...");
42     }
43
44     // send the request and bind the function call the handles the response
45     auto future = client_->async_send_request(
46         request,
47         std::bind(&ClientNode::handleResponse, this, _1)
48     );
49 }
50
51 void ClientNode::handleResponse(
52     rclcpp::Client<srvcli_custom_srv_in_pkg::srv::AddThreeInts>::SharedPtr future
53 ){
54     auto result = future.get();
55     RCLCPP_INFO(this->get_logger(), "Result of add_two_ints: %i", result->sum);
56 }
```

Codevoorbeeld 4.12
ClientNode.cpp

```

0 #include "rclcpp/rclcpp.hpp"
1 #include "example_interfaces/srv/add_two_ints.hpp"
2 #include "ClientNode.hpp"
3
```

```
4 int main(int argc, char **argv)
5 {
6     rclcpp::init(argc, argv);
7     rclcpp::spin(std::make_shared<ClientNode>());
8     rclcpp::shutdown();
9     return 0;
10 }
```

Codevoorbeeld 4.13
mainClient.cpp

4.4 Meer bronnen

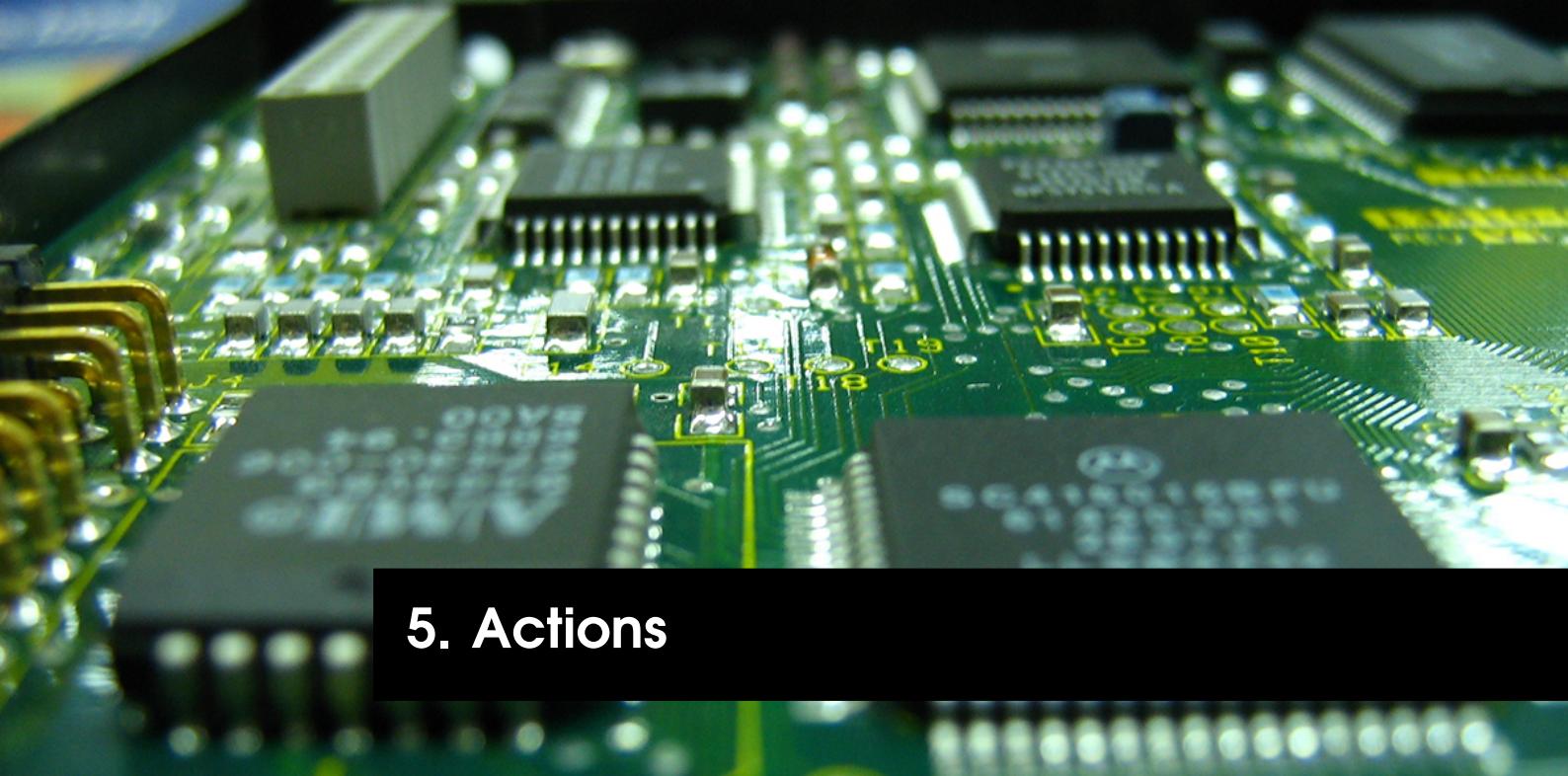
Andere voorbeelden van het maken van een server en een client kan men vinden in de voorbeeld-ROS-packages die komen bij deze reader of op:

<https://index.ros.org/doc/ros2/Tutorials/Writing-A-Simple-Cpp-Service-And-Client/>

Hoe je eigen service en message definieert kan je vinden op:

<https://index.ros.org/doc/ros2/Tutorials/Custom-ROS2-Interfaces/>
en extra verdieping:

<https://index.ros.org/doc/ros2/Tutorials/Single-Package-Define-And-Use-Interface/>



5. Actions

Het laatste communicatietype van ROS 2 is actions. Action communicatie bestaat uit de communicatierollen *action server* en *action client*. Actions servers zijn een soort service servers met het grote verschil dat een action server feedback geeft aan de action client terwijl hij de request uitvoert. De action-communicatie is afgebeeld in figuur 5.1. We zien daar dat een action eigenlijk bestaat uit één topic en twee services.

Om een actie te starten stuurt de action client via de goal service een goal request. De action server reageert hier op met een *accept* of met een *reject*. Als de goal request is geaccepteerd, dan gaat de action server de goal uitvoeren en geeft tijdens het uitvoeren feedback over het uitvoeren van de actie aan de action client. Als de actie klaar is of gecanceld wordt, dan stuurt de action server het resultaat terug naar de action client. In tegenstelling tot bij services kunnen acties voortijdig gestopt worden.

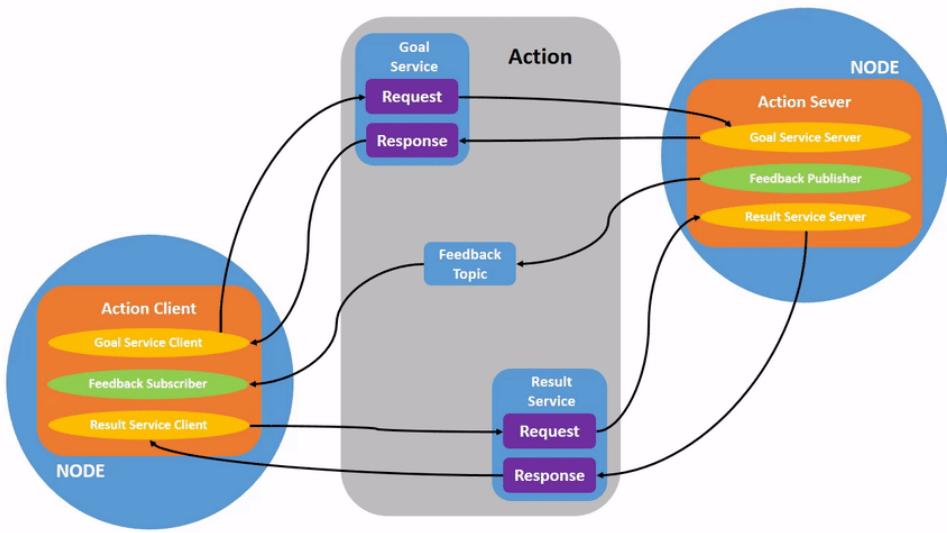
Als we naar afbeelding 5.1 kijken lijkt het alsof we ons zorgen moeten maken over twee services en een topic. Gelukkig neemt ROS 2 deze zorgen voor ons weg. Actions worden gezien als 1 communicatiestructuur en er zijn maar 3 berichten die we zelf moet definiëren: *request*, *result* en *feedback*. Net als bij services moeten we de structuur van deze berichten vastleggen. Bij actions leggen we deze structuur vast in een .action-bestand.

5.1 Action message structure

Net als bij services moet de structuur van de berichten aangeven worden. Dit doen we in een .action-bestand en wordt in de ROS documentatie een ‘action’ genoemd. In deze reader verwijzen we er naar met de term ‘action message structure’. In de action message structuur staan de parameters van de *goal request*, van de *result* en van de *feedback*. In het bestand zijn deze onderdelen gescheiden door middel van drie gedachtenstreepjes (-)²⁶. Met eerst de goal request, daarna de result en als laatste de structuur van de feedback. In codevoorbeeld 5.1 zien we de template voor een action message structure. In codevoorbeeld 5.2 zien we een uitgewerkte action message structure waarbij de goal request bestaat uit een int32, de result uit een vector<int32> en de feedback uit een vector<int32>. Net als bij services en topics beschrijven we per message een struct, dus we kunnen ook meerdere (verschillende) types toekennen aan één bericht.

²⁶zie: <https://nl.wikipedia.org/wiki/Gedachtenstreepje>

Figuur 5.1
Action Node en een client node.
(afbeelding van ros.org)



```

0 # Goal request
1 ---
2 # Result
3 ---
4 # Feedback

```

Codevoorbeeld 5.1
Template van een .action-bestand

```

0 int32 order
1 ---
2 int32[] sequence
3 ---
4 int32[] partial_sequence

```

Codevoorbeeld 5.2
Het .action-bestand voor een fibonacci-action server.

De mogelijke types die kunnen worden gebruikt in de action message structure zijn te vinden op:

<https://index.ros.org/doc/ros2/Concepts/About-ROS-Interfaces/>

Net zoals het message type van de publishers/subscribers (zie sectie 3.3) en de request and response structure van services (zie sectie 4.1) is het mogelijk de action message structure binnen de package te plaatsen, maar ook om deze buiten de package te plaatsen.

Let op: Bij het maken van een .action-bestand is het verplicht in de naamgeving PascalCase²⁷ te gebruiken. ROS zet de .action om naar een .hpp-bestand met de naam in snake_case²⁸, waarbij tussen elke kleine letter en hoofdletter van de PascalCase naamgeving nu een onderstrepingssteken (_) staat. Bijvoorbeeld de action message structure beschreven in *MyCustomAction.action* wordt geïncludeerd met:

```
0 #include "path/to/folder/my_custom_action.hpp".
```

In sectie 5.4 staan meer bronnen over action message structure. In deze bronnen staat ook hoe je .action-bestanden toevoegt aan je package en hoe je ze vanuit een andere package includeert.

²⁷ zie <https://nl.wikipedia.org/wiki/CamelCase>

²⁸ zie: https://en.wikipedia.org/wiki/Snake_case

²⁹ ook wel bekend als *underscore*. Zie <https://nl.wikipedia.org/wiki/Underscore>

5.2 Action Server

Een node noemen we een *action server* als het een membervariabele van het type *rclcpp_action::Server<T>* heeft. Om historische redenen zitten de benodigdheden voor action server en client niet in *rclcpp*, maar in *rclcpp_action*. Deze includeren we met:

```
0 #include "rclcpp_action/rclcpp_action.hpp"
```

Met *rclcpp_action* kunnen we vervolgens een action server declareren. Bijvoorbeeld de action server *action_server_* met de action message structure *custom_interfaces::action::Fibonacci*:

```
0 // 'using' allows us to use a shorter word for long types:
1 using FibAction = custom_interfaces::action::Fibonacci;
2 // [...]
3 rclcpp_action::Server<FibAction>::SharedPtr action_server_;"
```

Een action server moet drie typen berichten afhandelen: *goal request*, *cancel request* en *accept and execute*. Bij elk bericht hoort ook een taak. De action server moet na het ontvangen van een *goal request* of een *cancel request* het verzoek accepteren of weigeren. Bij de ontvangst van een *accept and execute* bericht moet de action server het uitvoeren van de *goal* starten. Tijdens het uitvoeren van de *goal* kan de action server feedback sturen naar de action client. Als de *goal* behaald is of als de *goal* wordt gecanceld wordt het resultaat gestuurd naar de action client.

Bij het initialiseren van de action server moeten we voor elk type bericht een functie toekennen die het bericht afhandelt. Deze functies hebben verplichte parameters die we bij het initialiseren binden met *std::placeholders*. Het initialiseren doen we met de template-functie *rclcpp_action::create_server<T>()* die als template de action message structure heeft. De complete initialisatie van de action server *action_server_* in de class *FibActionServer* met de functie *handle_goal* voor het afhandelen van de *goal request*, de functie *handle_cancel* voor het afhandelen van de *cancel request* en de functie *handle_accepted* voor het afhandelen van de *goal acceptatie en executie* ziet er uit als:

```
0 using namespace std::placeholders;
1 // 'using' allows us to use a shorter word for long types:
2 using FibAction = custom_interfaces::action::Fibonacci;
3 // [...]
4 this->action_server_ = rclcpp_action::create_server<FibAction>(
5     this,
6     "fibonacci",
7     std::bind(&FibActionServer::handle_goal, this, _1, _2),
8     std::bind(&FibActionServer::handle_cancel, this, _1),
9     std::bind(&FibActionServer::handle_accepted, this, _1));
```

Code voorbeeld 5.3

Initialisatie van de action server *action_server_*.

De functies die worden gebonden in tijdens de initialisatie van een *rclcpp_action::create_server* hebben net als de functies bij subscribers, service server en service client verplichte parameters en return types. Om de lezer daar bewust van te maken laten we de functie declaraties zien van de gebonden functies in code voorbeeld 5.3. In de volgende secties bekijken we de inhoud van deze functies.

```
0 // 'using' allows us to use a shorter word for these long types:
1 using FibAction = custom_interfaces::action::Fibonacci;
2 using GoalHandleFib = rclcpp_action::ServerGoalHandle<FibAction>;
3
4 // [...]
5
6 rclcpp_action::GoalResponse handle_goal(
```

```

7     const rclcpp_action::GoalUUID & uuid,
8     std::shared_ptr<const FibAction::Goal> goal
9 );
10
11 rclcpp_action::CancelResponse handle_cancel(
12     const std::shared_ptr<GoalHandleFib> goal_handle
13 );
14
15 void handle_accepted(const std::shared_ptr<GoalHandleFib> goal_handle);

```

Codevoorbeeld 5.4

De *action_server*_ functies die de berichten afhandelen.

5.2.1 Goal request functie

Het speciale return type van *handle_goal()* function in codevoorbeeld 5.4 geeft de mogelijkheid om terug te geven of de goal wordt geaccepteerd of niet. Dit doet de functie door of *rclcpp_action::GoalResponse::REJECT* of *rclcpp_action::GoalResponse::ACCEPT_AND_EXECUTE* terug te geven. We zien dit in codevoorbeeld 5.5. Let op de functie in codevoorbeeld 5.5 heeft verplicht twee parameters, maar we gebruiken de parameter *uuid*³⁰ niet. Om warnings hiervan te onderdrukken casten we de parameter naar *void*³¹.

```

0 rclcpp_action::GoalResponse FibActionServer::handle_goal(
1     const rclcpp_action::GoalUUID & uuid,
2     std::shared_ptr<const FibAction::Goal> goal
3 ){
4     RCLCPP_INFO(this->get_logger(), "Received goal request with order %d", goal->order);
5
6     (void)uuid;
7
8     // to show an example of rejecting we reject fibonacy sequences that are over 9000:
9     if (goal->order > 9000) {
10         return rclcpp_action::GoalResponse::REJECT;
11     }
12     return rclcpp_action::GoalResponse::ACCEPT_AND_EXECUTE;
13 }

```

Codevoorbeeld 5.5

De *handle_goal()* functie van *action_server*

Als de functie die de goal request afhandelt *ACCEPT_AND_EXECUTE* teruggeeft wordt de functie van de action server gebonden aan dit bericht aangeroepen (zie sectie 5.2.3). Accept and execute is dus een bericht van de action server naar de action client en naar zichzelf.

5.2.2 Cancel request functie

Op dezelfde manier als de goal request functie kan de functie die de cancel request afhandelt door middel van *rclcpp_action::CancelResponse::REJECT* of *rclcpp_action::CancelResponse::ACCEPT* aangeven of de cancel request wordt geaccepteerd of niet. Als het cancel request is geaccepteerd betekent dat niet dat de goal ook gelijk is gecanceld. Bij het uitvoeren van de goal moet hier dus actief op worden gecontroleerd en de actie ook echt stoppen (zie sectie 5.2.3).

³⁰UUID (universally unique identifier) is een manier om (zo goed als) unieke identificatoren te maken. Voor meer informatie: https://en.wikipedia.org/wiki/Universally_unique_identifier. Of met meer wiskunde: <https://towardsdatascience.com/are-uuids-really-unique-57eb80fc2a8>.

³¹Voor meer informatie over casten naar void: <https://stackoverflow.com/questions/34288844/what-does-casting-to-void-really-do>

5.2.3 Accepted en execute functie

De functie in de action server node die dit bericht afhandelt start meestal gelijk een andere thread, zodat de node weer nieuwe communicatie kan afhandelen. We zien dit in codevoorbeeld 5.6:

```

0  using namespace std::placeholders;
1  using FibAction = custom_interfaces::action::Fibonacci;
2  using GoalHandleFib = rclcpp_action::ServerGoalHandle<FibAction>;
3  // [...]
4
5  void FibActionServer::handle_accepted(const std::shared_ptr<GoalHandleFib> goal\_handle){
6      // this needs to return quickly to avoid blocking the executor, so spin up a new
7      // thread
8      std::thread{std::bind(&FibActionServer::execute, this, _1), goal_handle}.detach();
}

```

Codevoorbeeld 5.6

handle_accepted() start functie *execute()* in een thread. Hierin wordt de actie uitgevoerd.

Tijdens de uitvoering van de action kan de action server feedback sturen naar de action client en als de actie stopt, omdat de goal behaald of is gecanceld, moet het resultaat worden verstuurd. Dit doen we met behulp *rclcpp_action::ServerGoalHandle<T>*, ook wel de *goal handle* genoemd. Het is daarom belangrijk dat we de goal handle ook meegeven aan de functie die het uitvoeren doet. Dit gebeurt dan ook in codevoorbeeld 5.6. Gegeven de goal handle *goal_handle* kunnen we de goal opvragen, feedback sturen en het eindresultaat sturen. Zie hiervoor respectievelijk codevoorbeeld 5.7, 5.8 en 5.9. Vanuit de goal handle kunnen we ook informatie krijgen of een cancel request geaccepteerd is en aangeven dat de goal ook daadwerkelijk gecanceld is. Code hiervoor zien we in codevoorbeeld 5.10.

```
0  const auto goal = goal_handle->get_goal()
```

Codevoorbeeld 5.7

Gebruik van de *get_goal()* functie.

```

0  auto feedback = std::make_shared<FibAction::Feedback>();
1  auto & sequence = feedback->partial_sequence;
2  sequence.push_back(0);
3  sequence.push_back(1);
4
5  goal_handle->publish_feedback(feedback);

```

Codevoorbeeld 5.8

Het versturen van feedback met een goal handle.

```

0  auto result = std::make_shared<FibAction::Result>();
1  sequence.push_back(0);
2  sequence.push_back(1);
3
4  result->sequence = sequence;
5  goal_handle->succeed(result);

```

Codevoorbeeld 5.9

Het versturen van het result met een goal handle.

```

0  if(goal_handle->is_canceling()) {
1
2      // do, if needed, some actions to really cancel the action goal.
3
4      // set is_cancelling to false again and sends the (incomplete) result:
5      goal_handle->canceled(result); // see previous code example for how to make a result
6
7      return; // end the current function (and thread)

```

```
8 };
```

Codevoorbeeld 5.10

Het controleren op een cancel request acceptatie en de afhandeling daarvan.

5.2.4 Codevoorbeeld

In codevoorbeelden 5.11, 5.12 en 5.13 zien we alle ingredienten van de action server bij elkaar om een action server node te maken die de fibonacci reeks uitrektent.

```
0 #ifndef FIB_ACTION_SERVER_HPP
1 #define FIB_ACTION_SERVER_HPP
2
3 #include "rclcpp/rclcpp.hpp"           // Needed for nodes
4 #include "rclcpp_action/rclcpp_action.hpp" // needed for actions
5
6 // the location of our custom action.
7 // the hpp generated by ROS as a underscore (_) before every capital, expect the first:
8 // for example: /my/path/MyAction.action become /my/path/my_action.action
9 #include "custom_interfaces/action/fibonacci.hpp"
10
11 class FibActionServer : public rclcpp::Node {
12 public:
13     // 'using' allows us to use a shorter word for these long types:
14     using FibAction = custom_interfaces::action::Fibonacci;
15     using GoalHandleFib = rclcpp_action::ServerGoalHandle<FibAction>;
16
17     FibActionServer();
18
19 private:
20     rclcpp_action::GoalResponse handle_goal(
21         const rclcpp_action::GoalUUID & uuid,
22         std::shared_ptr<const FibAction::Goal> goal
23     );
24
25     rclcpp_action::CancelResponse handle_cancel(
26         const std::shared_ptr<GoalHandleFib> goal_handle
27     );
28
29     void handle_accepted(const std::shared_ptr<GoalHandleFib> goal_handle);
30
31     void execute(const std::shared_ptr<GoalHandleFib> goal_handle);
32
33     rclcpp_action::Server<FibAction>::SharedPtr action_server_;
34 };
35
36 #endif /* FIB_ACTION_SERVER_HPP */
```

Codevoorbeeld 5.11

ActionServerNode.hpp

```
0 #include "FibActionServer.hpp"
1 #include "rclcpp/rclcpp.hpp"           // Needed for nodes
2 #include "rclcpp_action/rclcpp_action.hpp" // needed for actions
3
4 // the location of our custom action.
5 // the hpp generated by ROS as a underscore (_) before every capital, expect the first:
6 // for example: /my/path/MyAction.action become /my/path/my_action.action
7 #include "custom_interfaces/action/fibonacci.hpp"
8
9 using namespace std::placeholders;
```

```

11 // constructor:
12 FibActionServer::FibActionServer(): Node("fibonacci_action_server") {
13
14     this->action_server_ = rclcpp_action::create_server<FibAction>(
15         this,
16         "fibonacci",
17         std::bind(&FibActionServer::handle_goal, this, _1, _2),
18         std::bind(&FibActionServer::handle_cancel, this, _1),
19         std::bind(&FibActionServer::handle_accepted, this, _1));
20 }
21
22 rclcpp_action::GoalResponse FibActionServer::handle_goal(
23     const rclcpp_action::GoalUUID & uuid, // uuid means universally unique identifier
24     std::shared_ptr<const FibAction::Goal> goal
25 ){
26     RCLCPP_INFO(this->get_logger(), "Received goal request with order %d", goal->order);
27     // we are not using the uuid. If you dont use a parameter you will get a warning.
28     // the parameters are predefined by ROS, so we cant remove the uuid parameter.
29     // So we supress the warning by casting the uuid variable to void:
30     (void)uuid;
31
32     // to show an example of rejecting we reject fibonacy sequences that are over 9000:
33     if (goal->order > 9000) {
34         return rclcpp_action::GoalResponse::REJECT;
35     }
36     return rclcpp_action::GoalResponse::ACCEPT_AND_EXECUTE;
37 }
38
39 rclcpp_action::CancelResponse FibActionServer::handle_cancel(
40     const std::shared_ptr<GoalHandleFib> goal_handle
41 ){
42     RCLCPP_INFO(this->get_logger(), "Received request to cancel goal");
43     // we are not using the parameter goal_handel
44     // to supress warnings we cast it to void:
45     (void)goal_handle;
46     return rclcpp_action::CancelResponse::ACCEPT;
47 }
48
49 void FibActionServer::handle_accepted(const std::shared_ptr<GoalHandleFib> goal_handle){
50     using namespace std::placeholders;
51     // this needs to return quickly to avoid blocking the executor, so spin up a new
52     // thread
53     std::thread{std::bind(&FibActionServer::execute, this, _1), goal_handle}.detach();
54 }
55
56 void FibActionServer::execute(const std::shared_ptr<GoalHandleFib> goal_handle){
57     RCLCPP_INFO(this->get_logger(), "Executing goal");
58
59     rclcpp::Rate loop_rate(1); // loop frequency
60
61     const auto goal = goal_handle->get_goal();
62     auto feedback = std::make_shared<FibAction::Feedback>();
63     auto & sequence = feedback->partial_sequence;
64     sequence.push_back(0);
65     sequence.push_back(1);
66     auto result = std::make_shared<FibAction::Result>();
67
68     for (int i = 1; (i < goal->order) && rclcpp::ok(); ++i) {
69         // Check if there is a cancel request
70         // is_cancelling is true if a cancelling message is send.
71         if (goal_handle->is_cancelling()) {

```

```

71         result->sequence = sequence;
72         goal_handle->canceled(result); // set is_cancelling to false again
73         RCLCPP_INFO(this->get_logger(), "Goal Canceled");
74         return;
75     }
76
77     // Update sequence
78     sequence.push_back(sequence[i] + sequence[i - 1]);
79
80     // Publish feedback
81     goal_handle->publish_feedback(feedback);
82     RCLCPP_INFO(this->get_logger(), "Publish Feedback");
83
84     // sleep until next loop (see loop_rate at the start of this function)
85     loop_rate.sleep();
86 }
87
88 // the goal is done (the for loop ended):
89 if (rclcpp::ok()) {
90     result->sequence = sequence;
91     goal_handle->succeed(result);
92     RCLCPP_INFO(this->get_logger(), "Goal Succeeded");
93 }
94 }
```

Codevoorbeeld 5.12
ActionServerNode.cpp

```

0 #include "FibActionServer.hpp"
1 #include "rclcpp/rclcpp.hpp"
2
3
4 int main(int argc, char ** argv)
5 {
6     rclcpp::init(argc, argv);
7
8     auto action_server = std::make_shared<FibActionServer>();
9     rclcpp::spin(action_server);
10    rclcpp::shutdown();
11
12    return 0;
13 }
```

Codevoorbeeld 5.13
mainActionServerNode.cpp

5.3 Action client

Een *action client* kan een goal indienen bij een action server. Als deze wordt geaccepteerd, dan krijgt de action client daarvan feedback³², en uiteindelijk het het resultaat. Als een goal loopt bij een action server, dan kan een action client ook een cancel request sturen naar de action server.

Een node neemt de communicatierol *action client* op zich door een member variabele te maken van het type *rclcpp_action::Client*<*T*>. Waarbij voor de template een action message structure moet worden ingevuld. Bijvoorbeeld met de action message structure *custom_interfaces::action::Fibonacci*:

```
0 // 'using' allows us to use a shorter word for long types:
```

³²Mits de action server deze stuurt

```

1  using FibAction = custom_interfaces::action::Fibonacci;
2
3 // [...]
4 rclcpp_action::Client<FibAction>::SharedPtr client_ptr_;

```

Het initialiseren van de action client is relatief makkelijk. Met de functie `rclcpp_action::create_client<T>()` hebben we enkel de action message structure en de naam van de action nodig. Bijvoorbeeld met de action message structure `custom_interfaces::action::Fibonacci` en de action naam `fibonacci`:

```

0  using FibAction = custom_interfaces::action::Fibonacci;
1 // [...]
2
3 this->client_ptr_ = rclcpp_action::create_client<FibAction>(
4     this,
5     "fibonacci");

```

Om een goal te versturen moeten we eerst een goal message maken. Deze maken we met behulp van een functie die ROS ons geeft met de action message structure:

```

0  using FibAction = custom_interfaces::action::Fibonacci;
1 // [...]
2
3 auto goal_msg = FibAction::Goal();
4 goal_msg.order = 10;

```

Voordat we de goal kunnen versturen moeten we instellen welke functies de berichten van de action server verwerken. Er zijn drie berichten die we terug kunnen krijgen: goal response, feedback en result. De goal response bevat of de action server de goal heeft geaccepteerd of heeft geweigerd. Het binden van functies aan de berichten gaat op een vergelijkbare manier als bij andere rollen. Daarna kunnen we goal versturen met `async_send_goal()` van de action client member:

```

0  using namespace std::placeholders;
1  using FibAction = custom_interfaces::action::Fibonacci;
2 // [...]
3
4 // setting the functions that handle the return-messages:
5 auto SGO = rclcpp_action::Client<FibAction>::SendGoalOptions();
6 SGO.goal_response_callback = std::bind(&FibActionClient::goal_response_callback, this, _1);
7 SGO.feedback_callback = std::bind(&FibActionClient::feedback_callback, this, _1, _2);
8 SGO.result_callback = std::bind(&FibActionClient::result_callback, this, _1);
9
10 // sending the goal:
11 auto goal_handle_future = this->client_ptr_->async_send_goal(goal_msg, SGO);

```

Via de parameters van de functies kunnen we de feedback en result binnen halen. Ook kunnen we via deze parameters meer informatie krijgen over het goal. Bijvoorbeeld of de result die we krijgen een resultaat is dat is ontstaan omdat het goal is behaald of omdat het goal is gecanceld. In sectie 5.3.1 zien we het gebruik hiervan in de codevoorbeelden.

5.3.1 Codevoorbeeld

In codevoorbeelden 5.14, 5.15 en 5.16 zien we alle ingrediënten van de action client bij elkaar om een action client server node te maken die een action server vraagt om een fibonacci reeks uit te rekenen.

```

0  #ifndef FIB_ACTION_CLIENT_HPP
1  #define FIB_ACTION_CLIENT_HPP
2
3  #include "rclcpp/rclcpp.hpp"

```

```

4 #include "rclcpp_action/rclcpp_action.hpp"
5
6 // the location of our custom action.
7 // the hpp generated by ROS as a underscore (_) before every capital, expect the first:
8 // for example: /my/path/MyAction.action become /my/path/my_action.action
9 #include "custom_interfaces/action/fibonacci.hpp"
10
11 class FibActionClient : public rclcpp::Node
12 {
13 public:
14     // 'using' allows us to use a shorter word for these long types:
15     using FibAction = custom_interfaces::action::Fibonacci;
16     using GoalHandleFib = rclcpp_action::ClientGoalHandle<FibAction>;
17
18     explicit FibActionClient(const rclcpp::NodeOptions & node_options = rclcpp::NodeOptions()
19         ());
20
21     // used in the main to stop the client when the goal is done.
22     bool is_goal_done() const;
23
24     void send_goal();
25
26 private:
27     rclcpp_action::Client<FibAction>::SharedPtr client_ptr_;
28     rclcpp::TimerBase::SharedPtr timer_;
29     bool goal_done_;
30
31     void goal_response_callback(std::shared_future<GoalHandleFib::SharedPtr> future);
32
33     void feedback_callback(
34         GoalHandleFib::SharedPtr,
35         const std::shared_ptr<const FibAction::Feedback> feedback);
36
37     void result_callback(const GoalHandleFib::WrappedResult & result);
38 };
39
40 #endif /* FIB_ACTION_CLIENT_HPP */

```

Codevoorbeeld 5.14
ActionClientNode.hpp

```

0 #include <iinttypes.h>
1 #include <memory>
2 #include <string>
3 #include <iostream>
4
5 #include "rclcpp/rclcpp.hpp"
6 #include "rclcpp_action/rclcpp_action.hpp"
7
8 #include "FibActionClient.hpp"
9 #include "custom_interfaces/action/fibonacci.hpp"
10
11 using namespace std::placeholders; // needed for the _1 and _2 in std::bind
12
13 FibActionClient::FibActionClient(const rclcpp::NodeOptions & node_options)
14 : Node("fib_action_client", node_options), goal_done_(false)
15 {
16     this->client_ptr_ = rclcpp_action::create_client<FibAction>(
17         this,
18         "fibonacci");
19

```

```
20     this->timer_ = this->create_wall_timer(
21         std::chrono::milliseconds(500),
22         std::bind(&FibActionClient::send_goal, this));
23 }
24
25 bool FibActionClient::is_goal_done() const
26 {
27     return this->goal_done_;
28 }
29
30 void FibActionClient::send_goal()
31 {
32     // cancel the timer.
33     this->timer_->cancel();
34
35     // the goal is not done.
36     this->goal_done_ = false;
37
38     // wait till the action server is also running.
39     if (!this->client_ptr_->wait_for_action_server(std::chrono::seconds(10))) {
40         RCLCPP_ERROR(this->get_logger(), "Action server not available after waiting");
41         this->goal_done_ = true;
42         return;
43     }
44
45     // create a new goal message:
46     auto goal_msg = FibAction::Goal();
47     goal_msg.order = 10;
48
49     // print in logger
50     RCLCPP_INFO(this->get_logger(), "Sending goal");
51
52     // setting the functions that handle the return-messages:
53     auto send_goal_options = rclcpp_action::Client<FibAction>::SendGoalOptions();
54     send_goal_options.goal_response_callback = std::bind(&FibActionClient::
55     goal_response_callback, this, _1);
56     send_goal_options.feedback_callback = std::bind(&FibActionClient::feedback_callback,
57     this, _1, _2);
58     send_goal_options.result_callback = std::bind(&FibActionClient::result_callback, this,
59     _1);
60 }
61
62
63
64 void FibActionClient::goal_response_callback(std::shared_future<GoalHandleFib::SharedPtr>
65     future)
66 {
67     auto goal_handle = future.get();
68     if (!goal_handle) {
69         RCLCPP_ERROR(this->get_logger(), "Goal was rejected by server");
70     } else {
71         RCLCPP_INFO(this->get_logger(), "Goal accepted by server, waiting for result");
72     }
73 }
74
75 void FibActionClient::feedback_callback(
    GoalHandleFib::SharedPtr,
```

```

76     const std::shared_ptr<const FibAction::Feedback> feedback
77     ) {
78         RCLCPP_INFO(
79             this->get_logger(),
80             "Next number in sequence received: %" PRIId32,
81             feedback->partial_sequence.back()
82         );
83     }
84
85 void FibActionClient::result_callback(const GoalHandleFib::WrappedResult & result)
86 {
87     this->goal_done_ = true;
88     // result.code contains information why the result is send.
89     // It is possible to get an incomplete result, for example when the
90     // goal is canceled.
91     switch (result.code) {
92         case rclcpp_action::ResultCode::SUCCEEDED:
93             break;
94         case rclcpp_action::ResultCode::ABORTED:
95             RCLCPP_ERROR(this->get_logger(), "Goal was aborted");
96             return;
97         case rclcpp_action::ResultCode::CANCELED:
98             RCLCPP_ERROR(this->get_logger(), "Goal was canceled");
99             return;
100     default:
101         RCLCPP_ERROR(this->get_logger(), "Unknown result code");
102         return;
103     }
104
105     RCLCPP_INFO(this->get_logger(), "Result received");
106     for (auto number : result.result->sequence) {
107         RCLCPP_INFO(this->get_logger(), "%" PRIId32, number);
108     }
109 }
```

Codevoorbeeld 5.15
ActionClientNode.cpp

```

0 #include "rclcpp/rclcpp.hpp"
1 #include "FibActionClient.hpp"
2
3 int main(int argc, char ** argv)
4 {
5     rclcpp::init(argc, argv);
6     auto action_client = std::make_shared<FibActionClient>();
7
8     // Let the client send one goal and then close it:
9     while (!action_client->is_goal_done()) {
10         rclcpp::spin_some(action_client);
11     }
12
13     rclcpp::shutdown();
14     return 0;
15 }
```

Codevoorbeeld 5.16
mainActionClientNode.cpp

5.4 Meer bronnen:

Een simpele uitleg van wat actions zijn:

<https://index.ros.org/doc/ros2/Tutorials/Understanding-ROS2-Actions/>
&
<http://design.ros2.org/articles/actions.html>

Hoe je je eigen actions maakt:

<https://index.ros.org/doc/ros2/Tutorials/Actions/Creating-an-Action/>

Hoe je je eigen action client en action server maakt:

<https://index.ros.org/doc/ros2/Tutorials/Actions/Writing-a-Cpp-Action-Server-Client/>



6. ROS tools en etc.

In de afbeeldingen 3.1 en 4.1 zien we meerdere nodes en verbindingen die aangeven hoe de nodes met elkaar communiceren. Dergelijke figuren noemen we een *communication graph* van de architectuur van het systeem. Deze type afbeeldingen geven de stroom van informatie aan en welke nodes afhankelijk zijn van welke nodes. Als we naar een groter systeem gaan, zoals te zien in figuur 6.1, wordt het lastiger om hier goed overzicht van te houden. Er zijn verschillende manieren om toch overzicht te houden, waaronder de belangrijkste: het modelleren van het systeem. Gelukkig biedt ook ROS 2 hier ons hulpmiddelen bij. Bekijk daarom goed wat er allemaal mogelijk is met ROS via de commandline. Naast de hulpmiddelen voor modelleren biedt de ROS 2 commandline commando's ook opties voor testen, analyseren en beheer. In de volgende sectie geven we een zeer incomplete lijst van de mogelijkheden.

6.1 Handige ROS commando's

ROS komt allerlei handige commando's om je packages te bouwen en te onderhouden. In tabel 6.1 staan er een aantal. Weet jij een commando die hier aan moet worden toegevoegd, neem dan vooral contact op met de auteur.

Tabel 6.1
Een incomplete
lijst met handige
ROS 2
commando's.

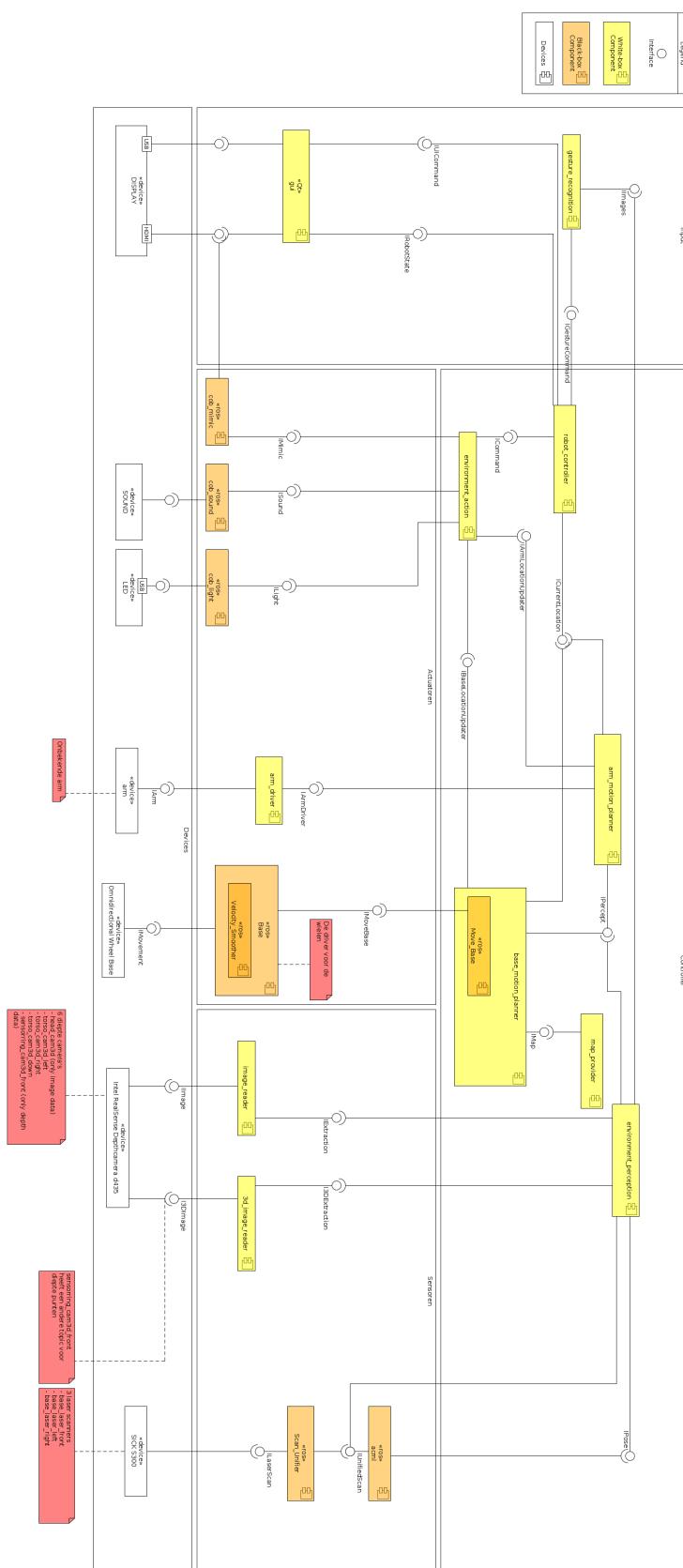
COMMAND	RESULTAAT
ros2 topic hz /[topic_name]	Geeft de frequentie statistieken van een topic.
rqt_graph	laat alle actieve nodes en communicatie zien
ACTIONS	
ros2 action list	Geeft alle actions in de ROS graph
ros2 action info <action>	Geeft het aantal clients en servers voor de action.

6.2 Achtergrond

Diepere uitleg over ROS 2 en de gemaakte designkeuzes zijn onder andere te vinden op:

<https://design.ros2.org/>

Figuur 6.1
De architectuur
van het *World Of
Robots*-project van
19-20 S2 bij de
opleiding ESD aan
de HAN.





7. ROS packages

ROS komt met een aantal handige packages (libraries) die het werken met en het programmeren van robots makkelijker maken. In dit hoofdstuk kijken we naar de transform library bekend als tf2³³

7.1 Tf2

Tf2 (transform library 2) is een ROS package (library) die helpt bij het omrekenen van verschillende coördinatensystemen naar elkaar. Dit is erg handig als we werken met hardwarecomponenten die moet samenwerken in de echte wereld. Deze sectie beschrijft wanneer je tf2 nodig hebt en hoe je tf2 gebruikt.

7.1.1 Meerdere coördinatensystemen

Elk hardwarecomponent dat interacteert met de wereld heeft zijn eigen *coordinate frame of reference* of kortweg *coordinate frame*. Hiermee bedoelen we dat elk hardwarecomponent zijn eigen nulpunt (referentiekader) heeft waar vanuit hij redeneert waar andere objecten staan. Zo zal een robotarm die op de HAN in Arnhem waarschijnlijk zeggen dat zijn middelpunt³⁴ van zijn base de coordinaten (0, 0, 0) heeft. Hiermee bedoelen we dat zijn coordinate frame op het middelpunt van zijn frame zit. Deze locatie van de coordinate frame is logisch voor de robotarm, maar onlogisch voor de rest van de wereld. Zeker als de arm schuin op een tafel is bevestigd op GPS coordinaten (51.98673585896382, 5.951850752998225) en op Rijksdriehoekcoördinaten³⁵ (193787, 444411). Het werken met een coordinate frame voor een robotarm maakt veel dingen makkelijker. De robotarm hoeft niet te achterhalen waar in de wereld het is. Alles wat het moet doen kan worden berecalculated vanuit zijn eigen locatie.

Deze egoïstische kijk van een component op coordinate frames geven wel uitdagingen als er meerdere hardwarecomponenten zijn die moeten samenwerken. Elk hardwarecompo-

³³<https://wiki.ros.org/tf2> ; Pas op: de meeste voorbeelden op deze link zijn met ROS 1.

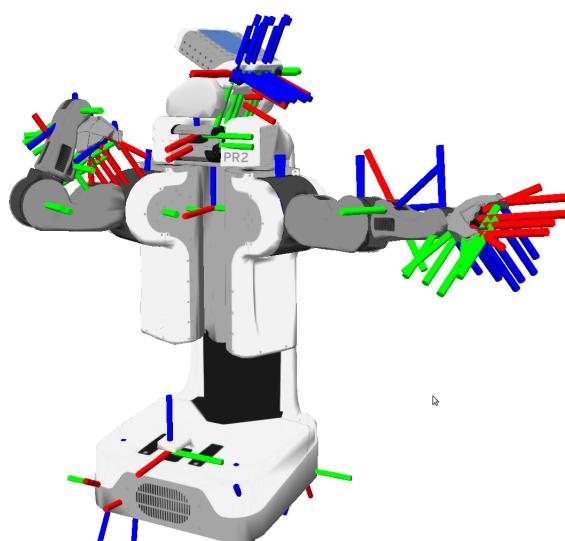
³⁴Midden- versus middel- : <https://www.schrijfwijzer.nl/taalvragen/verwarwoordenboek/verwarwoord/362/middel-midden>

³⁵zie voor informatie over Rijksdriehoekcoördinaten: <https://nl.wikipedia.org/wiki/Rijksdriehoeksco%C3%B6rdinaten>. GPS en Rijksdriehoekcoördinaten hebben ieder ook een andere referentiekader.

nenten heeft namelijk zijn eigen coordinate frame en dus een eigen nulpunt. De coördinaten waar een sensor een object ziet komt dan niet overeen met de coördinaten waar de robotarm heen moet bewegen om het object aan te raken. Dit wordt nog ingewikkelder als we de oriëntatie (Pitch, Yaw en Roll) ook meenemen. Sensors en actuatoren staan vaak in verschillende hoeken. Maar het kan erger. Het is ook mogelijk dat de locatie van een coordinate frame in de echte wereld steeds verandert. Bijvoorbeeld bij een sensor bevestigt op de “hand” van een robotarm.

Tf2 helpt ons met het managen van de vele coordinate frames die een robot kan hebben. Het geeft de mogelijkheid om coordinate frames van de verschillende (bewegende) onderdelen naar elkaar of naar een gezamelijke coordinate frame (veelal *world frame* genoemd) om te zetten. In figuur 7.1 zien we een robot met verschillende coordinate frames. De groene, blauwe en rode cilinders geven de oriëntatie en locaties van de (nulpunten) van de verschillende coordinate frames aan. bron: <https://wiki.ros.org/tf2>

Figuur 7.1
Een robot met
meerdere
coordinate frames
met verschillende
referentiekaders.
De groene, blauwe
en rode cilinders
geven de
oriëntatie en
locaties van de
(nulpunten) van
de verschillende
coordinate frames
aan. bron:
<https://wiki.ros.org/tf2>



7.1.2 Broadcasters en TransformStamped messages

Tf2 helpt ons bij het communiceren van over welke frames we hebben en wat de relatie is tussen de verschillende frames. Dit doen we door middel van *TransformStamped messages*, *broadcasters* en *listeners*. De *TransformStamped* messages bevatten de informatie over de frames. Een broadcaster publieert de *TransformStamped* messages en een listener geeft ons vervolgens de mogelijkheid om de locatie van elke frame te krijgen in het coördinaten stelsel van een andere frame. De listener leggen we uit in sectie 7.1.3.

Er zijn twee verschillende broadcasters: gewone *broadcasters* en *static broadcasters*. De static broadcaster geeft performance voordelen als we deze gebruiken voor frames die weinig of nooit veranderen. In codevoorbeeld 7.1 zien we de header file van een ROS node class die twee broadcasters heeft als members.

```

0 #ifndef _FRAMEBROADCASTER_H_
1 #define _FRAMEBROADCASTER_H_
2
3 #include "rclcpp/rclcpp.hpp"
4 #include <tf2_ros/transform_broadcaster.h>
5 #include <tf2_ros/static_transform_broadcaster.h>
```

```

6
7 class MinimalTF2Broadcaster : public rclcpp::Node
8 {
9     public:
10    MinimalTF2Broadcaster();
11
12    private:
13        void createArmFrame();
14
15        void moveBase();
16
17        rclcpp::TimerBase::SharedPtr timer_;
18
19        tf2_ros::TransformBroadcaster br;
20        tf2_ros::StaticTransformBroadcaster sbr;
21
22        double baseAngle_;
23    };
24
25 #endif

```

Code voorbeeld 7.1

frameBroadcaster.hpp; Een node met twee broadcasters. De twee frames maken samen een 1 DOF-robotarm (zie figuur 7.2 en source file in code voorbeeld 7.2).

Voordat de broadcasters een frame kunnen versturen moet dit in een TransformStamped messages worden gestopt. Een TransformStamped message van een frame heeft de volgende velden:

- **header.stamp:** bevat het tijdstip dat de nieuwe frame waar is. Kan worden gebruikt om terug te kijken in de tijd.
- **header.frame_id:** ook wel parent frame genoemd. We beschrijven in deze message de locatie en oriëntatie van de frame in de parent frame.
- **child_frame_id:** De naam van deze frame.
- **transform.translation.x:** x-coördinaat van het nulpunt van deze frame in de parent frame.
- **transform.translation.y:** y-coördinaat van het nulpunt van deze frame in de parent frame.
- **transform.translation.z:** z-coördinaat van het nulpunt van deze frame in de parent frame.
- **transform.rotation.x:** x-Quaternion³⁶ van het nulpunt van deze frame ten opzichte van de parent frame.
- **transform.rotation.y:** y-Quaternion van het nulpunt van deze frame ten opzichte van de parent frame.
- **transform.rotation.z:** z-Quaternion van het nulpunt van deze frame ten opzichte van de parent frame.
- **transform.rotation.w:** w-Quaternion van het nulpunt van deze frame ten opzichte van de parent frame.

In code voorbeeld 7.2 zien we de bijbehorende source file van de header file van code voorbeeld 7.1. In deze source file wordt er een TransformStamped message gemaakt in de functie *createArmFrame()* en in functie *MoveBase()*. De twee frames beschrijven samen een 1 DOF robotarm (zie figuur 7.2). De base frame heeft een locatie en oriëntatie in de world frame. De arm frame heeft een locatie en oriëntatie in de base frame. Indirect heeft hiermee de arm frame ook een locatie en orientatie in de world frame. De functie *MoveBase()* laat de

³⁶Quaternions zijn een manier om de oriëntatie in een 3D ruimte te beschrijven. Quaternion zijn fijner in gebruik in computerprogramma's. Zie voor meer informatie: <https://nl.wikipedia.org/wiki/Quaternion>

base frame ronddraaien (als een continuous servo). het ronddraaien van de base zorgt er voor dat de arm frame op een andere locatie is in de world frame. De locatie van de arm frame in de base frame blijft echter hetzelfde!

Figuur 7.2
De 1 DOF
robotarm.
Codevoorbeeld 7.2
maakt een frame
genaamd 'base' op
de locatie van de
schroef. Deze kan
dus ronddraaien
relatief tot de
wereld. De frame
'arm' uit
codevoorbeeld 7.2
zit aan het
uiteinde van het
witte armpje dat
vast zit aan de
'base'.



```

0 #include "frameBroadcaster.hpp"
1
2 #include <chrono>
3
4 #include "rclcpp/rclcpp.hpp"
5 #include <tf2_ros/transform_broadcaster.h>
6 #include <tf2_ros/static_transform_broadcaster.h>
7 #include <geometry_msgs/msg/transformStamped.h>
8 #include <tf2/LinearMath/Quaternion.h>
9
10 using namespace std::chrono_literals;
11
12 MinimalTF2Broadcaster::MinimalTF2Broadcaster()
13   : Node("MinimalTF2Broadcaster"), br(this), sbr(this), baseAngle_(0)
14 {
15   timer_ = this->create_wall_timer(
16     500ms, std::bind(&MinimalTF2Broadcaster::moveBase, this));
17
18   createArmFrame()
19 }
20
21 void MinimalTF2Broadcaster::createArmFrame(){
22   geometry_msgs::msg::TransformStamped static_transformStamped;
23
24   static_transformStamped.header.stamp = this->now();
25   static_transformStamped.header.frame_id = "base";
26   static_transformStamped.child_frame_id = "arm";
27
28   static_transformStamped.transform.translation.x = 0;
29   static_transformStamped.transform.translation.y = 5;
30   static_transformStamped.transform.translation.z = 0.0;
31
32   tf2::Quaternion q;
33   q.setRPY(0, 0, 0); // it is easier for humans to think in roll, pitch and yaw.
34   static_transformStamped.transform.rotation.x = q.x();
35   static_transformStamped.transform.rotation.y = q.y();
36   static_transformStamped.transform.rotation.z = q.z();
37   static_transformStamped.transform.rotation.w = q.w();
38
39   sbr.sendTransform(static_transformStamped);

```

```

40 }
41
42 void MinimalTF2Broadcaster::moveBase()
43 {
44     // change the rotation of the base:
45     baseAngle_ += 0.1;
46
47     geometry_msgs::msg::TransformStamped transformStamped;
48
49     transformStamped.header.stamp = this->now();
50     transformStamped.header.frame_id = "world";
51     transformStamped.child_frame_id = "base";
52
53     transformStamped.transform.translation.x = 5;
54     transformStamped.transform.translation.y = 2;
55     transformStamped.transform.translation.z = 0.0;
56
57     tf2::Quaternion q;
58     q.setRPY(0, 0, baseAngle_); // it is easier for humans to think in roll, pitch and yaw.
59
60     transformStamped.transform.rotation.x = q.x();
61     transformStamped.transform.rotation.y = q.y();
62     transformStamped.transform.rotation.z = q.z();
63     transformStamped.transform.rotation.w = q.w();
64
65     br.sendTransform(transformStamped);
66 }
```

Codevoorbeeld 7.2

frameBroadcaster.hpp; Een node met twee broadcasters. De twee frames maken samen een 1 DOF-robotarm (zie figuur 7.2). De base frame definieert zichzelf met een locatie en oriëntatie in relatie tot de world frame. De arm frame doet dit ook, maar dan in relatie tot de base frame.

Broadcasters testen

Als we één of meerdere frames hebben aangemaakt kunnen we deze testen met Rviz of met tf2_echo. Met Rviz kunnen we de locaties, orientaties en relaties van de frames visualiseren. Het gebruik van Rviz kan men vinden in de ROS2 documentatie. We laten hier de de simpelere test met Tf2_echo zien. Tf2_echo geeft ons de locatie en oriëntatie van een frame in het stelsel van een andere frame. Om codevoorbeelden 7.1 en 7.2 te testen kunnen we kijken waar de arm frame zich bevindt in de world frame. We doen dat door de node te runnen en in de een andere terminal het volgende uit te voeren³⁷:

```
0 ros2 run tf2_ros tf2_echo world arm
```

Als de code goed werkt zien we dat de locatie van de arm steeds veranderd³⁸³⁹. We kunnen ook testen of inderdaad de arm frame wel gelijk blijft in relatie tot de base frame:

```
0 ros2 run tf2_ros tf2_echo base arm
```

Codevoorbeeld 7.3
Hello world

Maar we kunnen de zaken ook van de andere kant bekijken. Soms wil juist vanuit het component redeneren in plaats vanuit de wereld. Je kan dus ook vragen waar het nulpunt van de world is in het coördinaten stelsel van de arm frame:

```
0 ros2 run tf2_ros tf2_echo arm world
```

Codevoorbeeld 7.4
Hello world

³⁷Het complete voorbeeld in een werkende package is te vinden in de .zip met voorbeelden.

³⁸De x-coördinaat loopt van 0 tot 10 en de y-coördinaat van -3 tot 7

³⁹nog even kijken of de code in de voorbeelden niet is geupdate. Ik dacht dat hier iets mee was.

7.1.3 Listeners

In sectie 7.1.2 zien we hoe we met een command in de terminal de locatie en oriëntatie van een frame in een andere frame kunnen zien. Dit willen we natuurlijk ook in onze code kunnen doen. Hiervoor gebruiken we de *listeners* van tf2. Listeners krijgen transformStamped messages binnen waarin de locatie en oriëntatie zit van een frame in het stelsel van een andere frame. In code voorbeeld 7.5 zien we een de .hpp van een node class die een listener heeft. Naast de listener heeft de node ook een buffer als member. De listener maakt gebruikt van deze buffer. In code voorbeeld 7.6 zien we de source van deze node. In de constructor (en specifiek de initializer lists) zien we het declareren van de buffer en de listener. In *listenAndPrint()* wordt de TransformStamped van de arm frame in relatie tot de world frame opgehaald en geprint op het scherm.

```

0  #ifndef _FRAMELISTENER_H_
1  #define _FRAMELISTENER_H_
2
3  #include "rclcpp/rclcpp.hpp"
4  #include <tf2_ros/transform_listener.h>
5  #include <tf2_ros/buffer.h>
6
7  class MinimalTF2Listener : public rclcpp::Node
8  {
9  public:
10     MinimalTF2Listener();
11
12  private:
13
14     void listenAndPrint();
15
16     rclcpp::TimerBase::SharedPtr timer_;
17
18     tf2_ros::Buffer tfBuffer;
19     tf2_ros::TransformListener tfListener;
20 };
21
22 #endif

```

Code voorbeeld 7.5

frameListener.hpp; De header file van een node met een listener.

```

0  #include "frameListener.hpp"
1
2  #include <chrono>
3
4  #include "rclcpp/rclcpp.hpp"
5  #include <tf2_ros/transform_listener.h>
6  #include <tf2_ros/buffer.h>
7  #include <geometry_msgs/msg/transform_stamped.h>
8  #include <tf2/LinearMath/Quaternion.h>
9
10 using namespace std::chrono_literals;
11
12 MinimalTF2Listener::MinimalTF2Listener()
13     : Node("MinimalTF2Listener"), tfBuffer(this->get_clock()), tfListener(tfBuffer)
14 {
15     timer_ = this->create_wall_timer(
16         500ms, std::bind(&MinimalTF2Listener::listenAndPrint, this));
17 }
18
19
20 void MinimalTF2Listener::listenAndPrint()

```

```

21 {
22     geometry_msgs::msg::TransformStamped transformStamped;
23
24
25     try{ // try to get the coordinates of arm in the world frame:
26         transformStamped = tfBuffer.lookupTransform("world", "arm", tf2::TimePoint());
27     }
28     // otherwise explain what went wrong and exit this function:
29     catch (const tf2::TransformException & ex) {
30         std::cout << "Failure at " << this->now().seconds() << std::endl;
31         std::cout << "Exception thrown:" << ex.what() << std::endl;
32         std::cout << "The current list of frames is:" << std::endl;
33         std::cout << tfBuffer.allFramesAsString() << std::endl;
34         return;
35     }
36
37     RCLCPP_INFO(this->get_logger(), "The arm is at: (%.5f,%.5f,%.5f)",
38                 transformStamped.transform.translation.x,
39                 transformStamped.transform.translation.y,
40                 transformStamped.transform.translation.z);
41 }
```

Codevoorbeeld 7.6

frameListener.hpp; De header file van een node met een listener.

7.1.4 Meer informatie:

De tutorials van tf2 met ROS 2 zijn nog erg minimalistics⁴⁰. Gelukkig is de werking van tf2 niet verandert met de overgang van ROS 1 naar ROS 2. De ROS 1 informatie van tf2 is ook voor ROS2 een goede bron. Let wel op: ROS 1 heeft een andere manier van het maken en opstarten van nodes. Op de onderstaande link vind je meer informatie over tf2:

<https://wiki.ros.org/tf2>

7.2 Rviz

In de voorbeelden .zip is een werkend voorbeeld met Rviz te vinden.

⁴⁰<https://index.ros.org/doc/ros2/Tutorials/tf2/>



8. Practicum

Opgave 8.1 — Nom nom nom.

Bekijk de onderstaande code en beantwoord de volgende vraag zonder de code uit te voeren op je computer:

1. Als de node wordt gestart. Welke naam heeft de node dan in ROS?

```
0 #include "rclcpp/rclcpp.hpp"
1
2 class VoedselNode : public rclcpp::Node
3 {
4 public:
5     VoedselNode();
6 }

0 #include "rclcpp/rclcpp.hpp"
1 #include "VoedselNode.h"
2
3 HelloNode::VoedselNode() : Node("Appelkoek"){
4     RCLCPP_INFO(this->get_logger(), "Hallo, ik ben Voedsel!");
5 }

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

Opgave 8.2 — Constructieve communicatie.

Bekijk de onderstaande code en beantwoord de volgende vragen zonder de code uit te voeren op je computer:

1. Wat doet de functie `std::bind()`?
2. Waarom hebben `using std::placeholders::_1;` nodig?
3. Als we ArchitectNode en BuilderNode (op volgende pagina) runnen wat zien we dan in de terminal verschijnen?
4. ArchitectNode deelt nu drie designs en wacht na het geven van een design even. We willen dat de node tussen elk design 10 seconden wacht. Welke regel moet worden aangepast en wat is de aanpassing?
5. De nodes maken gebruikt van een custom message. Schrijf de custom message.

```
0 #include "rclcpp/rclcpp.hpp"
1 #include "ArchitectNode.h"
2
3 ArchitectNode::ArchitectNode(): Node("Djoser"){
4     publisher_ = this->create_publisher<njl::msg::design>("designs", 10);
5
6     rclcpp::Rate timer(0.5);
7     shareDesign("*", 3);
8     timer.sleep();
9     shareDesign("#", 7);
10    timer.sleep();
11    shareDesign("+", 5);
12 }
13
14 void ArchitectNode::shareDesign(std::string pattern, unsigned int n){
15     auto message = njl::msg::design();
16     message.pattern = pattern;
17     message.n = n;
18     publisher_->publish(message);
19 }
```

```
0 #include "rclcpp/rclcpp.hpp"
1 #include "BuilderNode.h"
2 #include <string>
3
4 using std::placeholders::_1;
5
6 BuilderNode::BuilderNode(): Node("Pat en Mat"){
7     subscription_ = this->create_subscription<nijl::msg::Design>(
8         "designs", 10, std::bind(&BuilderNode::designReceiver, this, _1));
9 }
10
11 void BuilderNode::topic_callback(const Nijl::msg::Design::SharedPtr msg) const
12 {
13     unsigned int n = msg->n;
14     std::string floor = "";
15     for(unsigned int i=0; i < n; i++){
16         floor = "";
17         for(unsigned int j=0; j < i; j++){
18             floor += msg->pattern;
19         }
20         RCLCPP_INFO(this->get_logger(), floor.c_str());
21     }
22     for(unsigned int i=n-1; i >= 0; i--){
23         floor = "";
24         for(unsigned int j=0; j < i; j++){
25             floor += msg->pattern;
26         }
27         RCLCPP_INFO(this->get_logger(), floor.c_str());
28     }
29 }
```

Opgave 8.3 — Interne conversatie.

Bekijk de onderstaande code en beantwoord de volgende vragen zonder de code uit te voeren op je computer:

1. Wat staat er na het starten van de WeirdNode op het scherm?
2. Wat gebeurd er als we twee WeirdNodes starten?

```

0 #include "rclcpp/rclcpp.hpp"
1 #include "std_msgs/msg/string.hpp"
2 #include "WeirdNode.h"
3
4 using std::placeholders::_1;
5
6 WeirdNode::WeirdNode(const std::string & name): Node(name){
7     publisher_ = this->create_publisher<std_msgs::msg::String>("topic", 10);
8
9     subscription_ = this->create_subscription<std_msgs::msg::String>(
10        "topic", 10, std::bind(&WeirdNode::topic_callback, this, _1));
11
12    std::string content = "Wat heb je?";
13    publishMessage(content);
14}
15
16 void WeirdNode::topic_callback(const std_msgs::msg::String::SharedPtr msg)
17 {
18     const std::string content = msg->data.c_str();
19     RCLCPP_INFO(this->get_logger(), "I heard: '%s'", content.c_str());
20
21     if(content.size() > 1){
22         std::string new_content = content.substr(0, content.size()-1);
23
24         for(int i=0; i<1000000000; i++){} // wait a bit.
25
26         publishMessage(new_content);
27     }
28
29 void WeirdNode::publishMessage(std::string content) const{
30     auto message = std_msgs::msg::String();
31     message.data = content;
32     publisher_->publish(message);
33 }
34
35 #include "rclcpp/rclcpp.hpp"
36 #include "WeirdNode.h"
37
38 int main(int argc, char * argv[])
39 {
40     rclcpp::init(argc, argv);
41     rclcpp::spin(std::make_shared<WeirdNode>("Weirdo"));
42     rclcpp::shutdown();
43     return 0;
44 }
```

Opgave 8.4 — Abacus.

Maak een package met twee ROS nodes: één publisher en één subscriber. De publisher schrijft elke 2 seconden een willekeurig cijfer op een topic. De subscriber ontvangt drie berichten van het topic en telt deze bij elkaar op. Het resultaat print de subscriber via de logger. De subscriber print dus elke 6 seconden een resultaat. Als de subscriber later start dan de publisher, dan begint de subscriber bij het eerste bericht dat hij ontvangt via het topic (dit is het standaard gedrag van een subscriber).

Zorg er voor dat je package voldoet aan de conventies van ROS2.



Een Abacus (ook wel bekend als telraam).

Opgave 8.5 — Up and Down.

Maak een copy van de package van de opdracht Abacus en gebruik dit als basis voor de opdrachten A en B.

A:

Pas de publisher aan zodat hij nu willekeurig het woord plus, min of keer stuurt samen met het willekeurige getal. Voorbeeld van berichten op de topic:

```
0 plus 8
1 min 9
2 plus 3
3 plus 0
4 plus 2
5 keer 3
```

B:

Pas ook de subscriber aan, zodat hij de nieuwe berichten van de publisher kan verwerken. Ontvangt de subscriber een bericht met het woord plus dan verwerkt hij het getal met een plus. Ontvangt de subscriber een bericht het woord min verwerkt hij het getal met een min. Ontvangt de subscriber een bericht het woord keer verwerkt hij het getal met een keer. Als het eerste cijfer met het woord keer komt, dan verwerkt hij dat als $0 * <\text{getal}>$.

Opgave 8.6 — Libellebil.

Bekijk onderstaande code en beantwoord de onderstaande vragen zonder de code uit te voeren.

1. Wat is de service die serviceNode verleend?
2. Maak het .srv bestand dat wordt gebruikt door serviceNode.

```

0  #include "rclcpp/rclcpp.hpp"
1  #include "ServiceNode.h"
2
3  using std::placeholders::_1;
4  using std::placeholders::_2;
5
6  ServiceNode::ServiceNode(): Node("ServiceNode"){
7      service_ = this->create_service<srvcli_libellebil::srv::Word>(
8          "words",
9          std::bind(&ServiceNode::handleService, this, _1, _2)
10     );
11 }
12
13 std::string ServiceNode::esrever(std::string word){
14     for(size_t i=0; i<word.size()/2; i++){
15         std::swap(word[i], word[word.size()-i-1]);
16     }
17     return word;
18 }
19
20 void ServiceNode::handleService(
21     const std::shared_ptr<srvcli_libellebil::srv::Word::Request> request,
22     std::shared_ptr<srvcli_libellebil::srv::Word::Response> response
23 ){
24     std::string outputWord = esrever(request->input_word);
25
26     bool isIt = outputWord == request->input_word;
27
28     response->output_word = outputWord;
29     response->is_it = isIt;
30 }
```

Opgave 8.7 — Wijzerplaat.

1. Maak een ROS2 node die functioneert als een analoge klok. De node publiceert de huidige stand van de wijzers eens per seconde.
2. Maak van de analoge klok node ook een service server. Door middel van de service kan men de klok stopzetten en weer starten.
- .



Een zonnewijzer.

(Afbeelding van Mark Caldicott via pixabay.com)

Opgave 8.8 — Actie.

1. Welke berichten bij een action server/client moet je zelf definiëren in de Action message structure?
2. Welke berichten zijn al voorgedefinieerd bij een action server/client?

Opgave 8.9 — Conways action.

Schrijf een ROS action server die de rij van Conway kan bepalen. De rij van Conway staat ook wel bekend als de Look-and-say sequence, omdat het volgende element in de rij wordt gevormd door het huidige element te beschrijven. Meer informatie over de Look-and-say-sequence kan men vinden op:

- https://nl.wikipedia.org/wiki/Rij_van_Conway
- <https://www.youtube.com/watch?v=ea71JkEhytA>

De action server krijgt als goal de integers x en n mee. Aan de hand hiervan moet de node de rij van Conway bepalen die begint met x en die n lang is. De action server geeft elk tussenresultaat terug als feedback naar de action client. Voorbeelden van input van de service en de output staan in de tabel hieronder. Je mag aannemen dat x en n altijd positief zijn.

x	n	Output
5	3	15, 1115, 3115
367	5	131617, 111311161117, 311331163117, 1321232116132117, 1113121112131221161113122117
1	7	11, 21, 1211, 111221, 312211, 13112221, 1113213211
9	4	19, 1119, 3119, 132119



De wiskundige John Conway (1937 - 2020).

(foto by: Thane Plambeck)

Schrijf ook een action client te testen ook een service client en zorg dat de package voldoet aan de ROS2 conventies.

Opgave 8.10 — Pijpleiding.

Koppel de nodes gemaakt in opdracht 7.5, 7.7 en 7.9 aan elkaar. Bedenk zelf welke nodes met elkaar communiceren. Het kan zijn dat de eerder geschreven nodes een extra of andere communicatierol krijgen en je dus zaken aan je code moet toevoegen.



A. HelloNode package

In hoofdstuk 2 hebben we gekeken naar hoe de C++-code voor een simpele ROS 2 node er uitziet. In dit hoofdstuk maken we van HelloNode van codevoorbeeld 2.5, 2.6 en 2.7 een werkende package⁴¹. Dit hoofdstuk is geen vervanging de tutorials op <https://index.ros.org/doc/ros2/Tutorials/>, maar geeft de lezer wel een inzicht en toelichting in welke stappen doorlopen moeten worden voor het maken van een ROS 2 Foxy Fitzroy package op een Linux-machine. Een package is een collectie van één of meerdere ROS-nodes die logische wijs bij elkaar horen. Een typisch ROS-project bestaat uit meerdere packages.

Mocht je ROS 2 nog niet hebben geïnstalleerd dan kan dat via. In deze uitleg gaan we uit van ROS 2 versie *Foxy Fitzroy*.

<https://docs.ros.org/en/foxy/Installation.html>

ROS 2 environment

Als we met ROS aan de slag gaan moeten we altijd eerst de ROS 2 environment starten:

```
0 source /opt/ros/foxy/setup.bash
```

Workspace

Je kan de packages in een bestaande workspace plaatsen. Als je nog geen workspace⁴² hebt of het netjes in een nieuwe workspace wil plaatsen, dan moet je eerst een folder aanmaken. Wij noemen onze folder *reader_ws*. In deze workspace plaatsen we alle packages die in deze reader worden behandeld. Het is een ROS-conventie om de packages in de *src* (source) folder van de workspace te plaatsen. We maken onze workspace met *src*-folder met de volgende commandline:

```
0 mkdir -p ~/reader_ws/src
```

Het maken van een lege package

We bewegen onszelf naar de folder waar we de packages gaan plaatsen:

⁴¹De code is ook te vinden op de git van deze reader.

⁴²Een workspace is een bestand of folder dat de gebruiker de mogelijkheid geeft om allerlei source code bestanden en resources te verzamelen en mee te werken als één samenhangend geheel (vrij vertaald van wikipedia). Dit betekent dat als je een workspace hebt ongerelateerde bestanden (bijvoorbeeld zowel bestanden uit deze reader als andere projecten) dat je een rommel aan het maken bent.

```
0 cd ~/reader_ws/src
```

Het maken van een package gaat met de commando `ros2 pkg create --build-type ament_cmake <package_name>`⁴³. Het commando maakt voor ons de folderstructuur van de package en de bestanden *CMakeLists.txt* en *package.xml*. Deze bestanden helpen ons met het bouwen van de package en het starten van verschillende packages. We noemen onze package Hello:

```
0 ros2 pkg create --build-type ament_cmake hello
```

Code in de package

Gebruik je favouriete tekst-editor om *HelloNode.cpp* te maken zoals beschreven in codevoorbeelden 2.5, 2.6 en 2.7. Plaats *HelloNode.cpp* en *main.cpp* in de folder `/reader_ws/src/Hello/src` en *HelloNode.hpp* in de folder `/reader_ws/src/Hello/include/hello`. De inhoud van onze folder is nu:

```
0 \reader_ws
1   \src
2     \hello
3       CMakeLists.txt
4       package.xml
5       \include
6         \hello
7           HelloNode.hpp
8       \src
9         HelloNode.cpp
10        main.cpp
```

Codevoorbeeld A.1

De folder structuur inclusief bestanden van de package Hello.

Package.xml

In *package.xml* geven we informatie over de package Hello zoals de naam, beschrijving, license en de dependencies. Deze informatie is niet alleen handig voor anderen die de package gaan gebruiken, maar de dependencies zijn noodzakelijk voor het correct werken met ROS 2. Open *package.xml* in je favoriete teksteditor en zorg dat het de onderstaande inhoud heeft. Let vooral op regel 11. Hiermee zeggen we dat onze package afhankelijk is van *rclcpp*.

```
0 <?xml version="1.0"?>
1 <?xml-model href="http://download.ros.org/schema/package_format3.xsd" schematypens="http
  ://www.w3.org/2001/XMLSchema"?>
2 <package format="3">
3   <name>hello</name>
4   <version>0.0.0</version>
5   <description>A node that says hello to the world!</description>
6   <maintainer email="Jorn.Bunk@han.nl">Jorn Bunk</maintainer>
7   <license>Attribution 4.0 International (CC BY 4.0)</license>
8
9   <buildtool_depend>ament_cmake</buildtool_depend>
10
11  <depend>rclcpp</depend>
12
13 <test_depend>ament_lint_auto</test_depend>
14 <test_depend>ament_lint_common</test_depend>
15
```

⁴³Dit commando komt met nog veel meer functionaliteit. Bijvoorbeeld kunnen de dependencies automatisch aan *package.xml* en *CMakeLists.txt* toegevoegd worden door gebruik te maken van de flag *-dependencies*.

```

16 <export>
17   <build_type>ament_cmake</build_type>
18 </export>
19 </package>
```

CMakeLists.txt

In de CMakeLists.txt geven we de build instellingen van onze package. Open CMakeLists.txt in je favoriete teksteditor en zorg dat het de onderstaande inhoud heeft. De standaard CMakeLists.txt bevat ook een aantal onderdelen die we niet nodig hebben⁴⁴, deze opgeruimd. Er 3 belangrijke onderdelen aan dit bestand. Allereerst hebben we de *find_package()*-functies waarin de dependencies moeten worden genoemd. In het geval van deze package is dat enkel *rclcpp* en *ament_cmake*. Die laatste is standaard. Het tweede onderdeel is het toevoegen van de executable met *add_executable()* en *ament_target_dependencies()*, zodat ROS 2 de package kan uitvoeren. We moeten hierbij onze executable een naam geven. Er is hier voor de naam *greeter* gekozen. Als laatste moeten we ook zorgen dat ROS 2 onze executable kan vinden. Dit doen we met de functie *install()*. Een package kan ook bestaan uit meerdere executables. Hiervoor moeten regel 18 en 19 worden herhaald met de instellingen van de andere executables en moeten de namen van de executables worden toegevoegd aan het lijstje met TARGETS (regel 22/23).

```

1 cmake_minimum_required(VERSION 3.5)
2 project(hello)
3
4 # Default to C++14
5 if(NOT CMAKE_CXX_STANDARD)
6   set(CMAKE_CXX_STANDARD 14)
7 endif()
8
9 if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
10   add_compile_options(-Wall -Wextra -Wpedantic)
11 endif()
12
13 # find dependencies
14 find_package(ament_cmake REQUIRED)
15 find_package(rclcpp REQUIRED)
16
17 # The path to our header files:
18 # make sure the folder inside /include had the same name as the project!
19 include_directories(include/${PROJECT_NAME}/)
20
21 # add the executable
22 add_executable(greeter src/main.cpp src/HelloNode.cpp)
23 ament_target_dependencies(greeter rclcpp)
24
25 # Making ROS 2 able to find our package:
26 install(TARGETS
27   greeter
28   DESTINATION lib/${PROJECT_NAME})
29
30 ament_package()
```

Bouwen en rennen⁴⁵

Waarschijnlijk heb je al de dependencies die nodig zijn voor de package Hello, desondanks is het een goede gewoonte om te controlleren op missende dependencies. Dit doe je door

⁴⁴De standaard CMakeLists.txt bevat ook een aantal zaken niet die bij sommige packages wel nodig zijn, bijvoorbeeld als de package ook messages definieert.

⁴⁵Dit is een grapje.

dit in de hoofdfolder van je workspace (*reader_ws*) het volgende te runnen:

```
0 rosdep install -i --from-path src --rosdistro foxy -y
```

Het bouwen van de package doen we, nog steeds vanuit de folder *reader_ws*, nu met:

```
0 colcon build --packages-select hello
```

De package is nu gebouwd. Om onze package te runnen moeten we eerst de setup files uitvoeren:

```
0 . install/setup.bash
```

De node kunnen we nu starten met:

```
0 ros2 run hello greeter
```

Gebruik CTRL+C om de node te stoppen.



B. Frequently Asked Questions

B.1 ROS?!

B.1.1 **ondanks dat ik een package bouw blijft hij de oude gecompileerde bestanden uitvoeren?**

Waarschijnlijk bouw en run je op twee verschillende plekken. Waarschijnlijk ben je aan het bouwen in de package in plaats van in de hoofdfolder van je workspace.

B.1.2 **Ik heb al heel veel geprobeerd, maar de fout blijft maar bestaan?**

Waarschijnlijk bouw en run je op twee verschillende plekken. Waarschijnlijk ben je aan het bouwen in de package in plaats van in de hoofdfolder van je workspace.

B.1.3 **Rare errors**

Ik krijg rare errors?

Waarschijnlijk ben je vergeten de ROS 2 environment te activeren door middel van:

```
0 source /opt/ros/foxy/setup.bash
```

Codevoorbeeld B.1
Hello world

Dit moet je doen in elke terminal voordat je iets met ROS 2 doet (zowel runnen van nodes als het bouwen van packages).

B.1.4 **Colcon build geeft warning**

WARNING:colcon.colcon_ros.prefix_path.ament:The path '/een/path/' in the environment variable AMENT_PREFIX_PATH doesn't exist Waarschijnlijk heb je een package een andere naam gegeven. Deze warning kan geen kwaad. Wil je van de warning af? Verwijder dan de folders *install*, *build* en *log* (*rm -r build log install*) en start een nieuwe terminal⁴⁶.

B.1.5 **Colcon build geeft error message/service/action van andere package**

1. *CMake Error at CMakeLists.txt:24 (find_package): By not providing "Findpackage_name-in CMAKE_MODULE_PATH this project has asked CMake to find a package config*-

⁴⁶Dit kan waarschijnlijk ook door AMENT_PREFIX_PATH te resetten. Laat even weten als je het command hiervoor hebt.

guration file provided by "packagename", but CMake did not find one.

Je bent vergeten in je package.xml de package met de message/service/action toe te voegen aan aan het rijtje met <depend> </depend>.

2. *CMake Error at /opt/ros/foxy/share/amen_cmake_target_dependencies/cmake/ament_target_dependencies.cmake:77 (message): ament_target_dependencies() the passed package name 'package_name'*

Je bent vergeten de package met de message/action/service toe te voegen als dependicie aan je CMakeLists.txt

B.1.6 Message/service/action met een string

Een string die ik via een message/service/action heb verstuurd print als rare tekens?

Waarschijnlijk ben je vergeten om de functie `c_str()` te gebruiken. Je drukt nu de pointer af. Voorbeeld: Als je string in `data` zit, dan krijg je de string met `data.c_str()`.

B.1.7 Custom message/service/action hpp includeren

Als ik een custom message/service/action heb gemaakt hoe heet dan de hpp die ik moet includeren? Een custom message/service/action wordt door ROS omgezet naar een .hpp die je moet includeren in de C++ bestanden die gebruik maken van de message/service/action. Het door ROS gemaakte .hpp-bestand hanteert een andere bestandsnaamconventie. De naam van een custom message/service/action moet beginnen met een hoofdletter. Het hpp-bestand dat je moet includeren heeft vervolgens geen hoofdletters, maar voor elke hoofdletter, met uitzondering van de eerste, wordt een underscore (_) geplaatst. Het pad van het hpp-bestand is gelijk aan dat van je custom message/action/service. In de codevoorbeelden die komen bij deze reader op de git-omgeving kan je hier voorbeelden van vinden.

B.1.8 Error rond > en ::SharedPointer bij het maken van een action Node

Waarschijnlijk ben je rclcpp_action vergeten toe te voegen aan je package.xml.

B.1.9 Vtable

undefined reference to 'vtable for ...'. Waarschijnlijk heeft je class/node nog geen destructor.

B.2 Arduino?!

B.2.1 Serial Communicatie traag

Als ik communiceer met serial op arduino kost dat veel tijd.

Haal niet steeds het bericht op uit de buffer. Kijk eerst of er iets in de buffer staat door gebruik te maken van de functie `Serial.available()`: <https://www.arduino.cc/reference/en/language/functions/communication/serial/available/>