ROS2 Kommunikationsmethoden

Parameter	Kursinformationen
Veranstaltung:	`Softwareprojekt Robotik`
Semester	`Wintersemester 2021/22`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	`ROS Kommunikationsprinzipien`
Link auf GitHub:	https://github.com/TUBAF-Ifl-LiaScript/VL_Softwareentwicklung/blob/master/08_ROS_Kommunkation.md
Autoren	Sebastian Zug & Georg Jäger

Was ist eigentlich eine Middleware?

> Middleware im Kontext verteilter Anwendungen ist eine Software, die über die vom Betriebssystem bereitgestellten Dienste hinaus Dienste bereitstellt, um den verschiedenen Komponenten eines verteilten Systems die Kommunikation und Verwaltung von Daten zu ermöglichen.

Middleware unterstützt und vereinfacht komplexe verteilte Anwendungen, sonst müsste die Anwendung Fragen wie:

- Welche Informationen sind verfügbar?
- In welchem Format liegen die Daten vor, bzw. wie muss ich meine Informationen verpacken?
- Wer bietet die Informationen an?
- Welche Laufzeit haben die Daten maximal?

...

Eine Middleware befreit die Applikation davon diese Frage zu beantworten. Vielmehr bieten Middleware-Dienste einen Satz von Programmierschnittstellen, um einer Anwendung:

- eine beliebige "Lokalisierung" im gesamten Netzwerk zu ermöglichen
- eine standardisierte Interaktion mit einem anderen Dienst oder einer anderen Anwendung umzusetzen
- Daten zu filtern (Inhalte, Autorisierung)
- eine unabhängigen Netzzugriff unabhängig vom Netztyp sicherzustellen
- einen zuverlässigen Datenaustausch sicherzustellen.

Und in ROS2?

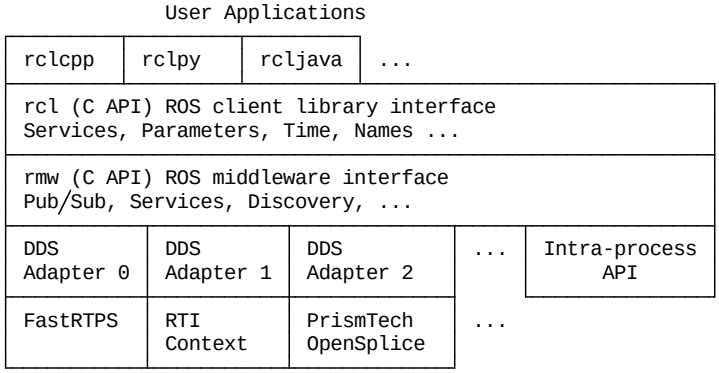
Für die Realisierung dieser Aufgabe stehen unterschiedlichen Lösungsansätze mit verschiedenen Schwerpunktsetzungen bereit. Entsprechend integriert ROS2 ein abstraktes Interface für ein Einbettung von Middelware-Lösungen, die den DDS Standard implementieren.

DDS stellt einen "Globalen Daten Raum" zur Verfügung, der diese allen interessierten verteilten Anwendungen zur Verfügung stellt.

- Datenobjekte werden mit einer Domain-ID, einem Topic und einen Schlüssel adressiert.
- Die Nachfrager (Subscriber) sind von den Produzenten (Publisher) entkoppelt.
- Filter ermöglichen die inhaltliche Definition von relevanten Informationen auf Subscriberseite.
- Die Verbindung wird über *Contracts* spezifiziert, die die *Quality of Service* (QoS) definiert
- Die Verbindung zwischen Subscribern und Publishern wird automatisch hergestellt.

Der DDS Standard wurde durch verschiedene Unternehmen und Organisationen unter dem Dach der Object Management Group vorangetrieben. Eine Beschreibung findet sich unter Link.
ROS2 hat als Default Lösung die Implementierung `rmw_fastrtps_cpp`, die von der Firma eProsima unter einer Appache 2.0 Lizenz verbreitet wird, integriert. Alternative Umsetzungen lassen sich anhand der unterstützten Hardware, des Overheads für den Nachrichtenaustausch bzw. anhand der Dauer für die Nachrichtenverbreitung abgrenzen. (vgl A performance comparsion of OpenSplice and RTI implementations). Daher sieht ROS2 ein abstraktes Interface vor, dass ein Maximum an Austauschbarkeit gewährleisten soll.

vgl. https://index.ros.org/doc/ros2/Concepts/DDS-and-ROS-middleware-implementations/



.

Welche Aufgaben bildet DDS für ROS2 über entsprechende Schnittstellen ab?

Discovery ... DDS ist vollständig verteilt, auch auf der Ebene des Discovery Systems und steht damit im Unterschied zu ROS1, dass ein zentrales Koordinationselemente `roscore` einsetzte. Damit entfällt der zentralen Fehlerpunkt, der für die Kommunikation zwischen Teilen des Systems erforderlich ist.

1. Wenn ein Knoten gestartet wird, wirbt er für seine Anwesenheit bei anderen Knoten im Netzwerk mit derselben ROS-Domäne (gesetzt mit der Umgebungsvariablen `ROS_DOMAIN_ID`). Knoten reagieren auf diese Werbung mit Informationen über sich selbst, damit die entsprechenden Verbindungen hergestellt werden können und die Knoten kommunizieren können.
2. Knoten bewerben ihre Präsenz regelmäßig, so dass auch nach der ersten Erkundungsphase Verbindungen zu neu gefundenen Einheiten hergestellt werden können.
3. Knoten informieren die anderen Knoten, wenn sie offline gehen.

Publish/Subscribe ... DDS implmentiert das Publish/Subscribe Paradigma in Kombination mit Quality of Service Attributen. Diese dienen der Koordination des Datenaustausches unter Berücksichtigung von zwingenden Anforderungen des Subscribers bzw. dem Verhalten des Publishers.

Services and Actions ... DDS verfügt derzeit nicht über einen Request-Response-Mechanismus, mit dem die entsprechenden Konzept der Dienste in ROS umgesetzt werden könnten. Derzeit wird in der OMG DDS-Arbeitsgruppe eine RPC-Spezifikation zur Ratifizierung geprüft, und mehrere der DDS-Anbieter haben einen Entwurf für die Implementierung der RPC-API.

Welche Rolle spielen die QoS Eigenschaften des Kommunkations-Layers?

ROS 2 bietet eine Vielzahl von Quality of Service (QoS)-Richtlinien, mit denen Sie die Kommunikation zwischen Knoten und die Datenhaltung optimieren können. Im Gegensatz zu ROS1, das vorrangig auf TCP setzte, kann ROS2 von Transparenz der jeweiligen DDS-Implementierungen profitieren.

Eine Reihe von QoS "Richtlinien" kombinieren sich zu einem QoS "Profil". Angesichts der Komplexität der Auswahl der richtigen QoS-Richtlinien für ein bestimmtes Szenario bietet ROS 2 einen Satz vordefinierter QoS-Profile für gängige Anwendungsfälle (z.B. Sensordaten). Gleichzeitig erhalten die Benutzer die Flexibilität, spezifische Profile der QoS-Richtlinien zu steuern.

QoS-Profile können für Publisher, Abonnenten, Service-Server und Clients angegeben werden. Damit wird die kombinierbarkeit der Komponenten unter Umständen eingeschränkt!

https://index.ros.org/doc/ros2/Concepts/About-Quality-of-Service-Settings/

https://index.ros.org/doc/ros2/Tutorials/Quality-of-Service/

DDS ist ein Beispiel einer Middleware.

- *Durability* ... legt fest, ob und wie lange Daten, die bereits ausgetauscht worden sind, "am Leben bleiben". `volatile` bedeutet, dass dahingehend kein Aufwand investiert wird, `transient` oder `persistent` gehen darüber hinaus.
- *Reliability* ... Die Reliability-QoS definiert, ob alle geschriebenen Datensätze (irgendwann) bei allen Readern angekommen sein müssen. Bei zuverlässiger (`reliable`) Kommunikation werden geschriebene Datensätze eines Topics, die aus irgendwelchen Gründen auf dem Netzwerk verloren gehen, von der Middleware wiederhergestellt, um diese Daten verlässlich den Readern zustellen zu können. Im Unterschied dazu definiert `best effort` eine schnellstmögliche Zustellung.
- *History* ... definiert, wie viele der letzten zu sendenden Daten und empfangenen Daten gespeichert werden. `Keep last` speichert n Samples, wobei die n durch den QoS Parameter *Depth* definiert wird. `Keep all` speichert alle Samples
- *Depth* ... erfasst die Größe der Queue für die History fest, wenn `Keep last` gewählt wurde.

Konfiguration	Durability	Reliability	History	Depth
Publisher & Subscriber	volatile	reliable	keep last	-
Services	volatile	reliable	keep last	
Sensor data	volatile	best effort	keep last	small
Parameters	volatile	reliable	keep last	larger
Default	volatile	reliable	keep last	small

Quelle: https://github.com/ros2/rmw/blob/release-latest/rmw/include/rmw/qos_profiles.h

Die QoS Parameter werden gegeneinander abgewogen und ggf. abgestimmt.

Publisher	Subscriber	Verbindung	Result
best effort	best effort	ja	best effort
best effort	reliable	nein	
reliable	best effort	ja	best effort
reliable	reliable	ja	reliable

Evaluieren Sie die QoS Mechanismen, in dem Sie die Qualität Ihrer Netzverbindung manipulieren. Eine Anleitung findet sich zum Beispiel unter Link

ROS Publish-Subscribe

Das Publish/Subscribe-Paradigma, bei dem der Publisher hat überhaupt kein Wissen darüber, wer der/die Subscriber sind generiert folgende Vorteile:

- Es entkoppelt Subsysteme, die damit unabhängig von einander werden. Damit steigt die Skalierbarkeit des Systems und gleichzeitig die Handhabbarkeit.
- Die Abarbeitung erfolgt asynchron und ohne Kontrollflussübergabe. Der Knoten ist damit allein auf den eigenen Zustand fokussiert.
- Der Publisher zusätzlich zum Veröffentlichen nicht auch noch komplexe Zielinformationen angeben muss.
- Publisher und Subscriber können die Arbeit jederzeit einstellen. Aus rein Kommunikationstechnischen Gründen beeinflusst dies das System nicht.
- Publisher und Subscriber können eine spezifische Nachrichtenstruktur verwenden, die auf die Anwendung zugeschnitten ist.

Auf der anderen Seite ergeben sich genau daraus auch die zentralen Nachteile:

- Die Zustellung einer Nachricht kann unter Umständen nicht garantiert werden.
- Der Ausfall einer Komponente wird nicht zwangsläufig erkannt.
- Das Pub/Sub-Pattern skaliert gut für kleine Netzwerke mit einer geringen Anzahl von Publisher- und Subscriber-Knoten und geringem Nachrichtenvolumen. Mit zunehmender Anzahl von Knoten und Nachrichten steigt jedoch die Wahrscheinlichkeit von Instabilitäten,

ROS implementiert eine themenbasierte Registrierung (topic based), andere Pub/Sub Systeme eine parameterbasierte (content based).

ROS Services

Bisher haben wir über asynchrone Kommunikationsmechanismen gesprochen. Ein Publisher triggert ggf. mehrere Subscriber. Die damit einhergehende Flexibilität kann aber nicht alle Anwendungsfälle abdecken:

- Berechne einen neuen Pfad
- Aktiviere die Kamera
- ...

In diesem Fall liegt eine Interaktion in Form eines Remote-Procedure-Calls (RPC) vor. Die Anfrage / Antwort erfolgt über einen Dienst, der durch ein Nachrichtenpaar definiert ist: eine für die Anfrage und eine für die Antwort. Ein bereitstellender ROS-Knoten bietet einen Dienst unter einem String-Namen an, und ein Client ruft den Dienst auf, indem er die Anforderungsnachricht sendet und in seiner Ausführung innehält und auf die Antwort wartet. Die Client-Bibliotheken stellen diese Interaktion dem Programmierer so dar, als wäre es ein Remote Procedure Call.

Dafür sind 3 Schritte notwendig:

1. Ein Service wird über ein Service File definiert, dass analog zu den benutzerdefinierten Paketen die Struktur der auszutauschenden Daten beschreibt. Dabei wird sowohl die Struktur des Aufrufes, wie auch die Antwort des Services beschrieben. Dabei wird das gleiche Format wie bei den nutzerspezifischen Messages verwendet.
2. Die Logik des Service (Entgegennahme des Requests/ Ausliefern der Antwort) wird in einem Knoten implementiert. Hier werden die Parameter der Anfrage ausgewertet, die Antwort bestimmt und diese auf das Ausgabeformat abgebildet.

Diese Vorgänge sollen nun in zwei Beispielen besprochen werden. Einmal anhand des Turtlesim-Beispiels und anhand einer eigenen Implementierung.

Manuelle Interaktion mit ROS-Services

```
ros2 run turtlesim turtlesim_node
```

Wie explorieren Sie die Services, die durch den `turtlesim_node` bereitgestellt werden?

`ros2` stellt zwei Schnittstellen für die Arbeit mit den Services bereit.

- `service` erlaubt den Zugriff auf die tatsächlich angebotenen Services während
- `srv` die Definitionsformate, die nicht zwingend auch genutzt werden darstellt.

```
> ros2 service list
/clear
/kill
/reset
/spawn
/turtle1/set_pen
/turtle1/teleport_absolute
/turtle1/teleport_relative
/turtlesim/describe_parameters
/turtlesim/get_parameter_types
/turtlesim/get_parameters
/turtlesim/list_parameters
/turtlesim/set_parameters
/turtlesim/set_parameters_atomically
>
> ros2 srv list | grep turtlesim
turtlesim/srv/Kill
turtlesim/srv/SetPen
turtlesim/srv/Spawn
turtlesim/srv/TeleportAbsolute
turtlesim/srv/TeleportRelative
```

Offenbar stellt die Turtlesim-Umgebung 5 Services bereit, deren Bedeutung selbsterklärend ist. Die zugehörigen Definitionen sind namensgleich zugeordnet. Auf die zusätzlich aufgezeigten Parameter wird im nächstfolgenden Abschnitt eingegangen.

Das Format lässt sich entsprechend aus den srv Dateien ablesen:

```
> ros2 srv show turtlesim/srv/Spawn
float32 x
float32 y
float32 theta
string name # Optional. A unique name will be created and returned if this is
  empty
---
string name
```

Versuchen wir also eine Service mittels `ros2 service call` manuell zu starten. Der Aufruf setzt sich aus mehreren Elementen zusammen, deren Konfiguration zwischen den ROS2 Versionen schwanken. An erster Stelle steht der Dienstname gefolgt von der Service-Definition und dem eigentlichen Parametersatz.

Offenbar wird eine neue Schildkröte in der Simulation erzeugt und mit einem generierten Namen versehen. In diesem Fall erfolgt als Reaktion auf den Request nicht nur eine allgemeine Antwort, vielmehr wird ein weiterer Knoten erzeugt der weitere Publisher und Subscriber öffnet.

Mit den anderen Services (`Kill`) kann dessen Verhalten nun adaptiert werden.

Integration in Client/Server Implemetierung

https://medium.com/@danieljeswin/introduction-to-programming-with-ros2-services-77273d7e8ddc

Nehmen wir an, dass wir ein Multiroboter-Szenario die Zustände der einzelnen Roboter in einer Datenbank speichert. Diese senden Ihre aktuelle Position an die Datenbank. Daraufhin können die Dispatcher-Knoten, diese Datenbank kontaktieren, um einen geeigneten Roboter für eine Aufgaben zu identifizieren. Wir gehen davon aus, dass dem Knoten die Roboter in einem Raum bekannt sind. Nun möchte er mehr über deren Zustand wissen. Entsprechend verschickt er eine Serviceanfrage an unsere Datenbank, die folgende Struktur aufweist:

```
string robotname
---
bool validname false
string location
bool busy
float32 batterylevel
```

Für die Serverseite wurde eine Beispielimplementierung realisiert, die unter Link zu finden ist. Hierfür wurde eine sehr rudimentäre Datenstruktur implementiert, die die Roboterliste enthält.

Service.cpp

```
...
struct Robot{
    public:
        int roomNumber;
        std::string name;
        float batterylevel;
        bool busy;
    public:
        //default constructor
        Robot(int number, std::string robot_name, float battery, bool actualbusy
          ){
            roomNumber=number; name=robot_name; batterylevel=battery; busy
              =actualbusy;}
};


std::string locations[4] = { "Kitchen", "Livingroom", "Bedroom", "Batroom"};

std::vector<Robot> RobotDB {
    Robot(0, "Erika", 11.232, true),
    Robot(1, "Maja", 12.0, false),
    Robot(2, "Julius", 4.0, true),
    Robot(3, "Alexander", 9.8, false),
};
...
```

Die entsprechenden Anfragen haben folglich das Format:

```
> ros2 service call /WhoIsWhere my_services/srv/RobotAvailability "{robotname:
  'Erika'}"
  waiting for service to become available...
  requester: making request: my_services.srv.RobotAvailability_Request(robotname
    ='Erika')
```

Welche Schwächen sehen Sie in der Implementierung?

Anwendung in Parameter Konfiguration

Eine besondere Variante der Services stellen die Parameter dar. Dies sind knoteninterne Größen, die über Services angepasst werden können. Darunter fallen zum Beispiel die Konfigurationsdaten

- einer Kamera,
- die gewünschte maximale Reichweite eines Ultraschallsensors,
- die Schwellwerte bei der Featureextraktion, Linienerkennung, etc.
- ...

Der Vorteil der Parameter liegt darin, dass diese ohne Neukompilierung angepasst werden können.

Zur Illustration des Mechanismus soll wiederum auf die Turtelsim-Umgebung zurückgegriffen werden.

```
> ros2 param list
/turtlesim:
  background_b
  background_g
  background_r
  use_sim_time
> ros2 param describe /turtlesim background_b
  Parameter name: background_b
    Type: integer
    Description: Blue channel of the background color
    Constraints:
      Min value: 0
      Max value: 255
> ros2 param get /turtlesim background_b
  Integer value is: 86
> ros2 param set /turtlesim background_b 10
  Set parameter successful
```

Der Hintergrund der Simulationsumgebung ändert sich entsprechend.

Der Vorteil des Parameterkonzepts liegt nun darin, dass wir:

- das Set der Parameter während der Laufzeit anpassen können. Damit kann das Testen der Anwendung im Feld (zum Beispiel bei der Konfiguration von Sensoren) ohne Neustart/Neukompilierung realisiert werden.
- die gewählten Sets einzeln oder im Block abspeichern können. Diese Konfigurationsfiles werden als yaml Dateien abgelegt und können für unterschiedliche Einsatzszenarien aufgerufen werden.

```
> ros2 param dump /turtlesim
  Saving to:  ./turtlesim.yaml
> cat turtlesim.yaml
  turtlesim:
    ros__parameters:
      background_b: 255
      background_g: 86
      background_r: 69
      use_sim_time: false
```

Der Aufruf kann dann entweder in der Kommandozeile erfolgen

```
ros2 run turtlesim turtlesim_node __params:=turtlesim.yaml
```

oder aber im Launch file

LaunchExample

```
ld = LaunchDescription([
        launch_ros.actions.Node(
            package='nmea_navsat_driver', node_executable
              ='nmea_serial_driver_node', output='screen',
            parameters=["config.yaml"])
    ])
```

Die Implementierung erfolgt anhand eines knoten-internen Parameterservers der wie folgt initialisiert wird:

ParameterImplementation.cpp

```
//Definition and Initialization
auto parameters_client = std::make_shared<rclcpp::SyncParametersClient>(node);
while (!parameters_client->wait_for_service(1s)) {
  if (!rclcpp::ok()) {
    RCLCPP_ERROR(node->get_logger(), "Interrupted while waiting for the service.
      Exiting.");
    return 0;
  }
  RCLCPP_INFO(node->get_logger(), "service not available, waiting again...");
}

// Setup callback for changes to parameters.
auto sub = parameters_client->on_parameter_event(
  [node](const rcl_interfaces::msg::ParameterEvent::SharedPtr event) -> void
  {
    on_parameter_event(event, node->get_logger());
  });

// Declare parameters that may be set on this node
node->declare_parameter("foo");
node->declare_parameter("bar");

// Set several different types of parameters.
auto set_parameters_results = parameters_client->set_parameters({
  rclcpp::Parameter("foo", 2),
  rclcpp::Parameter("bar", "hello"),
});
```

Anmerkung zu Zeile 11: Hier wird eine Lambda-Expression genutzt, um die Funktion, die im Falle einer Parameteranfrage aufgerufen wird, zu spezifizieren.

ROS Actions

Actions sind lang laufende Aufgaben vorgesehen sind, der Client wartet aber nicht auf das Eintreten des Resultats sondern setzt seine Arbeit fort. Entsprechend wird eine *Action* als asynchroner Call bezeichnet, der sich an einen *Action Server* richtet.

Das *Action* Konzept von ROS spezifiziert 3 Nachrichtentypen, die der Client an den Server richten kann:

Nachricht	Richtung	Bedeutung
Goal	Client → Server	... definiert die Parameter des Ziels einer Operation. Im Falle einer Bewegung der Basis wäre das Ziel eine *PoseStamped*-Nachricht, die Informationen darüber enthält, wohin sich der Roboter in der Welt bewegen soll. Darüber hinaus werden Randbedingungen definiert, wie zum Beispiel die maximale Geschwindigkeit.
Feedback	Server → Client	... ermöglicht es eine Information zum aktuellen Stand der Abarbeitung zu erhalten. Für das Bewegen der Basis kann dies die aktuelle Pose des Roboters entlang des Weges sein.
Result	Server → Client	... wird vom ActionServer an den ActionClient gesendet, wenn das Ziel erreicht ist. Dies ist anders als Feedback, da es genau einmal gesendet wird.

Beschrieben wird das Interface wiederum mit einem eigenen Filetyp, den sogenannten `.action` Files. Im Beispiel sehe Sie eine *Action*, die sich auf die Bewegung eines Outdoorroboters zu einer angegebenen GPS-Koordinate bezieht.

```
# Define the goal
float64 latitude
float64 longitude
```

Hinzu kommt noch die Möglichkeit eine *Action* mit *chancel* zu stoppen. Hierfür ist aber keine explizite Schnittstelle notwendig.

Beispiel

Achtung: Das folgende Beispiel ist erst unter ROS2 Eloquent lauffähig! Das Beispiel wurde der ROS2 Dokumentation unter Link entnommen.

```
ros2 run turtlesim turtlesim_node
ros2 run turtlesim turtle_teleop_key
```

Danach können Sie für unseren Turtlesim Umgebung sämtliche Kommunikationsinterfaces auf einen Blick erfassen:

```
> ros2 node info /turtlesim
/turtlesim
  Subscribers:

  Publishers:

  Services:

  Action Servers:
    /turtle1/rotate_absolute: turtlesim/action/RotateAbsolute
  Action Clients:
> ros2 action list -t
  /turtle1/rotate_absolute [turtlesim/action/RotateAbsolute]
> ros2 action info /turtle1/rotate_absolute
  Action: /turtle1/rotate_absolute
  Action clients: 1
      /teleop_turtle
  Action servers: 1
      /turtlesim
```

Welche Elemente des Turtlesim-Interfaces können Sie erklären? Wie gehen Sie vor, um sich bestimmter Schnittstellen zu vergewissern?

Welche Struktur hat das Action-Interface?

```
> ros2 interface show turtlesim/action/RotateAbsolute.action

  # The desired heading in radians
  float32 theta
  ---
  # The angular displacement in radians to the starting position
  float32 delta
  ---
  # The remaining rotation in radians
  float32 remaining
```

Die Definition des Ziels erfolgt mittels

```
ros2 action send_goal /turtle1/rotate_absolute turtlesim/action/RotateAbsolute
  {'theta: -1.57'} --feedback
```

Aufgabe der Woche

- Schreiben Sie einen neuen Service, der es erlaubt die Roboter abzufragen, die sich in einem Raum befinden.