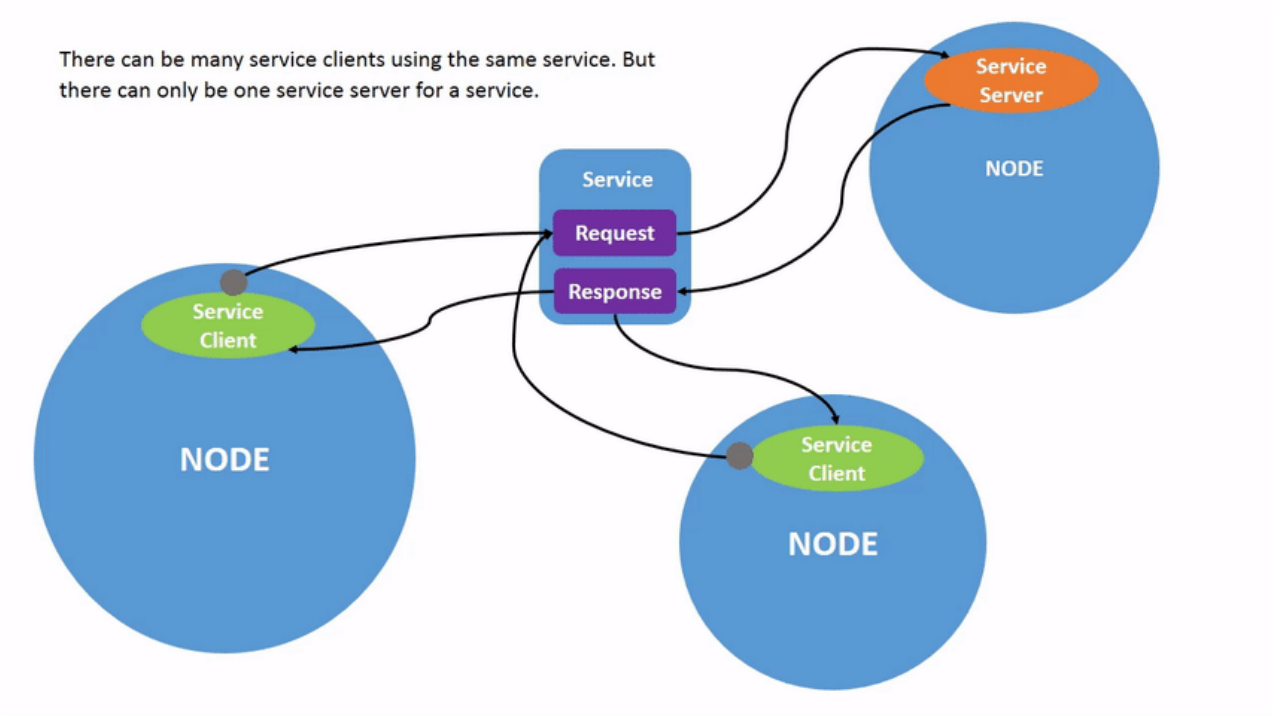
1. Theoretical background

Oficial documentation

- They are a type of call-response bidirectional data streaming

- Composed by 2 elements:
    - Server: Element that provides data when called by a client
    - Client(s): Element(s) that calls the server



3. Python services

3.1 Server

3.1.1 Structure

1. Structure a node
2. Import the service interfaces
3. In the **constructor**:
    - Instanciate a server defining: interface, service and callback
4. Create a callback with 2 parameters: A request and a response objects.
    - Get the request-object attributes.
    - Assign the response-object attributes.
        - They have the same name as in the **service file**
    - Return the response object
5. Main function:
    - Spin the server
6. Create an entry point and compile

3.1.2 Methods

- API
- Server instanciation

```
Node.create_servoce(<interface>, <srv_name>, <callback>)
```

3.1.3 Callbacks

- Parameters: attributes defined in the service file
    - request_object
    - response_object
- Contents:
    - Set the response parameters
- Returns:
    - response_object

```
def callback(self, request, response):
        response.<atrib> = request.<atribute> + request.
<atribute>
    return response
```

3.2 Client

1. Structure a node
2. Import the service interfaces
3. In the **constructor**:
    - Instanciate a client instance defining: interface and service
    - Wait until the service is available
    - Create a **request-object** as an instance of the interface
4. Create a **request method**:
    - Assign the request-object attributes
        - They have the same names as in the interface.
    - Receive the response as a **response-object** (future)
5. Main function:
    - Invoke the request method
    - **Spin once** the server
    - Check if the **response** is ready by the **future-object**
    - Create a **result-object** using the **future-object**
    - Get the **value of the result** as an attribute of the **result-object**.
        - It has the same names as in the interface.
    - *Note: If possible use exceptions handling to avoid errors.*
6. Create an entry point and compile

3.1.2 Methods

- API
- Client instanciation:

```
Node.create_client(<interface>, <srv_name>)
```

- Wait for the server:

```
while not self.cli.wait_for_service(timeout_sec = <seconds>):
    ...
```

- Instanciate a request object:

```
request_obj = <interface>.Request()
```

- Call the service: It returns a **future object**

```
future = Node.cli.call_async(<request_obj>)
```

- Read the result:
  - Check if the **future object** is ready
  - Instanciate a **result object**
  - Read the **result object** attributes

```
if <client>.future.done():
    result = my_client.future.result()
    <client>.get_logger().info(f'Result: {result.
<attribute>}')
```

3.1.3 Request method

- Parameters: Variables to fill the request attributes
- Contents:
  - Fill the request_object attributes
  - Call the service
- Returns: Empty

```
def request(self, <prm1>, <prm2>):
        self.request_obj.<atribute1> = <prm1>
        self.request_obj.<atribute2> = <prm2>

        self.future = self.cli.call_async(self.req)
```

4. C++

5. Remappings (CLI debugging)

- Allow to change the name of a service when it is launched

```
ros2 run pkg <node> -ros-args -r <old_name>:=<new_name>
```