

---

# **DDS-Demonstrators**

*Release TechPush*

**Jul 16, 2020**



---

## Contents

---

<b>1</b>	<b>Measure &amp; Blog</b>	<b>3</b>
1.1	Deliverables . . . . .	3
1.2	Main Table of Content . . . . .	4





**Work just started**

- All sources, including for documentation can be found in the official *main repository*: <https://bitbucket.org/HighTech-nl/dds-demonstrators>
- All development is done in *Feature forks*, using the [Forking Workflow](#)
- **ONLY** the *main repository* is used to generate the documentation: <https://DDS-Demonstrators.readthedocs.io/>

DDS<sup>1</sup> (Data Distribution Service) is a real-time, embedded “*data-centric*” **pub-sub** communication framework. Which is becoming popular in the (professional, industrial) IoT-world. It is also very *HighTech*; one of the reasons why we (Sogeti HighTech BL) are interested.

Sogeti-HighTech has bundled several innovation-projects into this **DDS-Demonstrators** program.

Each project will result in a simple *Demonstrator*, showing the power of DDS; and *how to use* DDS; including (demonstration) code, test, and lessons-learned<sup>2</sup>.

Most software is made in C, C++ or Python and will be run on an embedded Linux system. Often we use a Raspberry Pi, which is affordable; and a good reference for a modern embedded system.

- Python is mainly used as a Proof-of-Concept language; when the *execution* speed is less relevant than the development-velocity. Python is also used for scripting (for tests and measurements).
- When speed- or latency-measurements are needed, C/C++ is typically used; possibly after a Python PoC.
- Python is also often used for tool-like demonstrators (such as the *XScope*)

---

<sup>1</sup> See: [https://en.wikipedia.org/wiki/Data\\_Distribution\\_Service](https://en.wikipedia.org/wiki/Data_Distribution_Service)

<sup>2</sup> To share this knowledge many *DDS-Demonstrators* are treated as open-source. Then it becomes easy to share it with our (over 100) HighTech Software Developers, other Sogeti-professionals, and to show it to clients.



---

## Measure & Blog

---

The DDS-Demonstrators use the “Measure & Blog” approach: every project and every iteration starts with an (R&D) question, and ends with a publication on the answer.

Typical, the project included some “measurements”, and the publication is a simple “blog”.

The intermediate results, like code, automatic test- and measure-scripts, and other documentation are also *published*: should be stored in this version control system. In such a way that others can repeat the experiment easily and learn quickly.

Often, each “Measure & Blog” will lead to new questions. They should also be documented and can lead to new iterations.

More on this, later in a generic blog ...

### 1.1 Deliverables

Every iteration<sup>3</sup> sprint gives the following deliveries:

- 1) The *production code* (read: a DDS-Demonstrator, in functional, partial deliveries).
- 2) The design, where relevant: Use plantUML diagrams, text, etc.
- 3) The test- and measurement-code.
- 4) User manuals (SUM: Software User Manuals) - of the product (how to install, run, use ...).
- 5) Project / Sprint documentation (including estimates, comments).
- 6) The “Blog”: from research question by the measurements/results towards the conclusions/advice; typically short.
- 7) All kind of “convenient notes”; in the *Team pages*
- 8) “Free” documentation improvements on existing (used) documentation –*without adding cost*.

---

<sup>3</sup> The term iteration is used to generalize a (scrum) **sprint**. The same deliverables are expected at the end of a *super-sprint* (in safe: “PI”), an internship, an innovation-assignment and the end of a project.

- 9) All other short, cheap improvements to record knowledge.

## 1.2 Main Table of Content

### 1.2.1 DDS, an intro

DDS is a real-time, embedded “*data-centric*” **pub-sub** communication framework. It has a long history and is very well described. Although many documentation shows the area when it is described (a bit old-school). However, it is very alive!

A big benefits of DDS: It is truly distributed

- It has no central server
- This makes it robust;
- There is no single-point-failure

A drawback of DDS: it has a steep learning curve

- It has many, advances features
- It is simple to use; after one grasp the concepts
- These DDS-Demonstrators will help to understand & use DDS

The “DDS-stack” is independent of the platform (hardware & OS), the used programming language, and the supplier! Unlike many other protocol, both the “wire-protocol” and the “API” are standardized. This should make it possible to “plug and play” several implementations - at least on paper. Part of this program is to play and experiment with those promises; as well as with other features<sup>1</sup>.

DDS is implemented by several suppliers and comes in multiple packages. There are also some ‘open’ or ‘community’ editions. Given the goal of the DDS-Demonstrators, we will restrict ourselves to open-source implementations.(e.g. [CycloneDDS](#) and [eProxima Fast RTPS](#)).

We focus on *CycloneDDS*, an [Eclipse-Foundation](#) project<sup>2</sup>. And will add *eProxima Fast RTPS*, as we expect it may be smaller, more suitable for really small embedded systems. Again, we have to experiment with that<sup>3</sup>.

### More info

- Wikipedia: [https://en.wikipedia.org/wiki/Data\\_Distribution\\_Service](https://en.wikipedia.org/wiki/Data_Distribution_Service)
- OMG (*the* standards) <https://www.omg.org/omg-dds-portal/>
- DDS-Foundation: <https://www.dds-foundation.org>

### 1.2.2 Installation

Message-passing systems use either distributed or local objects. With distributed objects the sender and receiver may be on different computers, running different operating systems, using different programming languages, etc. In this case the bus layer takes care of details about converting data from one system to another, sending and receiving data across the network, etc. The following types of protocols will be handled

---

<sup>1</sup> We already have some indications this is different in practice. By example: a least one (working) python-binding is only able to talks to nodes with the same python-binding. More on that later.

<sup>2</sup> Which is a quite trivial selection method. We also like it has Dutch roots (like DDS itself).

<sup>3</sup> We also like *eProxima Fast RTPS* as it be used in ROS (Robot Operating System); which we also use.

## DDS: Data Distribution Service

DDS is a standard, realtime, platform & language agnostic, distributed communication-system; based on pub-sub. It was developed by/for Thales radar systems; then called “splice”. Later become a standard, with several vendors, both commercially and opensource. It is widely used: in finance, technical and scientific environments. It is also very suited for and popular in the IoT world. Also see: [https://en.wikipedia.org/wiki/Data\\_Distribution\\_Service](https://en.wikipedia.org/wiki/Data_Distribution_Service)

Recently, the Eclipse foundation has approved an proposal to have a (true) open-source implementation, called **Cyclone DDS**. It is based on a Community Edition of a commercial vendor.

- All sources are on [github](#).
- There is also a [python-binding](#) (also on github).
- Java and other language bindings are probably available, or upcoming.

## Cyclone DDS (cdds)

This implementation is used for the IPSC experiments (other are tried, but not (yet) working).

- It does work on my Mac, and the RaspberryPi (Model 2, “stretch” (9.1 Debian)
- All code is natively compiled, both on Mac and RaspberryPi (no cross-compiler needed!)
- Documentation is available, but limited. And often not updated to the Cyclone version.

---

**Note:** Don’t mix it up!

CycloneDDS is based on an earlier “community edition” of ADLINK; which has also put other versions on github; mostly in there “[atolab](#)” account. Although related; they are not the same!

- In some cases, there docs (and code) is *almost* the same. Or point to each other.
  - This is confusing; be warned!
  - An earlier version of this document even pointed to the wrong repro :-)
  - The python-binding *python-cdds* is even part of the atolab account.
  - That account has another python-binding (pydds) too; that is *NOT* working/compatible
- 

## Build & Install

### Base (C/C++)

Just clone the sources, run cmake and install; as described on <https://github.com/eclipse/cyclonedds>

Before one can compile it, one should install cmake (note make!), Marvin and Java-8 (or newer)

On Linux/raspberrypi:

```
> sudo apt-get install cmake
> sudo apt-get install oracle-java8-jdk
> sudo apt-get install maven
```

Check if the right java version is used. It should say something like:

```
> java -version
java version "1.8.0_65"
```

if not, then set the proper version with:

```
> sudo update-alternatives --config java
```

### Python bindings

Again, clone, and build; See: <https://github.com/atolab/python-cdds>

One needs (python3) jsonpickle (note the typo on the page: the C is missing). And cdds (see above):

```
> sudo apt-get install python3
> sudo pip install jsonpickle
```

### Links

- <https://projects.eclipse.org/proposals/eclipse-cyclone-dds>
- <https://github.com/eclipse/cyclonedds>
- <https://github.com/atolab/python-cdds>

## MQTT: Message Queuing Telemetry Transport

MQTT is an ISO standard publish-subscribe-based messaging protocol. MQTT works on top of TCP/IP and is widely used due to it being lightweight and simple in use. MQTT is especially popular in IoT applications with low bandwidth network connections. Its principle is very simple, there is a central server called the message broker. A publisher sends a message to the broker on a specific topic. The broker then sends this message to every device or process that subscribed to this topic.

### Install

#### Broker

There are two parts to installing MQTT, there is the broker and the client. Mosquitto is one such broker developed by the Eclipse Foundation and is easily installed under Debian Linux with:

```
sudo apt-get install mosquitto
```

For debugging purposes you might also want to install the command line tool:

```
sudo apt-get install mosquitto-clients
```

### Python

Python uses the paho-mqtt library and can be installed with the following:

```
pip install paho-mqtt
```

## ZMQ, OMQ: ZeroMQ

ZeroMQ, ZMQ or OMQ is a asynchronous messaging library, aimed at use in distributed or concurrent applications. The main strength of ZeroMQ comes from the capability of sending and receiving messages in many different ways. For example you can have messages between applications send based on a pub-sub system, request-reply or push-pull system. Within these systems it is then possible to create queues, proxies, pipelines, messages brokers and a whole lot of other components. This makes it possible to create many different ways of getting a message from sender to receiver depending on the need of the system.

## Notes

The original author “Pieter Hintjens” is *not* a Dutch-man; he was from Zuid-Africa. He is also passed away ...

## Install

### Python

Installing ZMQ for python on linux/raspberry pi can be easily done with:

```
sudo apt-get install libzmq3 libzmq3-dev python3-zmq
```

## Links

- <http://zeromq.org>

## 1.2.3 Backlog & requirements

Here you find a *backlog* of the DDS-Demonstrators; it will be updated/maintained by the *Product Owner(s)*.

---

**Tip:** When you –as part of a project– feel the need to add a Demonstrator-idea to the backlog, plz make a proposal in your *Team pages*!

When accepted it will be moved to here.

**Warning:** When using *needs*, tag it with ‘idea’!

---

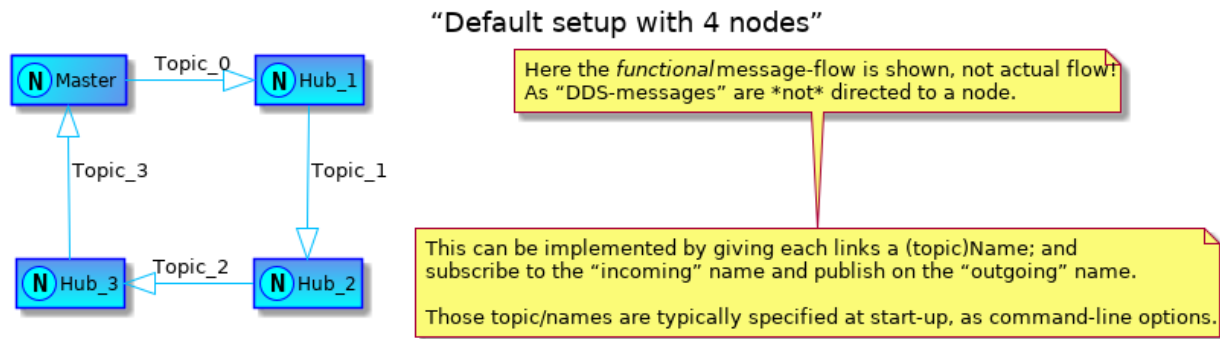
## RoundTrip & Flood

**RoundTrip** and **Flood** are two basic *Hello World* style Demonstrators.

A simple message is send around toward several (network) nodes. Typically, each *node* runs on a dedicated Raspberry-Pi. As DDS is network-agnostic; multiple nodes can run on the same computer. By example, on a PC/Laptop, which is great for demonstration and an easy start.

A standard setup is shown below, where 4 nodes are used; a trivial variations is to have more or less nodes

One node is called “master”, as it starts the roundtrip; and executes the measurements. The other nodes are called “hub” and have the same functionality; the “links” are set by configuration.



The difference between ‘RoundTrip’ and ‘Flood’ is how *fast* and in which *order* the message are processed.

Demonstrator: **RoundTrip** *RT\_001*

status: import

tags: RT

links incoming: *RT\_PY\_01c, RT\_Cpp\_01c, RT\_C\_01c*

In RoundTrip only **one** message is send around at the same time. The ‘Master’ starts a the sequence by sending one message (*by example: “0”*). And the receiving hub forwards it; often trivially changing it (*in the example: by adding “1” to the incoming message*).

Only when the ‘master’ receives the message (*let’s say “3”*), it send a new message. In the example, that would be “4” (3+1). And the sequence starts again.

After an configured number of roundtrips, the master stops sending/forwarding; and the demo is over.

Demonstrator: **Flood** *F\_001*

status: import

tags: F

links incoming: *F\_PY\_01c, F\_Cpp\_01c, F\_C\_01c*

In Flood the ‘hub’ works exactly as in [RoundTrip \(RT\\_001\)](#): it receives a message and forwards it. Typically, the message is changed by adding a hub-specific postfix (*by example the hub-number*)

The ‘master’ differs however: it does **NOT** wait for an incoming message to send the next message. Instead it sends all message as fast as possible.

Still, the ‘Master’ has to read the incoming message; but can dispose them; reading **should** have a higher priority the sending!



## Hints

## Todo

**Todo:** SetUp the M&B comparison between DDS and other protocols in this environment

**Tip:** Is this part of “DDS-Demonstrators”, or another HighTech-NL TechPush project?

## More links

- About the setup with some raspberry’s see [setup](#)
- Some roundtrip/flood see [RoundTrip](#)

## Variants

Requirement: <b>Python RoundTrip</b> <a href="#">RT_PY_01c</a>
<p>status: import</p> <p>links outgoing: <a href="#">RT_001</a></p>
Implemented the <a href="#">RoundTrip (RT_001)</a> Demonstrator in python; using the CycloneDDS stack and the “ <i>asis</i> ” “ <i>ato-lab</i> ” python binding.
Requirement: <b>C++ RoundTrip</b> <a href="#">RT_Cpp_01c</a>
<p>status: import</p> <p>links outgoing: <a href="#">RT_001</a></p>
Implemented the <a href="#">RoundTrip (RT_001)</a> Demonstrator in C++; using the CycloneDDS stack.

Requirement: <b>C RoundTrip</b> <i>RT_C_01c</i>
status: import  links outgoing: <i>RT_001</i>
Implemented the <a href="#">RoundTrip (RT_001)</a> Demonstrator in plan-old-C; using the CycloneDDS stack.

Requirement: <b>Python Flood</b> <i>F_PY_01c</i>
status: import  links outgoing: <i>F_001</i>
Implemented the <a href="#">Flood (F_001)</a> Demonstrator in python; using the CycloneDDS stack and the “asis” “atolab” python binding.

Requirement: <b>C++ Flood</b> <i>F_Cpp_01c</i>
status: import  links outgoing: <i>F_001</i>
Implemented the <a href="#">Flood (F_001)</a> Demonstrator in C++; using the CycloneDDS stack.

Requirement: <b>C Flood</b> <i>F_C_01c</i>
status: import  links outgoing: <i>F_001</i>
Implemented the <a href="#">Flood (F_001)</a> Demonstrator in plan-old-C; using the CycloneDDS stack.

## Notes

The “atolab” python-binding do work for the [RoundTrip \(RT\\_001\)](#) and [Flood \(F\\_001\)](#) Demonstrators; but there are some know issues. It looks like this library is kind of *PoC*:

- Python-objects are (typically) serialised into json and transfers as strings. This works for python-to-python, but not with other languages. It also not according to the “wire specifications”
- This Python-API does not expose the full (C/Conceptual) API. More advantage DDS-features are impossible.
- The Python interface is not very pythonic
- Note: There is not official pythonic-API.

For the basic-Demonstrator this is not an issue. For the long run, a better, completer and more pythonic-API would be welcome. Then the new (official) Java- and C++ style interfaces are a better start. Its is future work.

## Links

- CycloneDDS sources are on github: <https://github.com/eclipse/cyclonedds>
- The atolab-python binding <https://github.com/atolab/python-cdds>

## XScope

The **XScope** Demonstrator is kind of an “add-On” tool, to debug other *Demonstrators*. Several variant of the XScope are foreseen: a basic version, and possible some “subclasses” to tailor for specific Demonstrators.

Possible a generic one will become available once.

Demonstrator: <b>XScope (base)</b> <i>XS_001</i>
status: import tags: XS
<p>The (base) XScope is like an “oscilloscope”, for DDS-messages. It a passive tool, and only listening to topics. It will frequently “collect” samples of the (configured, list of) Topics and store/print/show them.</p> <p>When the value of (a part of) a Topic in numeric, it should be able to draw them on a time-scale. A very-basic version this draw-feature is allowed to do “off-line”, by example via an Excel (export to csv and draw later in Excel).</p> <p>By design, this XScope-tool should extendable. The basic functionality should be in the base-XScope; an optional, flexible adaptation-layer (towards a future DDS-Demonstrator) can be used later – by example to “find” the above mentioned numeric value in a Topic</p> <p>The “List of Topics” to monitor is preferable configurable; both in (eg. an) ini-file and on the commandline (without referring to a file).</p>

## Matrix Board Communication

This DDS-Demonstrator is inspired by the communication of the well know “Matrix Boards” above the (Dutch) Highways.

Each MatrixBoard shows the maximal allowed speed; during traffic-jams. It needs the input from the next two boards; whose sensors measure the speed at that location.

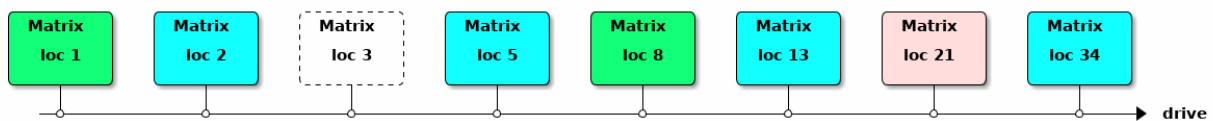
The real MatrixBoards have sensors (*to measure the number of cars, there speed, etc*), an actuators (*showing the speed*), and some simple algorithms (like: “*the step is speed between 2 boards is small*” and “*when the speed changes, the light should blink for some time*”). The communication between board is are hard-wired; by using land-lans.

Demonstrator: <b>MatrixBoard</b> <i>MBC_001</i>
<pre>status: import tags: MBC</pre>
<p>The MatrixBoard DDS-Demonstrator shows some dynamic aspects of DDS in a similar setup. It uses DDS for all communication.</p> <p>It does not implement the real MatrixBoards: it has no sensors, no actuators, nor the exact algorithms. The former two are (very simple) simulated; and the algorithms are replaces by simple rules (roughly doing the same).</p> <p>When running a set of MatrixBoards, each MatrixBoard is started with a location (thinks about a “hectometer sign”).</p> <p>It should automatically “find” the next two ‘upstream’ MatrixBoards and <i>subscribe</i> to those; only.</p> <p>Whenever the set of MatrixBoards changes, a MatrixBoard should automatically discover the new setup and act accordingly (cancel and add subscriptions)</p>

## Example

Given a setup like below, where the MatrixBoard at `loc=3` does not yet exist; and `loc=21` does.

- MatrixBoard at `loc=1` should subscribe to `loc=2` and `loc=5`.
- MatrixBoard at `loc=8` should subscribe to `loc=13` and `loc=21`.



Now suppose, MatrixBoard `loc=3` is added and/or `loc=21` is abruptly killed. Then:

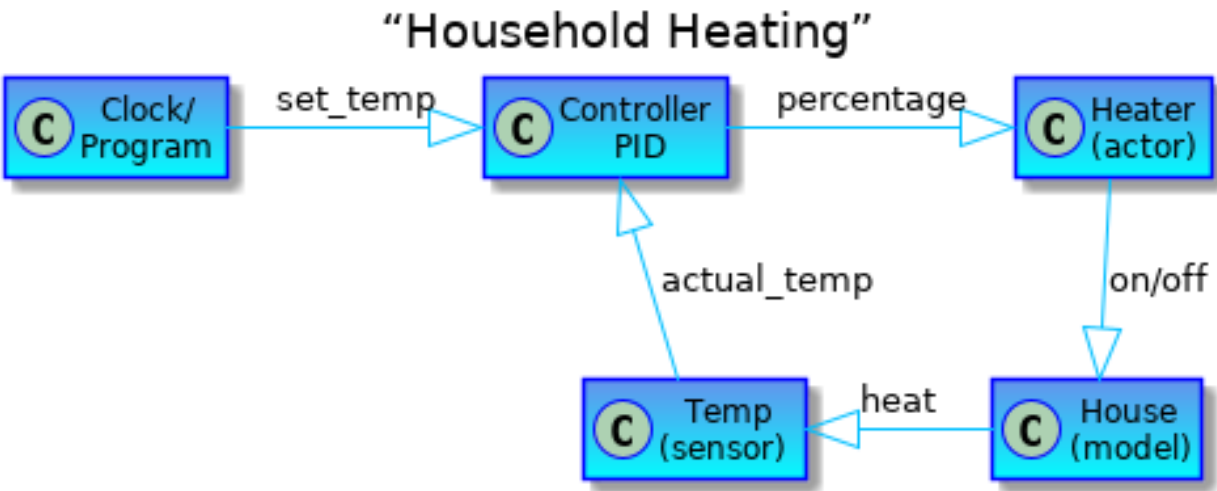
- MatrixBoard at `loc=1` will use `loc=2` and `loc=3` (and **not** listen to `loc=5` anymore).
- MatrixBoard at `loc=8` will add `loc=34` to is subscription-list.

Household Heating Demonstrator

**Warning:** On Hold

This Demonstrator is on-hold for now; it has a technical complexity that was initially not foreseen. And was not part of the project.

As this control-loop has a feedback loop, a simple “event driven” architecture can/will result in (indirect) self-triggering of all nodes. That is not indented! This is easy to solve using “controll-loop concepts”; but that is not the goal of this Demonstrator.



Demonstrator: **Household Heating** *HH\_001*

status: on-hold  
tags: HH  
style: needs\_red

To Be Done

Lists of Requirements

## Open Requirements

Title	ID	Status	Outgoing	Incoming
C Flood	<i>F_C_01c</i>	import	<i>F_001</i>	
C++ Flood	<i>F_Cpp_01c</i>	import	<i>F_001</i>	
Python Flood	<i>F_PY_01c</i>	import	<i>F_001</i>	
C RoundTrip	<i>RT_C_01c</i>	import	<i>RT_001</i>	
C++ RoundTrip	<i>RT_Cpp_01c</i>	import	<i>RT_001</i>	
Python RoundTrip	<i>RT_PY_01c</i>	import	<i>RT_001</i>	

---

**Note:** ‘import’ counts as ‘open’

---

## In progress

No needs passed the filters

Title	ID	Status	Outgoing	Incoming
-------	----	--------	----------	----------

## New Idea’s

No needs passed the filters

Title	ID	Status	Outgoing	Incoming
-------	----	--------	----------	----------

## 1.2.4 Demonstrators

Each Demonstrator will be documented in it own directory; by the developer(s). See below for the list of DDS-Demonstrator (in various stages).

Title	ID	Status	Outgoing	Incoming	Tags
Flood	<i>F_001</i>	import		<i>F_PY_01c; F_Cpp_01c; F_C_01c</i>	F
Household Heating	<i>HH_001</i>	on-hold			HH
MatrixBoard	<i>MBC_001</i>	import			MBC
RoundTrip	<i>RT_001</i>	import		<i>RT_PY_01c; RT_Cpp_01c; RT_C_01c</i>	RT
XScope (base)	<i>XS_001</i>	import			XS

---

**Note:** for developers

Please create your own directory (within your Feature-branch/fork) with a name as set by the product-owner. Both in the `src` and `docs/doc/Demonstrators` directory. Use the doc-directory to document the Demonstrator.

See the [Team pages](#) directory for ‘œscratch’ documentation on personal/team-level. Also, create a subdir there.

---

The (toplevel) requirements and backlog of Demonstrator can be found: [Backlog & requirements](#)

## PingPong

### Description

Ping Pong is a way to test connectivity between two computers or programs. It works by sending a message from one entity to another. This other entity will reply on the first message and so on. The figure below visualises this concept.

.. uml:

```
@startuml
ping -right->> pong : message
pong -left->> ping : message
@enduml
```

### Implementation

There are different implementations of the PingPong. A short description is made for the execution of a particular implementation. This description contains information about how the implementation can be compiled and executed.

This list contains the different implementations:

### C++ PingPong

**authors** Sam Laan

**date** March 2020

### Description

This is the documentation for the C++ PingPong solution using the Cyclone DDS library. It is assumed you have already installed Cyclone DDS. If you haven't please consult the setup guide of team Hurricane([Setup Guide for CycloneDDS](#)).

### How to run

To run the program navigate to the /src/demonstrators/PingPong/C++ folder. The source code and CMakeLists of the program are in this folder. Now to build the solution perform the following commands:

```
mkdir build
cd build
cmake ..
make -j4
```

An executable called "PingPong" should have appeared in the build folder. It is now possible to run the program using the following command:

```
./PingPong
```

The program will ask if you want to run as ping or as pong. Type 1 and press enter for ping, type 2 and press enter for pong. In ping mode, the program will subscribe to a topic which is one higher than its id. For example, if you choose id 1 it subscribes to topic 2. After choosing ping or pong, the program will ask for an id. This is the previously mentioned one. Pong should always be started first because it waits for a ping to come in. An example scenario of when the program will start ping-ponging is as follows:

- Start pong with id 2
- Start ping with id 1

For more information about the program please refer to the source code. It is heavily commented and should provide all the information you need.

### PingPong

- First start Ping, then Pong; in 2 terminals
- You have about 6 seconds between – the SLEEP is hardcoded
- Stop (both) by killing (^C) them
- Ping and Pong may run on different systems (or on the same).
- Same network segment

or go to RoundTrip folder and look at the readme there for setting up a roundtrip

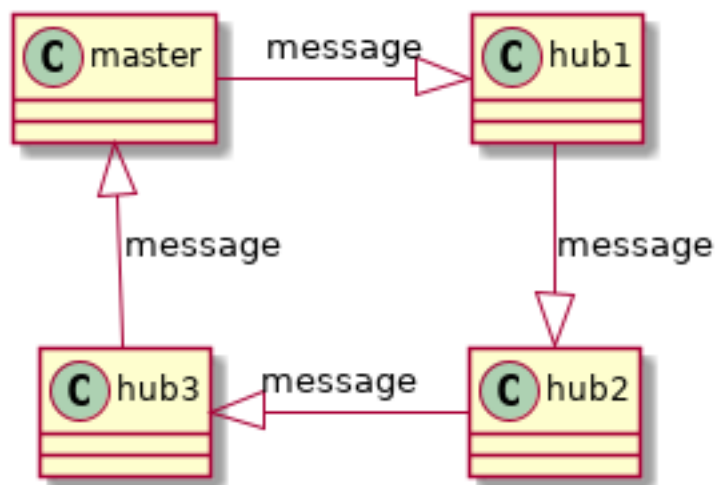
### RoundTrip

#### Description

To test the different ways of communication, what is called a roundtrip is used. In a round trip, different processes or systems send messages to one another. There can only be one master, but there can be an infinite amount of hubs. The master and hubs together form a circle, where one message loops through all the devices.

The round trip is used for measuring the time needed to send and receive a message on a network. The time between each device can easily be measured.

A roundtrip with one master and three hubs looks like this:



#### Implementation

There are different implementations of the round trip. A short description is made for the execution of a particular implementation. This description contains information about how the implementation can be compiled and executed.



This list contains the different implementations:

## Round trip and Flood in C

**authors** Stefanos Florescu

**date** March 2020

The Roundtrip demonstrator has two different experiments. The first experiment sets up one main master hub and three hubs. The master hub sends the value 1 and starts measuring time upon sending the message. This message travels from hub to hub with each hub adding 1 until it returns to the master. Upon arrival of the modified integer from the third hub at the master, the clock is stopped and the time it took for the roundtrip is displayed.

Similarly to the simple roundtrip experiment, the flood experiment has the same setup of master and three hubs. The only difference is that the master now sends a bunch of messages at the same time. The hubs read the messages one by one, process them (by adding 1) and pass them on. The master will start measuring time from the moment he sends his first message and up to the moment he receives the last message from the last hub.

More information on these demonstrators can be found in the Backlog.

## Building and running the executables

Make sure your local repo is up to date with the [HighTech-nl repo](#) and move into the directory / dds-demonstrators/src/demonstrators/RoundTrip/C:

```
$ mkdir build
$ cd build
$ cmake ..
$ make
```

Also, after the build directory has been initialized using the `cmake ..` command, the two scripts `run_roundtrip.sh` and `run_flood.sh` can be used to compile the executables and run the experiments. These two scripts are similar with the only differences being that they call different executables. For example the `run_roundtrip.sh` includes the following:

```
#!/bin/bash

cmake --build $PWD/build/
echo "Deleting old log file."
rm $PWD/roundtrip.log

echo "Running Roundtrip Demonstrator...press ^C to exit"
./build/Hub_roundtrip topic_2 topic_3 &
./build/Hub_roundtrip topic_3 topic_4 &
./build/Hub_roundtrip topic_4 topic_1 &
./build/Master_roundtrip topic_1 topic_2 > roundtrip.log
```

The master will output all his measurements into the `.log` file in each respective case. Furthermore, the first argument passed to each Hub/Master executable is the topic name on which that entity will be listening on, while the second argument is the topic on which that entity will write to. In order to see the output of the experiment simply open the respective `.log` file.

---

**Note:** The Master/Hub can be executed on different nodes, such as Raspberry Pis and PCs. See [Setting up Raspberry Pi network](#) for details on how to connect to the Raspberry Pis.

---

**Note:** In the case of the flood experiment, the number of samples the Master will send can be defined in the `Master_flood.c` file by changing the `#define FLOOD_SAMPLES` number.

---

### Running the Round trip in a Docker Container

Similarly to what we described above, the Round trip and Flood experiments can be executed from a Docker Container. First make sure you follow the guide on how to build the Docker Image which can be found in the [Docker Guide: How to create and run a Docker Image](#) under “Building Demo Image”. First create a new container using:

```
$ docker run -it --network="host" --name <container name> <image name>
```

You can open another terminal/Powershell and have as many processes on that container as you want using per terminal:

```
$ docker exec -it <existing container name> bash
```

Navigate into the `Roundtrip/` directory and either experiment using:

```
$ ./docker_run_roundtrip.sh
$ ./docker_run_flood.sh
```

**Warning:** Due to the networking issue of the Docker Desktop (see in [Docker Guide: How to create and run a Docker Image](#) under “Running Docker Containers” for more info) on Windows/MacOS we need to change the DDS networking interface in the `docker_cyclonedds_config.xml` in `/root/`. The video in [Matrix Board Communication in C](#) shows how this is done. On a Linux machine this is not required and your container will communicate over the network.

### Round trip in C++

**authors** Joost Baars

**date** Mar 2020

### Description

The round trip C++ directory can be found in `src/demonstrators/RoundTrip/C++/`. This directory contains a folder for the round trip implementation as well as the flood implementation.

This page is split into two major chapters. One for the round trip application and one for the flood application.

### Dependencies

The dependencies necessary for succesful compilation of the round trip / flood application in C++ are the following:

- CMake
- C++17 compiler (Gcc 7+ for example)
- CycloneDDS library

## Round trip

This chapter contains the information about the round trip application. This application can be found in the directory: `src/demonstrators/RoundTrip/C++/RoundTrip`.

## Configuring

First go to the RoundTrip C++ directory (see [Round trip](#)).

The C++ round trip implementation contains an implementation for the round trip with the use of callbacks or the use of polling and reading. One of these implementations can be chosen before building the project.

The implementation can be configured by altering the following line in the `CMakeLists.txt`:

```
set(PROJECT roundtrip_read)
```

This line can be found at the top of the `CMakeLists` file. The `roundtrip_read` can be changed to `roundtrip_callback` for the callback implementation.

## Building

First go to the round trip C++ directory (see [Round trip](#)).

Execute the following commands:

```
mkdir build && cd build
cmake ..
make
```

These commands compile the code and create the executable.

## Execution

The application must be executed using various parameters. The application can be executed using the following command:

```
./RoundTrip <device ID> <number of devices> <total round trips>
```

The following example starts 4 nodes for the round trip. The slaves of the round trip are started in the background. Only the master is started in the foreground in this example. Each device pings a total of 1000 times. Therefore, there are 1000 round trips.

```
./RoundTrip 2 4 1000 & ./RoundTrip 3 4 1000 & ./RoundTrip 4 4 1000 &
./RoundTrip 1 4 1000
```

A more thorough description of the parameters can be found when executing the application with no parameters.

---

**Note:** Execution of the program

The round trip is initiated by the device with `<device ID> = 1`. Therefore, `<device ID> = 1` should always be started after the other ID's are started.

The devices added with <device ID> must be an increment of the previous one. This function does not dynamically search for the next device! So if there are three devices, device 2 and 3 should be started first. Afterwards, device 1 can be started.

The <total round trips> parameter should be the same for each application.

---

### Implementation

Each round trip application creates a participant containing a reader and a writer. The writer writes to the topic of the ID above it. So applications with ID = 1 writes to the application with ID = 2. The reader reads from its own topic. These topics have the name “roundtrip” with the ID behind it. So the topic with ID = 1 is “roundtrip1”.

The application with ID = 1 initiates the round trip. Therefore, it starts with writing to topic “roundtrip2”. The application with ID = 2 then receives the message of the application with ID = 1, and sends a message to the next application.

### Read

The read implementation continuously executes the `dds_take()` function to get the latest message of its own topic. `dds_take()` is used so the message is directly removed from DDS and so it won't be read again. When a message is received, the application uses `dds_write()` for writing to the next application.

### Callback

The callback implementation uses a callback function for receiving the latest message and writing to the next application. This implementation sleeps until data is received on the readers topic. When data is received, a callback function is executed where `dds_write()` writes to the next application in the round trip.

### Flood

This chapter contains the information about the flood application. This application can be found in the directory: `src/demonstrators/RoundTrip/C++/Flood`

### Building

First go to the round trip C++ directory (see *Flood*).

Execute the following commands:

```
mkdir build && cd build
cmake ..
make
```

These commands compile the code and create the executable.

### Execution

The application must be executed using various parameters. The application can be executed using the following command:

```
./Flood <device ID> <number of devices> <total messages>
```

The following example starts 4 nodes for the flood application. The slaves of the flood are started in the background. Only the master is started in the foreground in this example. There are a total of 1000 messages send by the master. For a flood to be finished, these messages should all be correctly received by the master.

```
./Flood 2 4 1000 & ./Flood 3 4 1000 & ./Flood 4 4 1000 &  
./Flood 1 4 1000
```

A more thorough description of the parameters can be found when executing the application with no parameters.

---

**Note:** Execution of the program

The flood is initiated by the device with <device ID> = 1. Therefore, <device ID> = 1 should always be started after the other ID's are started.

The devices added with <device ID> must be an increment of the previous one. This function does not dynamically search for the next device! So if there are three devices, device 2 and 3 should be started first. Afterwards, device 1 can be started.

The <total messages> parameter should be the same for each application.

---

## Implementation

Each flood application creates a participant containing a reader and a writer. The writer writes to the topic of the ID above it. So applications with ID = 1 writes to the application with ID = 2. The reader reads from its own topic. These topics have the name “flood” with the ID behind it. So the topic with ID = 1 is “flood1”.

The application with ID = 1 initiates the flood. Therefore, it starts with writing to topic “flood2” as fast as possible. The application with ID = 2 then receives the messages of the application with ID = 1, and sends a message to the next application for each message received.

The slaves (ID not equal to 1) just send a message to the other device in the flood loop as soon as they receive a message. The master (ID = 1) sends messages as fast as possible to the next device in the flood loop.

If the master sends a message, a value is incremented. The master keeps sending messages until the value is equal to the <total messages> parameter that the user inserted when executing the application. If the master receives a message, a value is incremented. At the end, this value should be the same as the <total messages> parameter that the user inserted when executing the application.

## Slave

The slave implementation continuously executes the `dds_take()` function to get the latest message of its own topic. `dds_take()` is used so the message is directly removed from DDS and so it won't be read again.

When a message is received, the application uses `dds_write()` for writing to the next application.

## Master

The master implementation continuously executes the `dds_take()` function to get the latest message of its own topic. Afterwards, it writes a new message to the next device in the flood loop.

Therefore, at the beginning, the master only writes to the next devices. Afterwards, the loop will eventually be in a stable condition where the master reads a message first and writes a new message afterwards. In the end, the master won't send messages anymore, and will only read the last messages that exist in the flood loop.

### Round trip in Python

#### Description

The round trip python directory can be found in `src/demonstrators/RoundTrip/Python/`. This directory contains the source for the round trip and also the flood implementation.

The `DDShub.py` needs to be setup with a publishing and subscribing topic, acting as a hub. The `RoundTrip.py` and `Flood.py` act as the master and do not have to be setup. Here the publishing and subscribing topic are hard coded. So for one hub the subscribing topic and another hub the publishing topic are also fixed, which 'connect' to the master.

To setup a round trip of a master and three hubs the following needs to be configured.

```
./DDShub.py Master_Out <publish name hub1>
./DDShub.py <publish name hub1> <publish name hub2>
./DDShub.py <publish name hub2> Master_in
./RoundTrip.py
```

An example of this, for the hubs, is present in bash script `run_L3.sh`. To setup a flood the `RoundTrip.py` needs to be replaced with `Flood.py`.

#### Execution

Use `DDShub.py` to setup either a flood or round trip setup (see `run_L3.sh` for an example). To run the flood example:

```
./run_L3.sh
./Flood.py <number of messages to send>

for <number of messages to send> a value can be specified. If omitted, a
↳value of 1000 will be used.
```

To run the round trip example:

```
./run_L3.sh
./RoundTrip.py
```

The above will only do one run. For more accurate measurements multiple runs have to be made. This can be achieved by using a loop (script `run_RoundTrip_10times.sh`). This script also logs each output of a single run into a separate log file.

```
#!/bin/bash
i=1
while [ $i -le 10 ]
do
    ./RoundTrip.py > ~/RoundTrip_python_${i}.log
    let "i = i + 1"
done
```

A similar script can also be created for executing flood 10 times, with the number of messages as argument.

```
#!/bin/bash
i=1
while [ $i -le 10 ]
do
    ./Flood.py $i > ~/Flood_python_$i.log
    let "i = i + 1"
done
```

## Measurements

For the measurements a setup has been chosen of four hubs of which one is the master doing all the measurements.

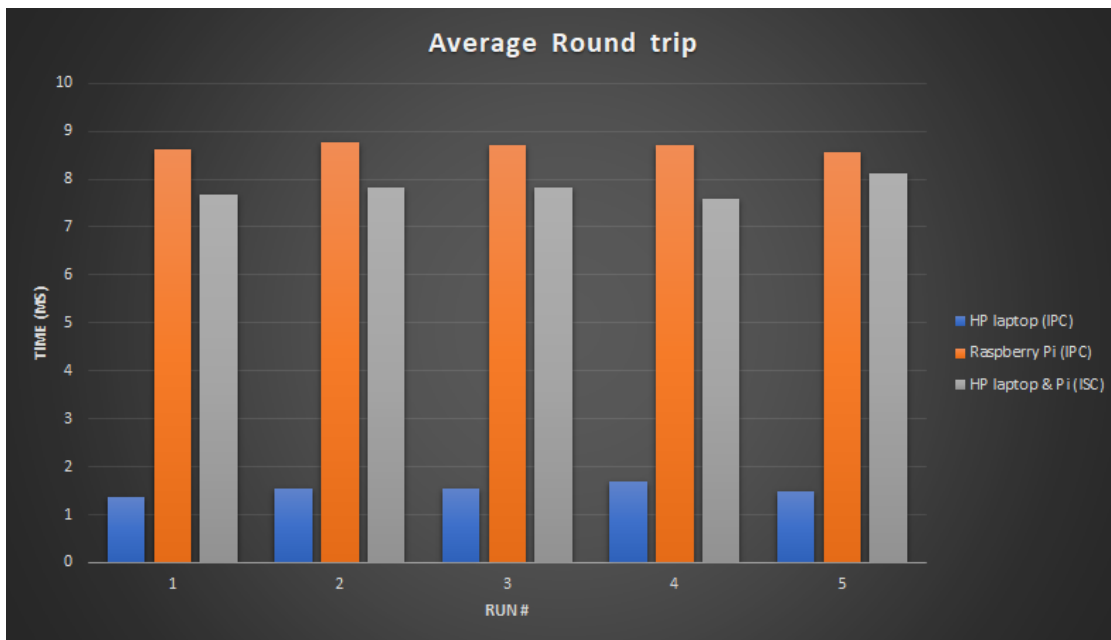
The measurements for the inter process communication (IPC) of the roundtrip and flood experiments have been gathered on a hp laptop, issued by Capgemini, and a Raspberry Pi, issued by the product owner. All the hubs run on the same hardware, either the laptop or the Raspberry Pi.

The measurements for the inter system communication (ISC) of the round trip and flood experiments have been gathered on both above mentioned hardware, for the IPC, and additionally an ethernet router. Two hubs run on the laptop, the master and hub 2, and two hubs run on the Raspberry Pi, hub 1 and hub 3. Since the communication will start at the master to hub 1, from hub 1 to hub2, from hub 2 to hub 3 and from hub 3 to the master, there is no inter process communication on either the laptop or the Raspberry Pi. There is only inter system communication.

Specifications of the hardware: HP laptop: HP Elitebook 850GS, Intel i7-8650 CPU @ 1,9 GHz 2,11 GHz Raspberry Pi: Raspberry Pi 3 Model B v1.2, Broadcom BCM2837 CPU @ 1,2 GHz Ethernet router: Netgear WGR614 v7

For the round trip several runs have been performed, with the averages as the following result:

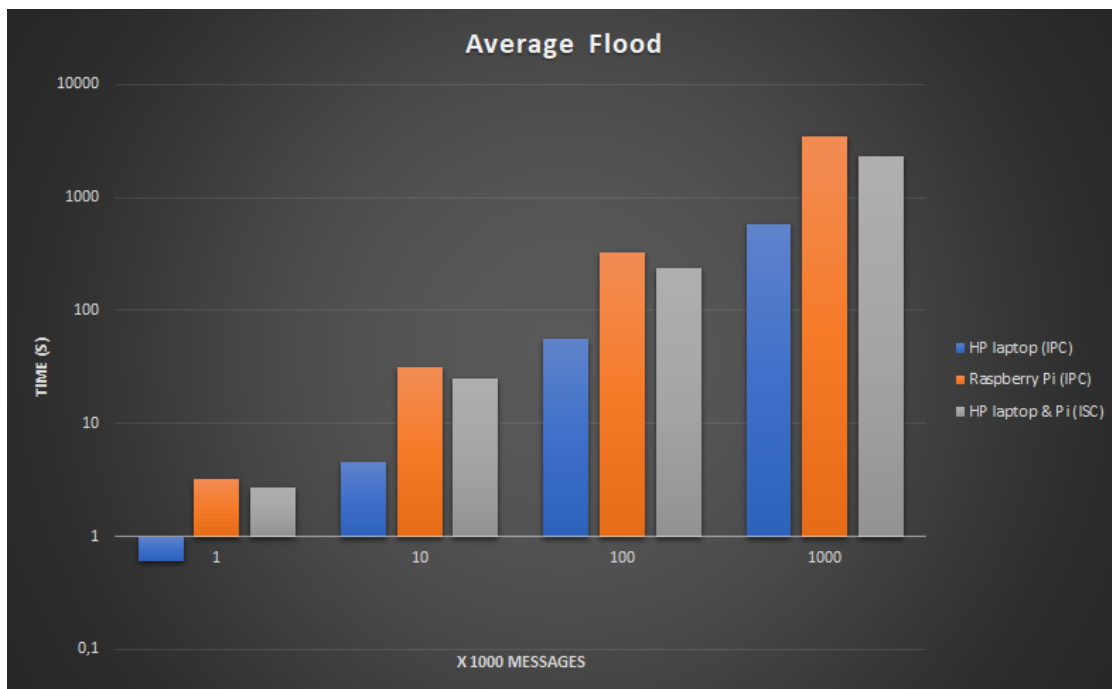
	Round trip (ms)				
	run 1	run 2	run 3	run 4	run 5
HP laptop (IPC)	1,357	1,549	1,534	1,689	1,484
Raspberry Pi (IPC)	8,623	8,775	8,705	8,718	8,574
laptop & Pi (ISC)	7,691	7,822	7,825	7,599	8,126



For the ISC measurements the expected time for a round trip would be roughly: “(Raspberry Pi(IPC) + HP laptop (IPC))/2”, which is approximately 5 ms. However, based on the figures above there is a difference of around 2,5 to 3,1 ms. This time difference is consumed by the router, since this hardware has been added to the configuration.

For the flood also several runs have been performed, with the averages of all runs per different number of messages as the following result: (The time measured is the time it takes from sending the first message and receiving the last message)

	Flood (s), x 1000 messages			
	1	10	100	1000
HP laptop (IPC)	0,6	4,6	56,1	578,5
Raspberry Pi (IPC)	3,2	31,6	321,5	3480,1
laptop & Pi (ISC)	2,7	24,7	235,5	2291,7



From the flood measurements a linear pattern can be deducted for both hardware (IPC), although this was not expected for the runs on the Raspberry Pi, which has no active cooling system. The same linear pattern can also be seen for the ISC measurement.

## xScope - A overview of the DDS traffic in one domain

### Introduction

### Goal of the xScope

The goal of the xScope is to visualize DDS traffic and learn the basics of DDS. The xScope is able to subscribe to various ‘Topics’ and log the data into a .csv file. The xScope is able to quickly give a graph for first impression at the end of a measurement. A detailed graph can be made with the .csv file generated.

### Getting started



## On a Raspberry Pi (linux)

### 1. Build and Install Cyclone DDS

Go to the repository of [Cyclone DDS](#). Follow the installation instructions in README.

---

**Note:** Specifying the `DCMAKE_INSTALL_PREFIX` path is not required. Directly do `cmake ..` and `cmake --build .`

---

### 2. Build and Install Python API of cyclonedds

Go to the repository of [cdds-python](#). Follow the installation steps.

---

**Note:** Jsonpickle needs to be installed for python3 (pip3 install jsonpickle)

Errors may happen while building the Chameleon python API (in 'bit' folder). The key is to build the library (ddsstubs.so and ddsc.so) for the python binding. Try the following steps if the normal steps do not work.

Copy the bit folder to the cyclone dds folder under example. Add `addsubdirectory(bit)` to the `CMakeLists.txt`. And rebuild cyclone dds. By doing so you could get the library files in the lib folder in `/build`. Copy this 2 files to `/usr/local/lib`.

---

After installation you can test the API by running

```
$ python3 test_flexy_writer.py
```

### 3. Get the project files

Clone/Download the repository in bitbucket.

### 4. Install dependencies and Run the xScope

---

**Note:** For xScope to run the following dependencies need to be installed: numpy, pandas, matplotlib and tkinter

---

```
$ pip3 install numpy
$ pip3 install pandas
$ pip3 install matplotlib
$ sudo apt-get install python python3-tk
```

---

**Note:** Be aware to set the display settings, since xScope is a gui based application.

---

```
$ cd <directory name of repo>/src/demonstrators/XScopy/Python/poc1
$ ./configure
$ python3 xScope.py
```

## On Windows

### 1. Install Visual studio with C/C++ supports

### 2. Build and install Cyclone DDS using VS compiler

**Warning:** Chameleon cannot be built on windows currently. Try to cross-compile the library file (.dll) on Linux.

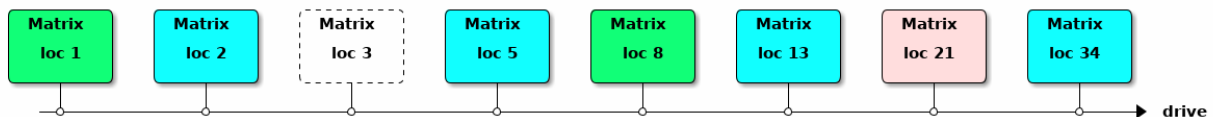
## How to run the xScope

### How to run the xScope

- Make sure the device on which the xScope will be runned is connected on a network where DDS traffic is available.
- Open xscope\_config.py and add the ‘Topics’ from the DDS traffic which you want to see in the xScope.
- Save the file.
- Run the xScope python file in a Terminal (./xScope.py).
- Wait for some DDS traffic.
- When you have enough data close the xScope by pressing ctrl+c.
- The .csv file will be generated in the same folder as the xScope.py file and a graph will be shown.
- Close the graph to close the xScope.

## Matrix Board Communication

The Matrix Board Communication DDS Demonstrator shows the application of the DDS protocol in a dynamic environment. Each board should publish the traffic it senses. Each matrix board should dynamically find the two closest upstream matrix boards and subscribe to them so it receives the traffic values they sense. According, to these two numbers the board should calculate the speed limit it will display. The traffic sensors are just random number generators that have some lower and upper bound. Similarly, the displayed speed limit starts from the maximum initial limit and decrements or increments in discrete steps while never going above the maximum.



## Implementation

The Matrix Board Communication currently has the following implementations. Note that in some languages there are multiple proof of concepts in order to show that the dynamic aspect can be implemented in various ways.

### Matrix Board Communication in C

**authors** Stefanos Florescu

**date** March 2020

Before following this guide in order to execute the matrix board communication make sure you have successfully installed CycloneDDS ([Setup Guide for CycloneDDS](#)).

## Building the executable

Make sure your local repo is up to date with the [HighTech-nl repo](#) and move into the directory / dds-demonstrators/src/demonstrators/MatrixBoard/C:

```
$ mkdir build
$ cd build
$ cmake ..
$ make
```

## Running the Matrix Board Communication

The demonstrator can be executed either on one machine by creating different processes (in different terminal windows) or by creating instances of the matrix boards on different nodes (i.e. on Raspberry Pis). A combination of the two methods can also be utilized, in other words running multiple processes on each of the nodes in the network.

In order to run a matrix board instance execute the following command by providing an integer number which is the ID of that particular board (i.e. a positive number):

```
$ ./MBC <MBC_ID>
```

**Note:** In the case that the wrong network interface is chosen by DDS at startup of the application see the notes in [Setup Guide for CycloneDDS](#) for details on how to utilize the desired network interface.

**Note:** See [Setting up Raspberry Pi network](#) for a complete guide on Raspberry Pi networking as well as the IP addresses for the network that is already in place.

## Running the Matrix Board Communication in a Docker Container

Similarly to what we described above, the MBC experiment can be executed from a Docker Container. First make sure you follow the guide on how to build the Docker Image which can be found in the [Docker Guide: How to create and run a Docker Image](#) under “Building Demo Image”. First create a new container using:

```
$ docker run -it --network="host" --name <container name> <image name>
```

You can open another terminal/Powershell and have as many processes on that container as you want using per terminal:

```
$ docker exec -it <existing container name> bash
```

Navigate into the MatrixBoard/ directory and run one of the three MBCs using:

```
$ ./MBC_poc<1 or 2 or 3> <MBC_ID>
```

A demo with two different containers can be seen in the video below. The terminals on the left side are processes of container named dds1 while the ones on the right belong to container dds2. Then we change the network interface so that it works properly on Windows (more details on this can be found in the warning right below). Finally, we start Matrix Board with ID 1 and 3 in container dds1 and Matrix Board with ID 2 and 4 in dds2. This configuration demonstrates the inter-container communication as well as the communication within the same container.

**Warning:** Due to the networking issue of the Docker Desktop (see in [Docker Guide: How to create and run a Docker Image](#) under “Running Docker Containers” for more info) on Windows/MacOS we need to change the DDS networking interface in the `docker_cyclonedds_config.xml` in `/root/`. In the video below you can find an example of how to do that. On a Linux machine this is not required and your container will communicate over the network.

### Matrix Board Communication: Proof of Concept 1 (poc1)

The first version of the Matrix Board Communication (MBC) was a simple implementation that has as a goal to introduce the user to a dynamic application of the DDS protocol. Each matrix creates the following entities:

- 1 Participant on the default domain.
- 1 Write topic, used to publish the traffic recorded by that particular MBC.
- 1 Writer, used to publish the traffic of that particular MBC.
- Array of read topics, starting from the ID of that particular MBC and up to a predefined constant number.
- Array of readers, starting from the ID of that particular MBC and up to a predefined constant number.

In order to achieve the dynamic switching of boards, we create an array of topics and an array of readers on all those topics up to a predefined number (`MAX_SEARCH`). As a result, the dynamic behavior is achieved in a purely algorithmic method, by simply iterating up to that `MAX_SEARCH` until we find the two closest active MBCs. Furthermore, each MBC continuously writes its own topic without checking if there are any active readers. This method, albeit simple, will not have great performance especially due to the following reasons:

1. Large switching time, especially for large gaps between MBCs (i.e. between board with ID 1 and board 9000).
2. Increased overhead in each polling iteration, since the closest MBCs are always recalculated.
3. Increased algorithmic complexity, due to two nested for loops used for the dynamic search algorithm.
4. Unnecessary network usage, since in every polling iteration we write the traffic without checking for events (i.e. new data being available and/or active readers on that topic).
5. Unnecessary memory usage, since we only require at most 2 read topics and 2 readers per MBC and we actually have two arrays of size `MAX_SEARCH - MBC_ID` with signed 32-bit integers at each entry.

### Matrix Board Communication: Proof of Concept 2 (poc2)

The second version of the Matrix Board Communication (MBC) has as a goal to expand upon poc1 while solving some of the issues of poc1 that were presented previously. This solution is based on the use of status and event driven DDS implementation. Similarly to poc1, each matrix creates the following entities:

- 1 Participant on the default domain.
- 1 Write topic, used to publish the traffic recorded by that particular MBC.
- 1 Writer, used to publish the traffic of that particular MBC.

- Array of read topics, starting from the ID of that particular MBC and up to a predefined constant number.
- Array of readers, starting from the ID of that particular MBC and up to a predefined constant number.

The main search algorithm remains the same, but now each MBC writes to its own topic only when there are matched active MBCs that should read that topic. This is achieved by using on the writer side the `dds_get_publication_matched_status` function which returns a struct containing information on the number of matched readers, the total number of matched readers, the change of readers and the handle of the last matched reader (more information on this function can be found [Statuses for DataWriters](#)). Similarly on the reader side we introduce the function `dds_get_subscription_matched_status`, which provides the same information about matched writers to that particular reader (more information on this function can be found [Statuses for DataWriters](#)). As a result, we can now see if we have an active writer to a topic without having to do a `dds_take` or `dds_read` to see whether there are any available samples on that topic. Although, these additions do not fix all of the performance issues introduced with poc1, the list of performance bottlenecks is reduced to the following:

1. Lower than poc1 but still large switching time, especially for large gaps between MBCs (i.e. between board with ID 1 and board 9000).
2. Reduced overhead by comparison to poc1 but still increased overhead in each polling iteration, since the closest MBCs are always recalculated.
3. Increased algorithmic complexity, due to two nested for loops used for the dynamic search algorithm.
4. Unnecessary memory usage, since we only require at most 2 read topics and 2 readers per MBC and we actually have two arrays of size `MAX_SEARCH - MBC_ID` with signed 32-bit integers at each entry.

### Matrix Board Communication: Proof of Concept 3 (poc3)

The third version of the Matrix Board Communication (MBC) is the most complex and has as a goal to solve all of the previously introduced issues. The approach in this case is completely new by introducing the following novelties:

- Membership, each MBC will have to announce its liveness by writing to a dedicated topic.
- Liveness changes, using listeners attached to callback functions (more information can be found [Types of Listeners](#)).
- Usage of `sample_info` to extract valuable sample information.
- Hash tables, each MBC will have a hash table where each active MBC will be stored.

Each MBC will have the following entities:

- 1 Participant on the default domain.
- 1 Write topic, used to publish the traffic recorded by that particular MBC.
- 1 Writer, used to publish the traffic of that particular MBC.
- 2 Readers, used to read the traffic published by other MBCs.
- 1 Membership topic, where the liveness of all active MBCs can be found.
- 1 Membership writer, used to announce the liveness of that particular MBC.
- 1 Membership reader, used to read the membership topic.

In a nutshell, a new active MBC will announce that it is alive by writing to the membership topic its own ID number, all of the other MBCs will detect this liveliness change using their liveliness change callback function and will read the membership topic. Upon reading the ID of the new MBC each of the previously active MBCs will add that new MBC in their hash table using the modulo of the publication handle of that particular sample over the predefined hash table size as the hash code. Furthermore, after adding the new MBC all of the previously active MBCs will also write to the membership topic their own IDs. Therefore, the new MBC will add all of the pre existing MBCs to its own hash table. Similarly, when a MBC goes offline, the other MBCs detect that event through the same liveliness change callback function and extract the publication handle of the offline MBC using the function `dds_get_liveliness_changed_status` which returns a similar struct to the functions presented in poc2. Using that handle each of the online MBCs removes the offline MBC from their hash table.

At this point it is interesting to present some of the functions the hash table utilizes for all of the operations mentioned above.

```
//used to calculate the hash code.
dds_instance_handle_t hashCode(dds_instance_handle_t key);

//search for a MBC based on its handle.
struct DataItem* hash_search(dds_instance_handle_t key);

//insert a new MBC based on its handle.
void hash_insert(dds_instance_handle_t key, int32_t data);

//delete a MBC based on its handle.
struct DataItem* hash_delete(struct DataItem* item);

//print all contents of the hash table (i.e. mainly for debugging).
void hash_display();

//count the number of entries in the hash table.
int hash_count();

//find the smallest ID number in the hash table (excluding the ID of the MBC_
↳requesting this).
int hash_return_first_min(int32_t mbc_id);

//find the second smallest ID number in the hash table (excluding the ID of the MBC_
↳requesting this).
int hash_return_second_min(int32_t mbc_id, int32_t min1);
```

Using the two last functions from the code block above, each MBC can determine to which other MBCs it has to subscribe in order to receive their traffic values. This implementation solves the algorithmic complexity issue since the hash table finds entries based on their hash and not by looping over all of them. Furthermore, memory is used efficiently without keeping unnecessary entities and hash entries. Overall, this implementation minimizes the time required by the dynamic subscription and outperforms the previous proofs of concept by a vast margin.

## Links

- Statuses for DataWriters: <https://tinyurl.com/rx269d5>
- Statuses for DataReaders: <https://tinyurl.com/wf8bgdv>
- Types of Listeners: <https://tinyurl.com/vfp5adr>

## Matrix Board Communication in C++

**authors** Joost Baars

**date** June 2020

### Description

This page describes the matrix board demonstrator in C++. The matrix board demonstrator is explained in [Matrix Board Communication](#). This page starts with how the application can be built and executed. Afterward, the structure of the application is described in detail.

The matrix board C++ directory can be found in `src/demonstrators/MatrixBoard/C++/`. This directory contains the matrix board application in C++.

This matrix board application contains a similar implementation compared to the `poc3` of the C implementation ([Matrix Board Communication in C](#)). Therefore, this is a dynamic implementation of the matrix board application.

### Dependencies

The necessary dependencies for succesful compilation of the matrix board application in C++ are the following:

- CMake
- C++17 compiler (Gcc 7+ for example)
- Cyclone DDS library

Installing Cyclone DDS and the other dependencies can be found in: [Clean Install with internet](#).

### Building

First, go to the C++ matrix board directory (see [Description](#)).

Execute the following commands:

```
mkdir build && cd build
cmake ..
make
```

These commands compile the code and create the executable for the C++ matrix board application.

### Execution

The application must be executed using one parameter. The application can be executed using the following command:

```
./MatrixBoard <Matrixboard ID>
```

A more thorough description of the parameters can be found when executing the application with no parameters.

---

**Note:** Execution of the program

The `<Matrixboard ID>` parameter should be unique for every matrix board. Additionally, it may not be less than 1.

The order at which matrix boards are started does not matter.

---

### Example

For example, two matrix boards can be started to see the communication between the two applications.

Open two terminals and go to the location where you built the matrix board application.

Execute the following in one terminal

```
./MatrixBoard 10
```

And the following command in the other terminal:

```
./MatrixBoard 15
```

When the second matrix board is started, logging should be shown. The logging that can occur is described in the chapter below.

### Logging information

The matrix board application prints information to the terminal/standard output. This logging contains information that is classified with different tags.

Each tag is described below.

#### INFO

The INFO tag shows general information of the matrix board. It contains a starting message. The INFO tag is also used to display the speed of the matrix board. If the maximum speed changes, the new maximum speed is printed using the INFO tag.

When new traffic data is received, it is also shown with the INFO tag. The newly received traffic data is printed towards the terminal including the matrix board ID from which the traffic data came.

#### CONNECTED or DISCONNECTED

The CONNECTED and DISCONNECTED tag are shown when changes occur with the subsequent matrix boards.

An example of the logging can be seen below. In this example, only one subsequent matrix board is connected. There is no second subsequent matrix board on the network. The connected subsequent matrix board has an ID of 5.

```
[2020-06-04 00:10:07.677]: CONNECTED: Subsequent matrix board: 5
[2020-06-04 00:10:07.678]: DISCONNECTED: After the subsequent matrix board_
↪disconnected!
```

Each of the two subsequent matrix boards has its own message regarding if it is connected or not.

#### ERROR

The ERROR tag is used to show that an error occurred with the communication or the matrix board.

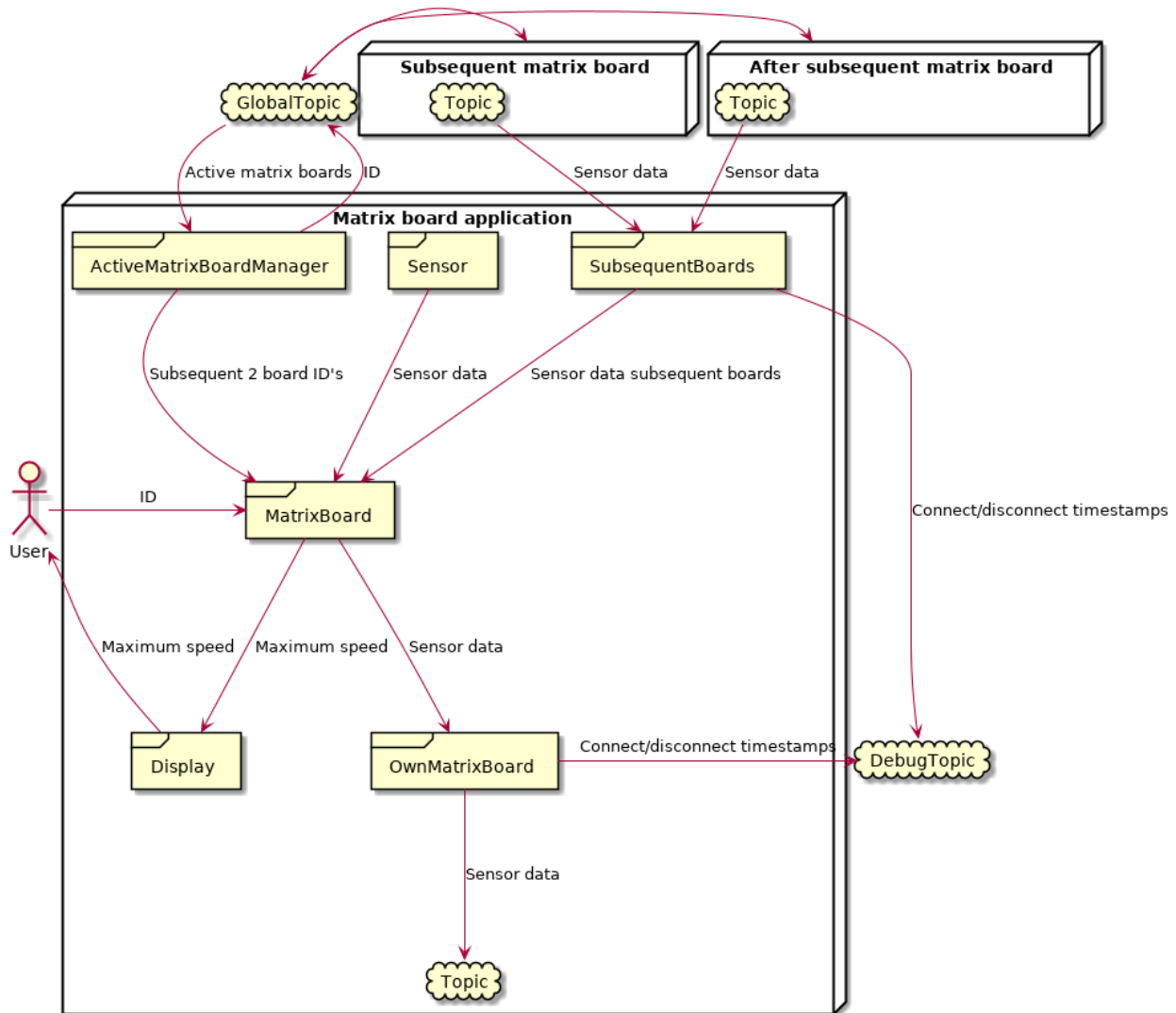
### Implementation

This matrix board implementation works dynamically. Therefore, the order of adding matrix boards does not matter as well as the ID's of the matrix boards.



For example, a matrix board with ID 1 can have a matrix board with ID 1000 as the subsequent matrix board (if no other matrix boards exist in between).

In the deployment diagram below, the communication between different sub-classes can be seen. The clouds in the diagram are the different DDS topics. This diagram only shows the communication of one of the matrix boards.



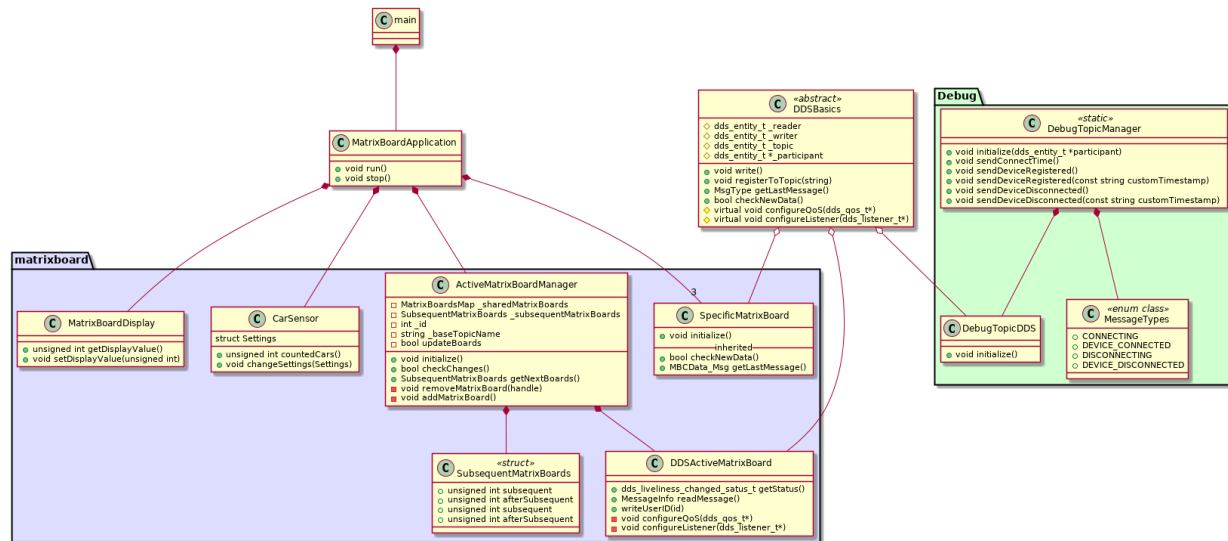
This diagram shows that there is one “global” topic, which is a topic that does not belong to a specific matrix board. All matrix boards write to that topic once at the beginning of their lifecycle. Afterward, they only read the ID’s of the other matrix boards from that topic.

The other topics belong to a matrix board. The matrix board that belongs to the topic (same ID) writes its sensor data to that topic. The preceding two matrix boards read the data from that particular topic.

The debug topic can be used to calculate the time between connecting and being connected to a topic. As well as disconnecting and being disconnected from a topic. This is not done in the matrix board application, but the possibility is there because of the debug topic.

## Class diagram

The class diagram is shown below. This class diagram shows that the intelligence of the system is the `MatrixBoard` class. This class executes everything and contains a `run()` function that runs the matrix board application.



Each part of DDS is separated into its own class. This is divided into 2 classes and one abstract base class.

The `ActiveMatrixBoardManager` class manages the “global” topic and can return the ID’s of the two subsequent matrix boards.

The `DDSSpecificMatrixBoard` class manages the DDS part of the “global” topic.

The `SpecificMatrixBoard` manages a matrix board topic with a certain ID. This class can read and write from the topic it is connected to. This class is used for the 2 subsequent matrix boards (read-only) and for its own matrix board (write-only). Therefore, there are 3 classes of these.

There are two packages in this class diagram. One for the matrix board functionalities. The other is the debug package. The `DebugTopicManager` class can be used for sending debug messages to a DDS debug topic (using the `DebugTopicDDS` class). The `DebugTopicManager` class is a fully static class, so the debug functions are easily accessible in the matrix board.

## Global topic / ActiveMatrixBoards class

This chapter describes the application in a technical manner.

The `ActiveMatrixBoards` class contains the logic for defining the subsequent 2 matrix boards. This class uses the `DDSSpecificMatrixBoard` class for communication with the “global” topic.

The settings for the “global” topic are configured in the `DDSSpecificMatrixBoard` class. The “global” topic is configured to have a history of 1 and a durability transient local. The history of 1 ensures that the last message of each writer is stored and can be resent. The durability transient local ensures that if a new reader connects, it receives previously send messages from the writer. This durability setting also specifies that the previously sent messages are stored by the writer. This is important because if the writer disconnects, we don’t want to see it’s message anymore. This combination makes sure that the last message of a writer is resent to a newly connected reader, but only if that writer still exists.

Each matrix board only writes once to the “global” topic, containing the ID of that matrix board. This message will automatically resend if a new matrix board connects (this is done by DDS). Therefore, all matrix boards know the ID of every other matrix board. Because when they connect, they receive the last message of each matrix board (containing

the ID). And if they are alive when a new matrix board connects, it just reads the message of the newly connected matrix board.

The liveliness is also configured for the “global” topic. This is used for removing disconnected matrix boards and reconnecting to a different matrix board that is alive. DDS uses handles for each DDS instance. This handle is added to every DDS message and is also added to the liveliness status. Therefore, this can be used to know what matrix board disconnected when the liveliness changes.

The `ActiveMatrixBoardManager` class contains an ordered map with the ID and the handle of each matrix board. When a matrix board disconnects, this matrix board is removed from the map based on its handle. This map is ordered based on the ID of the matrix board. Therefore, the subsequent matrix boards can easily be found within the map. Scalability wise, this map becomes larger as more matrix boards are on the network. Therefore, this method is not ideal for a fully scalable matrix board network.

When the `ActiveMatrixBoardManager` class registers to the topic, it reads all existing messages from the topic and fills the ordered map with the existing matrix boards. If there is a liveliness update, it checks if a device is added or removed. If a device got added, it reads the new message on the topic. If a device got removed, it reads the handle of the disconnected matrix board and removes that device from the map. Afterward, it checks if the subsequent matrix boards got changed. If so, this is updated to the new ID's.

## MatrixBoard class

This class executes everything and contains the `run` loop for the matrix board application.

At the beginning of the `run` loop, the `registerToTopic()` functions are executed for both the `OwnMatrixBoard` and the `ActiveMatrixBoards` classes. This initializes DDS to register to the topic. Furthermore, a timer is initialized to send the traffic data to its own topic with a certain interval. This timer is set on 1 second. So traffic data is sent to its own topic every second.

There are mainly 3 parts in the matrix board `run` loop. The first is checking the timer. If the timer has passed, the newest sensor data is requested and send to its own topic using the `OwnMatrixBoard` class. Additionally, the display of the matrix board is also updated in this timer.

The second part is checking for updates on the “global” topic. This is done by executing the `checkChanges()` function in the `ActiveMatrixBoards` class. If there are changes, the `SpecificMatrixBoard` objects that point to the subsequent matrix boards are updated. These objects are stored in a so-called *smart-pointer* which handles the allocation and deallocation of the resource. This *smart-pointer* is simply set to a new `SpecificMatrixBoard` object when the ID's of the subsequent matrix boards changes.

The third and last part of the `run` loop reads the traffic data from the 2 subsequent matrix boards (if they exist). First, the *smart-pointer* is checked if there is an object stored. If so, the `checkNewData()` function from the `SpecificMatrixBoard` class is executed. At last, this data is read using `getLastMessage()` function if new data is received. If there was no object stored in the *smart-pointer* in the first place, then nothing would happen. As there is apparently no subsequent matrix board.

## Specific Matrixboard class

This class connects to a specific matrix board topic. Each matrix board is connected to its own topic and writes its own sensor data towards that topic. Additionally, each matrix board is connected to 2 subsequent matrix board topics (if they exist). Therefore, there are a maximum of 3 specific matrix board objects created.

The specific matrix board class creates its own DDS participant. This is done because specific matrix board objects should be easily deletable and new objects for different topics should be easily creatable. If the participant was not managed by the specific matrix board, then the previously allocated readers or writers of the specific matrix board would stay alive when the specific matrix board object would be removed. This results in no liveliness change when

it should. Additionally, this would use more memory and would have a negative impact on the dynamic aspect of the matrix board application.

The specific matrix board has by default a liveliness callback function for the writer and a different callback function for the reader. The reader gets a callback if the liveliness changes of the writer on the topic. The writer gets a callback when a reader connects or disconnects from the topic. The use of these callbacks are only the debug messages. To detect when a device connects/disconnects from a certain topic.

### Debug topic

The class diagram also contains a debug topic. This debug topic is added for debug purposes. A timestamp is sent to the debug topic when a matrix board connects to a specific matrix board topic and when it is connected to that topic. When a device disconnects from the specific matrix board topic, a timestamp is also sent to the debug topic.

The debug functionalities can be used to see when a device connects or disconnects. Additionally, the time of connecting/disconnecting can be measured.

### Links

- [DDS durability](#)

### Matrix Board Communication in Python

For the matrix board different implementations have been created.

#### Proof of concept 1 - python

This implementation uses a single topic to which all nodes write and read. On detection of another reader it is determined whether the information for this reader is relevant.

---

**Note:** For this implementation to run the following dependency needs to be installed: pynput This dependency is used for the keyboard strokes to be handled during the execution.

---

```
pip3 install pynput
```

---

**Note:** Be aware to set the display settings, these are required

---

```
cd <name of the repo>/src/demonstrators/MatrixBoard/Python/poc1
python3 node.py <location marker>
```

for <location marker> 17.1, 17.2, etc. can be used.

#### Proof of concept 2 - Python

This implementation uses more than one topic. Each node writes to its own topic and determines if there is a next topic to read from. The information of the two next topics will be used. In case one of the next topics is not a preferred topic and a preferred one becomes available, the preferred one will replace the other.

Usage:

```
python3 MatrixBoard.py <topic number>
```

The <topic number> must be a value: 1..10

## Compared to other protocols

Below are the measurements using RoundTrip, where ms/msg represents the amount of time it takes for the message to make a round trip.

RoundTrip		
Language: python3		
	IPC measurements	ISC measurements
protocol	ms/msg	ms/msg
MQTT	4.694029412	6.052571429
ZMQ	0.540805556	1.617088235
DDS	under construction	under construction

Below are the measurements using RoundTrip and testing the bandwidth, where ms/msg represents the amount of time it takes for a bunch of messages to make a round trip.

Bandwidth		
Language: python3		
	IPC measurements	ISC measurements
protocol	ms/msg	ms/msg
MQTT	9.5	6.5
ZMQ	0.4	7.3
DDS	under construction	under construction

## Bandwidth

### MQTT-Bandwidth

to start the bandwidth test go to either the isc or ipc folder and run: sh run.sh they both use the same MQTT.py script be sure to read the corresponding readme to make sure the necessary components are in place

### MQTT-IPC

run the run.sh file and it will set up a local round trip

requirements: mqtt must be installed then this script can run locally

### MQTT-ISC

Running the run.sh shel script wil setup a roundtrip over four different systems.

The following components must be in place in order for it to work: three systems(raspberries) with python paho mqtt installed. one system with mosquitto and python paho mqtt installed The ips of the external systems must be 192.168.0.200, 192.168.0.201, 192.168.0.203 run.sh must be run on a system with ip 192.168.0.202

### ZMQ-bandwidth

there are two folders ipc and isc each folder contains a run.sh. Run that file and it will setup either a local roundtrip using the bandwidth method(ipc) or it will launch programs spread over 4 raspberries the isc folder also has a loop.sh which simply runs run.sh continuously

### Matrixboards

### MQTT

#### Description

The problem description for the matrixboard can be found at: [Matrix Board Communication](#).

#### Implementation

There are different implementations of the MQTT matrixboard solution. A short description is made for the execution of a particular implementation. This description contains information about how the implementation can be compiled and executed.

This list contains the different implementations:

#### MQTT Matrixboard

**authors** Sam Laan

**date** March 2020

#### Description

This page contains the information needed to run the MQTT C++ matrixboard solution. It is assumed that the MQTT broker and paho C++ library are already installed. If not please refer to the MQTT C++ page ([MQTT C++](#)).

#### How to run

---

**Note:** For collecting measurements The collection of measurement data is built within the application. It is stored in a .csv file. To create this file please refer to [Measurement file](#)

---

To run the program navigate to the /src/demonstrators/ComparedToOtherProtocols/MQTT/MatrixBoard/C++ folder of the repository. The source code and CMakeLists of the program are in this folder. To build the solution perform the following commands:

```
mkdir build
cd build
cmake ..
make -j4
```

An executable called 'mqttmatrixboard' should have appeared in the build folder. To make the program work a broker must be running. This broker can be started by using the following command in terminal:

```
mosquitto
```

When the broker is started, it's time to start the program. This can be started by running the following command:

```
./mqttmatrixboard
```

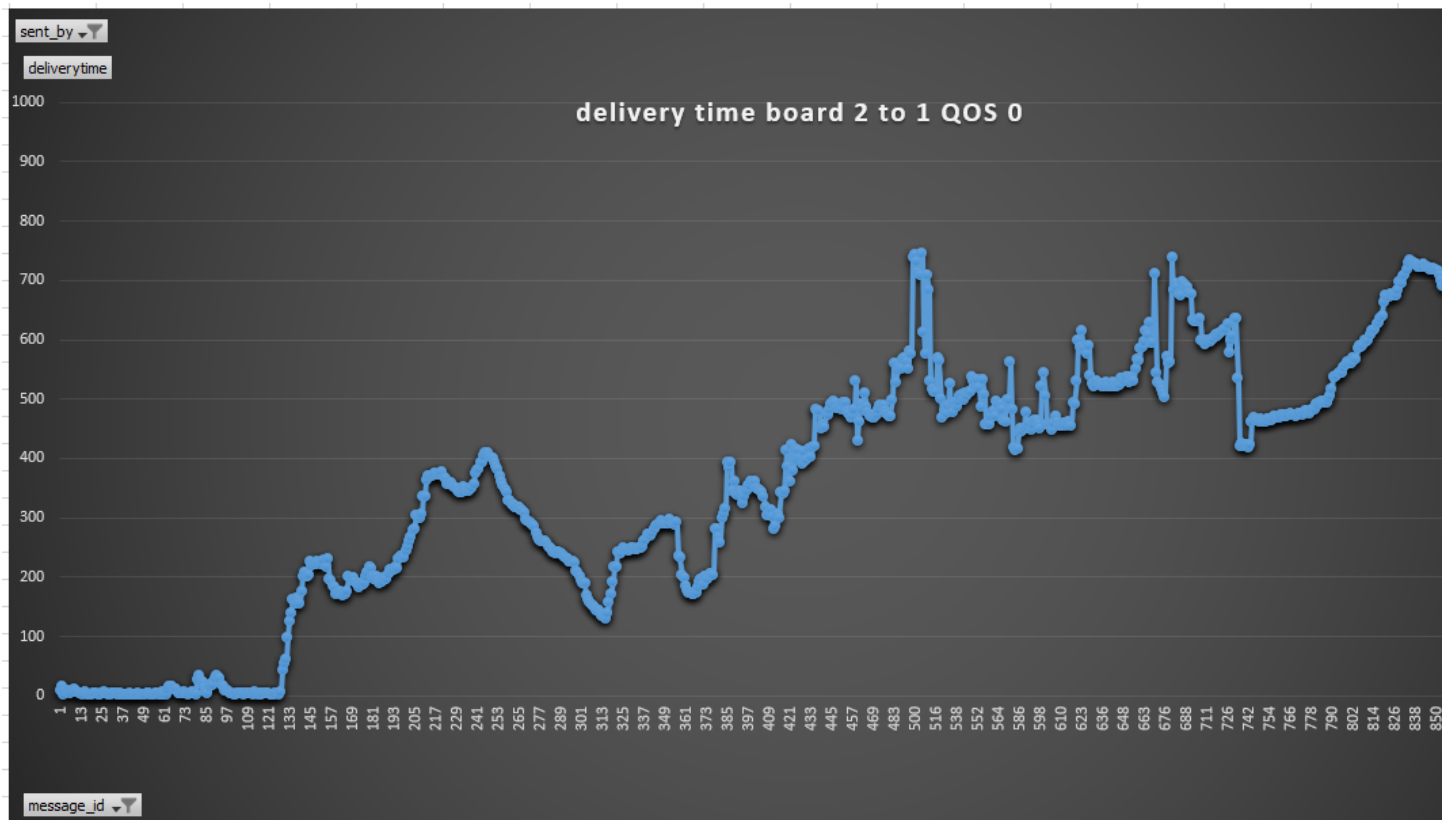
The matrixboard application will ask for a number. This number is the id of the matrixboard. The matrixboard will publish to a topic that has the matrixboard id as its name. Other matrixboards can subscribe to this topic to receive the information. Matrixboards will subscribe according to the matrixboard problem description ([Matrix Board Communication](#)). Besides that, this number will also be used as client id within MQTT. If you try to start another client(matrixboard) with the same ID it will kill the old matrixboard and then start the new one.

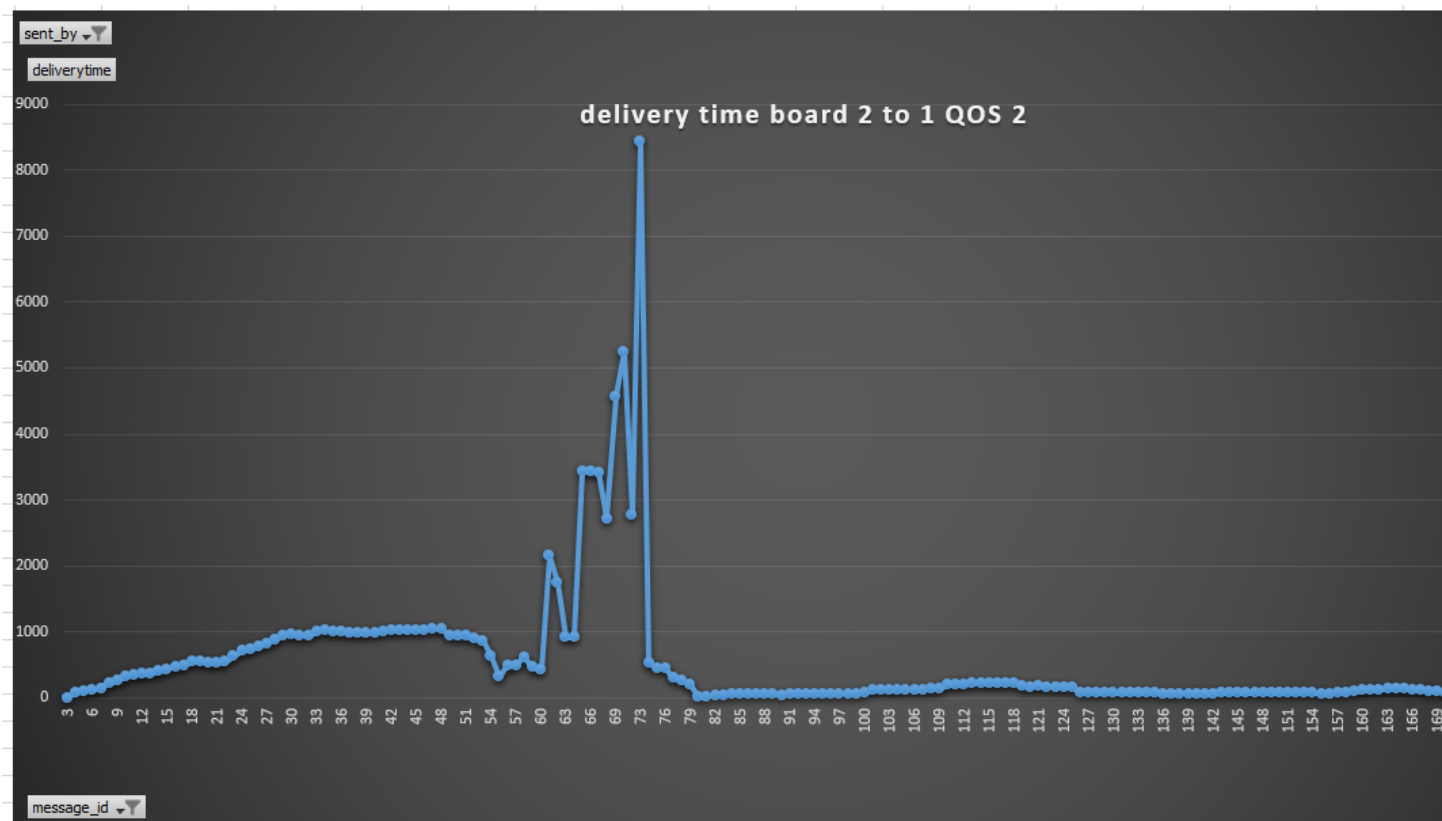
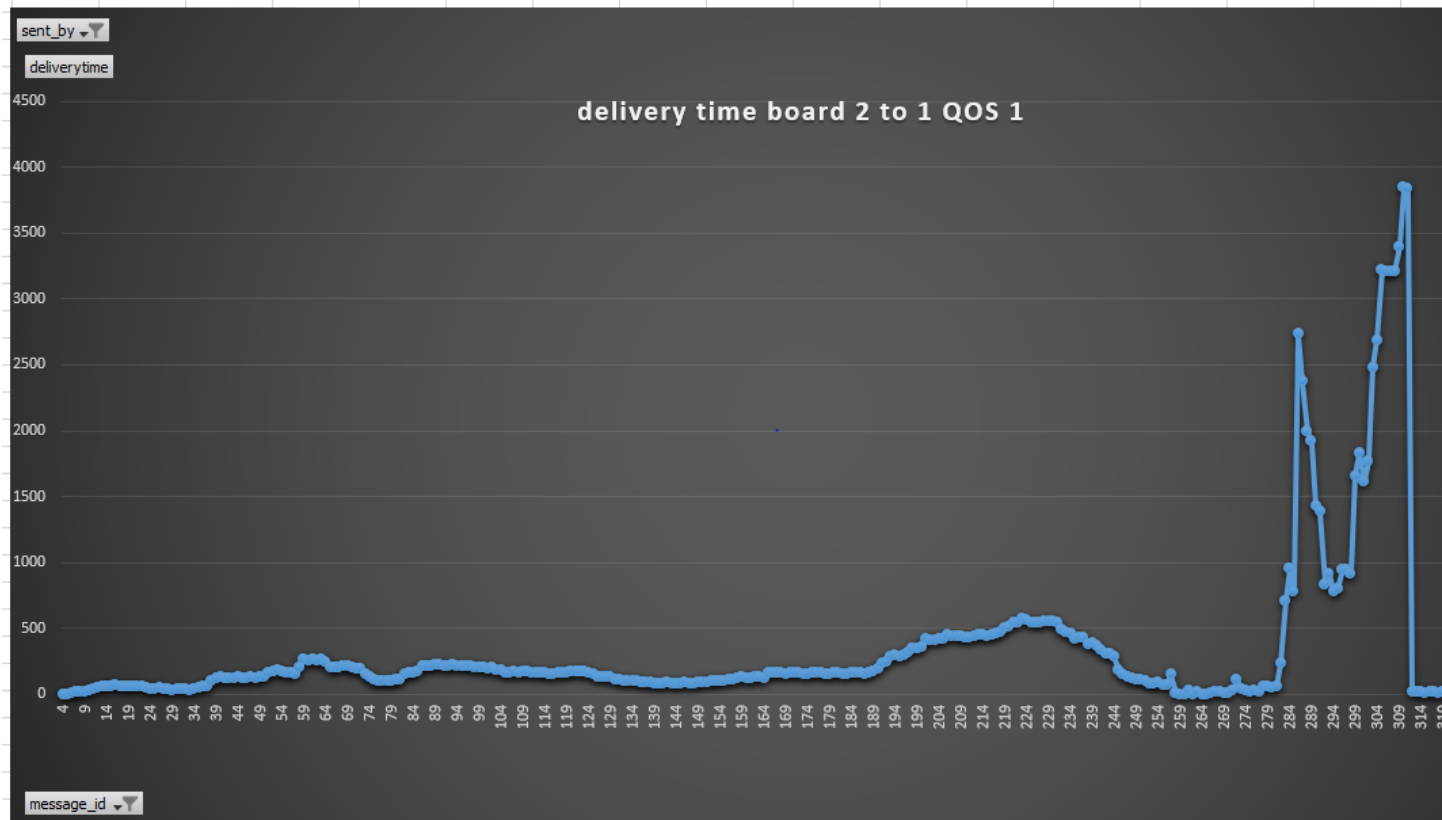
## Performance

The collection of performance measurements is built into the application itself. It writes the time it sends and receives messages to a CSV file. This CSV file can be used to visualise the data using various tools to do so. In this case, Excel has been used. The choice for Excel was made because it is a commonly known and used program and it is installed on a Capgemini/Sogeti laptop by default. The processed\_measurements.xlsx file shows an example of how to visualise the CSV data using Excel and Power Pivot. More on Excel and Power Pivot can be found here [Power Pivot](#).

## Findings

The performance test was executed with 3 matrixboards and resulted in the following graphs (time is measured in ms).







These graphs show that the QoS 0 can send the most messages in a short amount of time. The downside is that the application crashed after about 950 messages because of an overflowing buffer MQTT error [-12]. This is because MQTT was not designed to handle this much messages in such a short amount of time(see the resilient mqtt application link). With QoS 1 the delivery time is more stable, this is because QoS 1 is slower which causes the buffer to not overflow during the test. An increase in delivery time can be seen once board 3 is up(around 184). Besides this, the spike is when board 3 goes the down. After that messaging resumes with low delivery time. Almost the same can be said for QoS 2 although QoS 2 is a bit slower compared to QoS 1. Taking almost a second for receiving a message whilst 2 boards are up. This data was collected using the first version of the MQTT matrixboard solution. Delivery time is the time between the sending of a message from one board To the time of receiving the message by another board.

## Measurement file

The file in which the measurements are stored in CSV format is called measurements.csv. An executable is built during the ‘make’ process to create this CSV file in correspondence with the data that will be stored in it. This executable deletes any existing file called measurements.csv in the build folder and creates a new one containing the data headers only. To start the executable run the following command:

```
./measurementsfiles
```

## Links

- resilient mqtt application: <https://www.hivemq.com/blog/are-your-mqtt-applications-resilient-enough/#throttling>

## ZMQ

**authors** Sam Laan

**date** March 2020

## Description

The problem description for the matrixboard can be found at: *Matrix Board Communication*.

## Dynamic discovery problem

---

**Note:** From the ZeroMQ guide (<http://zguide.zeromq.org/page:all>)

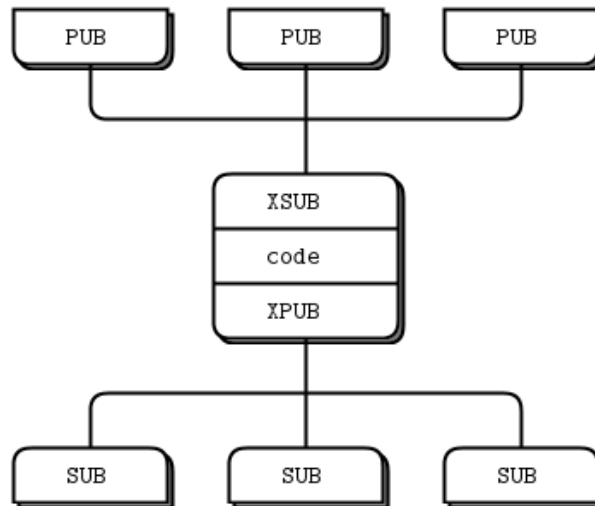
---

One of the problems you will hit as you design larger distributed architectures is discovery. That is, how do pieces know about each other? It’s especially difficult if pieces come and go, so we call this the “dynamic discovery problem”.

There are several solutions to dynamic discovery. The simplest is to entirely avoid it by hard-coding (or configuring) the network architecture so discovery is done by hand. That is, when you add a new piece, you reconfigure the network to know about it. In practice, this leads to increasingly fragile and unwieldy architectures. Let’s say you have one publisher and a hundred subscribers. You connect each subscriber to the publisher by configuring a publisher endpoint in each subscriber. That’s easy. Subscribers are dynamic; the publisher is static. Now say you add more publishers.

Suddenly, it's not so easy any more. If you continue to connect each subscriber to each publisher, the cost of avoiding dynamic discovery gets higher and higher.

The very simplest answer is to add an intermediary. A static point in the network to which all other nodes connect. In classic messaging, this is the job of the message broker. ZeroMQ doesn't come with a message broker as such, but it lets us build intermediaries quite easily. It's better to think of intermediaries as simple stateless message switches. A good analogy is an HTTP proxy; it's there but doesn't have any special role. Adding a pub-sub proxy solves the dynamic discovery problem in our example. We set the proxy in the "middle" of the network. The proxy opens an XSUB socket, an XPUB socket, and binds each to well-known IP addresses and ports. Then, all other processes connect to the proxy, instead of to each other. It becomes trivial to add more subscribers or publishers.



We need XPUB and XSUB sockets because ZeroMQ does subscription forwarding from subscribers to publishers. XSUB and XPUB are exactly like SUB and PUB except they expose subscriptions as special messages. The proxy has to forward these subscription messages from subscriber side to publisher side, by reading them from the XPUB socket and writing them to the XSUB socket. This is the main use case for XSUB and XPUB.

### Implementation

There are different implementations of the zmq matrixboard solution. A short description is made for the execution of a particular implementation. This description contains information about how the implementation can be compiled and executed.

This list contains the different implementations:

#### ZMQ C++ Matrixboard

**authors** Sam Laan

**date** March 2020

### Description

This page contains the information needed to run the ZMQ C++ matrixboard solution. It is assumed that you have already installed the zeromq C and C++ libs. If not please refer to the ZMQ C++ page([ZMQ C++](#)).

## How to run

---

**Note:** For collecting measurements The collection of measurement data is built within the application. It is stored in a .csv file. To create this file please refer to [Measurement file](#)

---

To run the program navigate to the /src/demonstrators/ComparedToOtherProtocols/ZMQ/MatrixBoard/C++ folder of the repository. The source code and CMakeLists of the program are in this folder. Now to build the solution perform the following commands:

```
mkdir build
cd build
cmake ..
make -j4
```

Two executables called “zmqmatrixboard” and “matrixboardproxy” should have appeared in the build folder. These can be started using the following commands:

```
./matrixboardproxy
./zmqmatrixboard
```

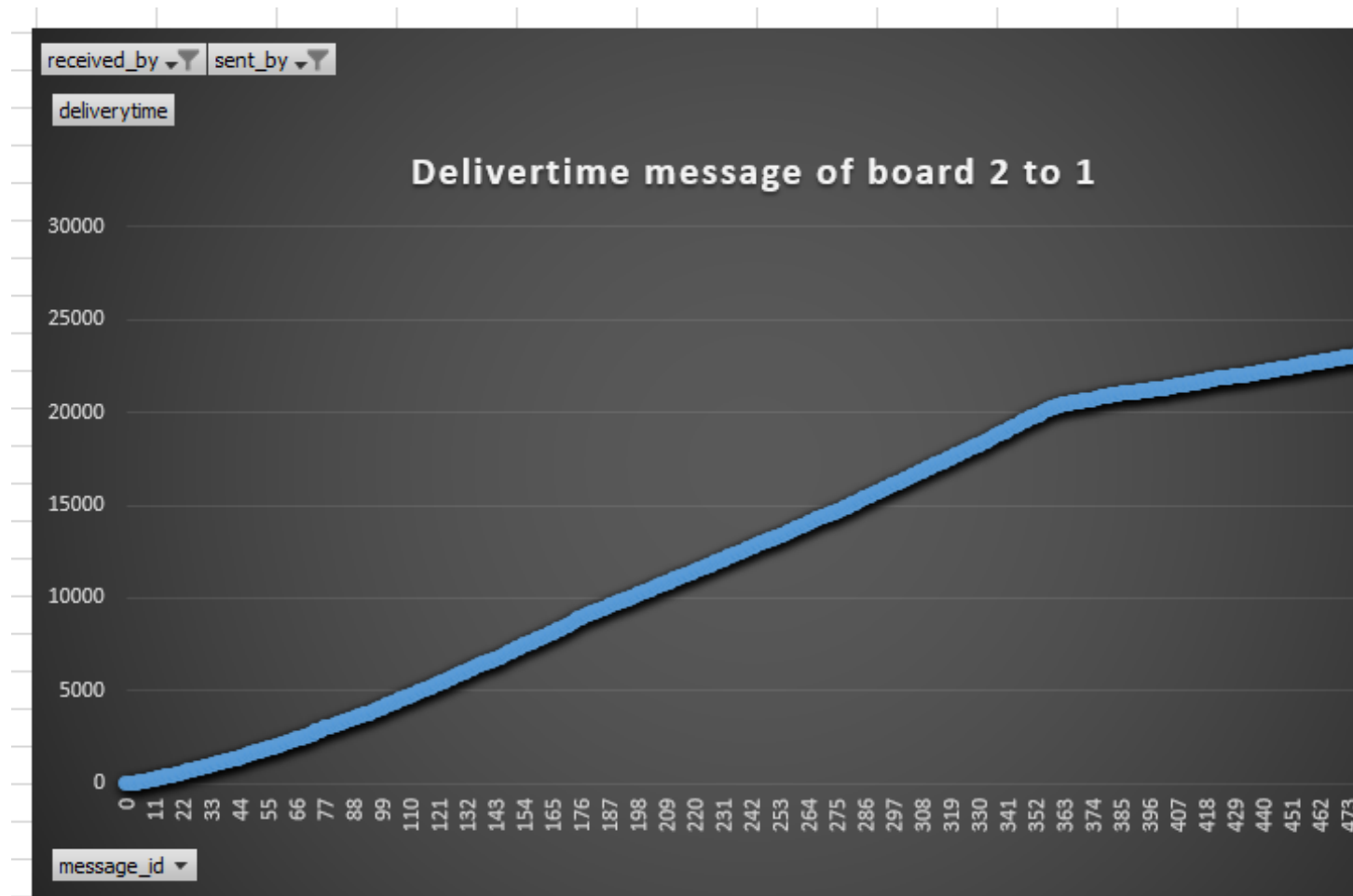
The matrixboardproxy will show on which ports it runs. The proxy is necessary for discovery, more on that in the section below. The zmqmatrixboard program will ask for a number. This number can range from 1 till 9. This number will be the id of the matrixboard and will determine on which port the matrixboard publishes.

## Performance

The collection of performance measurements is built into the application itself. It writes the time it sends and the time it receives messages to a CSV file. This CSV file can be used to visualise the data using various tools to do so. In this case, Excel has been used. The choice for Excel was made because it is a commonly known and used program and it is installed on a Capgemini/Sogeti laptop by default. The processed\_measurements.xlsx file shows an example of how to visualise the CSV data using Excel and Power Pivot. More on Excel and Power Pivot can be found here [Power Pivot](#).

## Findings

The performance test was executed with 3 matrixboards and resulted in the following graphs(time is measured in ms).



The graph shows that the delivery time steadily increases. This is because the application can send messages faster than it can process them. This results in a message queue which keeps on growing. The last message was not received because the application was stopped before it could receive it. This explains the drop in the graph at the last message.

### Measurement file

The file in which the measurements are stored in CSV format is called `measurements.csv`. An executable is built during the 'make' process to create this CSV file in correspondence with the data that will be stored in it. This executable deletes any existing file called `measurements.csv` in the build folder and creates a new one containing the data headers only. To start the executable run the following command:

```
./measurementsfiles
```

### MQTT, DDS, ZMQ Matrix board application in C++

**authors** Sam Laan

**date** Jun 2020

## Description

The matrix board C++ directory can be found in `src/demonstrators/ComparedToOtherProtocols/Multiple/MatrixBoard/C++/`. This directory contains the matrix board application in C++.

This matrix board application contains a similar implementation compared to the `poc3` of the C implementation (*Matrix Board Communication in C*). This is therefore a dynamic implementation of the matrix board application.

The application was written for usage with real-time posix threads. It is also possible to run it on a non real-time posix system.

## Dependencies

The dependencies necessary for successful compilation of the matrix board application in C++ are the following:

- CMake
- C++17 compiler (Gcc 7+ for example)
- paho.mqtt.cpp library (installation: [MQTT C++](#))
- cppzmq library (installation: [ZMQ C++](#))
- Cyclone DDS library (installation: [Setup Guide for CycloneDDS](#))
- MQTT server compatible with MQTT 5 (mosquitto (installation: [MQTT C++](#)) for example)

## Building

To build the application run the following commands in the matrix board application source directory.

```
mkdir build && cd build
cmake ..
make
```

These commands will compile the source code and generate an executable.

To build the ZMQ proxy run the following commands in the matrix board application source directory.

```
cd zmqproxy
mkdir build && cd build
cmake ..
make
```

These commands will compile the source code and generate an executable for the proxy.

## Execution

---

**Todo:** Fix *MQTT, DDS, ZMQ Matrix board application in C++*.

The matrix board application sends out messages each 24 milliseconds, and is also receiving messages from two other boards at this pace. Each time a message is sent or received, the message and a timestamp are written to a file. This is done by creating a thread so that the main applications performance will be affected minimally by writing these messages. The application may crash after a while due to the high number of threads created to write measurements.

An error will occur because no further threads can be started at that point. If you think you can go ahead i wish you the best of luck!

---

The application needs to be started as super user, otherwise the starting of the posix threads will fail due to their attributes. The first parameter of the program determines which protocol to use. The value of this parameter can be either DDS, ZMQ or MQTT. The number of parameters after the first one differs per protocol. To start the application with DDS the following parameter structure needs to be followed:

```
sudo ./rt_matrixboard DDS <Matrixboard ID>
```

In practice this means that matrix board one can be started by running the following command:

```
sudo ./rt_matrixboard DDS 1
```

When starting the application with ZMQ more parameters are needed. This can be done using the following parameter structure.

```
sudo ./rt_matrixboard ZMQ <Matrixboard ID> <Server sub address> <Server pub_↵  
↵address> <Own address>
```

In practice this means that matrix board one for example, can be started by running the following command:

```
sudo ./rt_matrixboard ZMQ 1 tcp://192.168.178.32:5550 tcp://192.168.178.↵  
↵32:5551 tcp://192.168.178.39:5552
```

Besides this, for the communication to work using ZMQ the proxy should be started. The zmq proxy can be started from the build folder of the zmqproxy folder. Connect the socket according to the ports and address used for the proxy. The proxy can be started by running the following command:

```
./matrixboardproxy
```

When starting the application with MQTT more parameters are needed. This can be done using the following parameter structure.

```
sudo ./rt_matrixboard MQTT <Matrixboard ID> <Broker address> <QoS>
```

In practice this means that matrix board one for example, can be started by running the following command:

```
sudo ./rt_matrixboard MQTT 1 tcp://192.168.178.32:1883 0
```

For the MQTT implementation to work a broker must have been started. Connect the matrix board application according to the network address of the broker. When using the mosquitto broker the following terminal command can be used to start it:

```
mosquitto
```

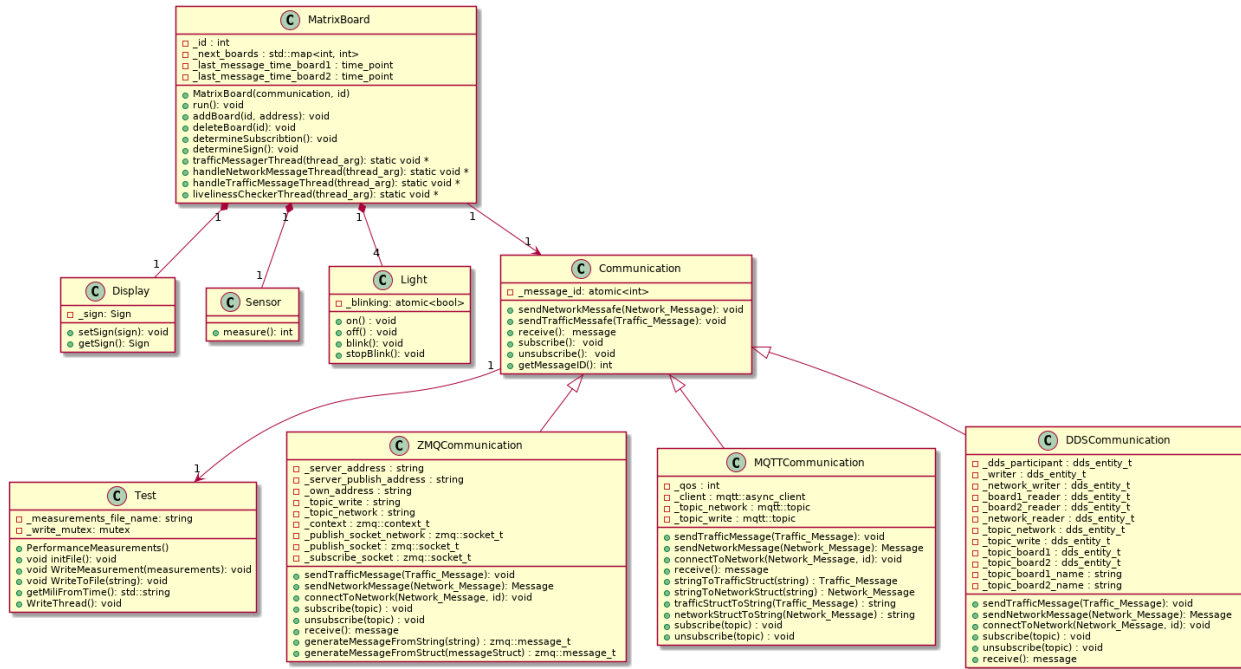
This terminal will show connecting clients. If no clients pop up when starting the matrix board application but the communication still works please check if there is another mosquitto instance running. If the clients don't pop up and the communication does not work check the address of the broker with the address parameter of the application.

A more thorough description of the parameters can be found when executing the application with a wrong number of parameters or can be read from the source of main.cpp.

## Implementation

The figure below shows the design of the application in a class diagram.

```
sudo ./rt_matrixboard DDS <Matrixboard ID>
```



The communication of the boards is done through a few topics. There is one network topic, each matrix board is subscribed to this topic. All messages concerning the connection of matrix boards are posted on this topic. This makes it possible for the boards to know which other boards are within the network so that they may subscribe accordingly. For publishing traffic intensity values each matrix board has a topic equal to its ID. To receive the traffic intensity values of a board other boards will subscribe to this topic. To receive the messages the matrix board makes use of the communication interface class. Furthermore, this is also used to subscribe and unsubscribe from topics. The protocol which the matrix board is going to use is determined before its initialization. Based on a variable passed in the application either a DDS, MQTT or a ZMQ communication object will be initialised. This will be passed to the matrix board, which now uses this protocol for its communication.

Each protocol object implements the communication interface functions. However, there is, of course, some difference amongst the implementation. For connecting to the network ZMQ connects to the proxy, this proxy is used as an intermediary which all networking messages will be posted to. This allows ZMQ to detect other endpoints in the network to which they can connect. Other than ZMQ, MQTT connects to a broker, this is also an intermediary but all the MQTT messages go to this intermediary, therefore it only has to connect to this broker. Next DDS, has endpoint discovery built-in and connects directly to its subscriptions, it does need a unique reader and writer for each topic to read or write from, these are implemented. This also means that the DDS implementation has no single point of failure as it does not depend on any intermediary. On the contrary, if the broker breaks the MQTT implementation will not be able to communicate anymore. Next, in the ZMQ implementation discovery will be impossible if the proxy breaks down. However, sockets already connected can still communicate. Another thing taken from this is the difference of connecting, while the MQTT connects to a broker, ZMQ connects through sockets and DDS connects through the protocol itself. Besides this difference, there is also a difference in the type of message sent. While DDS takes structs as messages the others do not. To keep the output of the communication the same struct is used, as DDS uses structs naturally, the used structs were the structs generated for DDS using an IDL file. This IDL file can be found in the source directory of the application. However, the other protocols implement functions to turn these structs into the types they need to send messages and the other way around for the application.

The matrix board class implements all the functionality of the matrix board application. It starts a thread to send out its traffic intensity and a thread to check the liveliness of its subscriptions. The traffic messages are sent with a minimal gap of 24 milliseconds. For the liveliness, subscriptions which the board has not received messages from for two seconds or longer are deemed disconnected. The board will notify the rest of the system of this disconnection by

sending a message to a common topic.

The main thread will handle the received messages. If a network message is received a thread will be started to handle the message, the main thread can receive a new message while the network message is still being processed. In addition, the same is true for traffic messages. The network message processing thread determines whether a subscription should be added, deleted or if there is no action to be taken based on the contents of the message and the current subscriptions. On the other hand, the traffic thread updates the traffic intensity values of the subscriptions, it also determines which sign should be shown based on the updated value. To set the displayed sign, it makes use of the display object. If there is a board to display, this will be set and the lights will blink, this is done using the light objects.

The application uses the test class to write their measurements to a file. These measurements are all timing measurements, the high-resolution clock is used to get the most precise measurements.

### PingPong

#### MQTT-PingPing

- There is only 1 program: PingPong
- You need to a MQTT broker: See <http://mosquitto.org>
- Start that first, then run PingPong
- It will print some status EVERY few messages; seconds or seconds – depending on system-speed
- Terminate with ^C

Note: on a RaspberryPi, when mosquitto is installed (apt-get) it will run a broker by default – fine

#### ZMQ-PingPong

- First start Ping, then Pong; in 2 terminals
- Optionally: specify port number (same for both)
- Stop (both) by killing (^C) them
- Currently, Ping & Pong use tcp on localhost
- TODO: use other hosts, other protocols (like ipc) and other network-patters (see ZMQ, docs)

next to the ping pong there are the roundtrip and bandwidth folder which include instructions on how to set them up

### RoundTrip

#### MQTT-RoundTrip

In the sub files programs are contained to run the round trips. Each file contains instructions to run them.

#### MQTT Round trip in C++

**authors** Sam Laan

**date** Apr 2020



## Description

The round trip C++ directory can be found in `src/demonstrators/ComparedToOtherProtocols/MQTT/RoundTrip/C++/`. This directory contains a folder for the round trip implementation as well as the flood implementation.

This page contains information about the round trip and flood applications made in C++ using MQTT.

## Dependencies

The dependencies necessary for successful compilation of the round trip / flood application in C++ are the following:

- CMake
- C++17 compiler (GCC 7+ for example)
- paho.mqtt.cpp library (installation: [MQTT C++](#))
- MQTT server compatible with MQTT 5 (mosquitto (installation: [MQTT C++](#)) for example)

## Round trip

This chapter contains information about the round trip application. This application can be found in the directory: `src/demonstrators/ComparedToOtherProtocols/MQTT/RoundTrip/C++/RoundTrip`.

## Configuring

The CMake file can be found in the source directory of the round trip. This file can be configured to either build the implementation using polling for the round trip or the one using a callback. To build the polling implementation make the following change to the Cmake file.

```
set (PROJECT roundtrip_read)
```

The line to change can be found at the top of the Cmake file. To build the callback implementation set this line to:

```
set (PROJECT roundtrip_callback)
```

## Building

To build the application run the following commands in the round trip implementation source directory.

```
mkdir build && cd build
cmake ..
make
```

These commands will compile the source code and generate an executable.

## Execution

The application takes several parameters for its execution. It can be executed using the following command in the build folder:

```
./RoundTrip <device ID> <number of devices> <total round trips> <QoS>
```

A more thorough description of the parameters can be found when executing the application without parameters or can be read from the source of main.cpp. The following example starts 4 nodes for the round trip using MQTT QoS level 1. The slaves of the round trip are started in the background. Only the master is started in the foreground in this example. Each device pings a total of 1000 times. Therefore, there are 1000 round trips.

```
./RoundTrip 2 4 1000 1 & ./RoundTrip 3 4 1000 1 & ./RoundTrip 4 4 1000 1 &  
./RoundTrip 1 4 1000 1
```

---

**Note:** Execution of the program

The round trip is initiated by the device with <device ID> = 1. Therefore, <device ID> = 1 should always be started last.

The devices added with <device ID> must be an increment of the previous one. This function does not dynamically search for the next device! So if there are four devices, device 2 should be started first, then 3 after that 4 and as last device 1 should be started as 1 should always be started last.

The <total round trips> parameter should be the same for each application.

---

## Implementation

Each round trip application creates an MQTT client. The client publishes to the topic of the ID above it. So the application with ID = 1 publishes to the application with ID = 2 and so on. The client reads from its topic. These topics have the name “roundtrip” with the ID behind it. So the topic with ID = 1 is “roundtrip1”.

The application with ID = 1 initiates the round trip. Therefore, it starts with publishing to topic “roundtrip2”. The application with ID = 2 then receives the message of the application with ID = 1 and sends a message to the next application. The last application always publishes to the first making the round trip complete.

## Read

The read implementation continuously executes the `try_consume_message` function to get the latest message of its own topic. `try_consume_message` tries to read the next message from the queue without blocking. When a message is received, the application uses `publish` to publish to the next application.

## Callback

The callback implementation uses a callback function for receiving the latest message. This callback sets a message flag which will trigger the publishing of a message to the next application.

## Flood

This chapter contains information about the flood application. This application can be found in the directory: `src/demonstrators/ComparedToOtherProtocols/MQTT/RoundTrip/C++/Flood`.

## Building

First, go to the flood C++ directory (see [Flood](#)).

Execute the following commands:

```
mkdir build && cd build
cmake ..
make
```

These commands will compile the source code and generate an executable.

## Execution

The application takes several parameters for its execution. It can be executed using the following command in the build folder:

```
./Flood <device ID> <number of devices> <total messages> <QoS>
```

A more thorough description of the parameters can be found when executing the application without parameters or can be read from the source of main.cpp. The following example starts 4 nodes for the flood using MQTT QoS level 1. The slaves of the round trip are started in the background. There are a total of 1000 messages sent by the master. For a flood to be finished, these messages should all be correctly received by the master.

```
./Flood 2 4 1000 1 & ./Flood 3 4 1000 1 & ./Flood 4 4 1000 1 &
./Flood 1 4 1000 1
```

---

**Note:** Execution of the program

The flood is initiated by the device with <device ID> = 1. Therefore, <device ID> = 1 should always be started last.

The devices added with <device ID> must be an increment of the previous one. This function does not dynamically search for the next device! So if there are four devices, device 2 should be started first, then 3, after that 4 and as last device 1 should be started as 1 should always be started last.

The <total messages> parameter should be the same for each application.

---

## Implementation

Each flood application creates an MQTT client. The client publishes to the topic of the ID above it. So the application with ID = 1 publishes to the application with ID = 2 and so on. The client reads from its topic. These topics have the name “flood” with the ID behind it. So the topic with ID = 1 is “flood1”.

The application with ID = 1 initiates the flood. It does this by publishing to topic “flood2”. The application with ID = 2 then receives the messages of the application with ID = 1 and sends a message to the next application for each message received.

The slaves (ID not equal to 1) just send a message to the next device in the flood loop as soon as they receive a message. The master (ID = 1) sends messages as fast as possible to the next device in the flood loop.

If the master sends a message, a value is incremented to keep track of the number of messages sent. The master keeps sending messages until this value is equal to the <total messages> parameter that the user inserted when executing the application. If the master receives a message, another value is incremented. In the end, this value should also be the same as the <total messages> parameter that the user inserted when executing the application.

### Slave

The slave implementation uses a callback which sets a message flag on receiving a message. these messages are received on its topic. When a message is received, the application uses `publish()` for publishing a message to the next application.

### Master

The master implementation uses a callback which sets a message flag on receiving a message. these messages are received on its topic. Afterwards, it writes a new message to the next device in the flood loop.

Therefore, in the beginning, the master only writes to the next devices. Afterwards, the loop will eventually be in a stable condition where the master reads a message first and writes a new message afterwards. In the end, the master won't send messages anymore, and will only read the last messages that exist in the flood loop.

### MQTT-IPC

run the `run.sh` file and it will set up a local round trip

requirements: mqtt must be installed then this script can run locally

### MQTT-ISC

Running the `run.sh` shell script will setup a roundtrip over four different systems.

The following components must be in place in order for it to work: three systems(raspberries) with python paho mqtt installed. one system with mosquitto and python paho mqtt installed The ips of the external systems must be 192.168.0.200, 192.168.0.201, 192.168.0.203 `run.sh` must be run on a system with ip 192.168.0.202

### ZMQ-RoundTrip

In the sub files programs are contained to run the round trips. Each file contains instructions to run them.

### ZMQ Round trip in C++

**authors** Sam Laan

**date** Jun 2020

### Description

The round trip C++ directory can be found in `src/demonstrators/ComparedToOtherProtocols/ZMQ/RoundTrip/C++/`. This directory contains a folder for the round trip implementation as well as the flood implementation.

This page contains information about the round trip and flood applications made in C++ using ZMQ.

## Dependencies

The dependencies necessary for successful compilation of the round trip / flood application in C++ are the following:

- CMake
- C++17 compiler (GCC 7+ for example)
- cppzmq library (installation: [ZMQ C++](#))

## Round trip

This chapter contains information about the round trip application. This application can be found in the directory: `src/demonstrators/ComparedToOtherProtocols/ZMQ/RoundTrip/C++/RoundTrip`.

## Configuring

The CMake file can be found in the source directory of the round trip. This file can be configured to either build the implementation using polling for the round trip or the one using a blocking receive method. To build the blocking receive implementation make the following change to the Cmake file.

```
set (PROJECT roundtrip_read)
```

The line to change can be found at the top of the Cmake file. To build the polling implementation change the following line in the Cmake file to:

```
set (PROJECT roundtrip_polling)
```

## Building

To build the application run the following commands in the round trip implementation source directory.

```
mkdir build && cd build
cmake ..
make
```

These commands will compile the source code and generate an executable.

## Execution

The application takes several parameters for its execution. It can be executed using the following command in the build folder:

```
./RoundTrip <device ID> <Total devices> <Round trips> <Pub address> <Sub address>
```

A more thorough description of the parameters can be found when executing the application without parameters or can be read from the source of `main.cpp`. The following example starts 4 devices for the round trip using ZMQ. The slaves of the round trip are started in the background. Only the master is started in the foreground in this example. Each device pings a total of 1000 times. Therefore, there are 1000 round trips.

```
./RoundTrip 4 4 1000 tcp://127.0.0.1:5554 tcp://127.0.0.1:5553 &  
./RoundTrip 3 4 1000 tcp://127.0.0.1:5553 tcp://127.0.0.1:5552 &  
./RoundTrip 2 4 1000 tcp://127.0.0.1:5552 tcp://127.0.0.1:5551 &  
./RoundTrip 1 4 1000 tcp://127.0.0.1:5551 tcp://127.0.0.1:5554
```

---

**Note:** Execution of the program

The round trip is initiated by the device with <device ID> = 1. Therefore, <device ID> = 1 should always be started last.

The devices added with <device ID> must be an increment of the previous one. This function does not dynamically search for the next device! So if there are four devices, device 2 should be started first, then 3 after that 4 and as last device 1 should be started as 1 should always be started last.

The <Pub address> parameter contains the address to which the device publishes. The <Sub address> parameter contains the address to which the device subscribes. Each device should subscribe to the subsequent device. This means that device 2 should subscribe to the publish address of device 1. Device 1 should subscribe to the publish address of the last device to complete the circle.

The <Round trips> and <Total devices> parameter should be the same for each application.

---

## Implementation

Each round trip application creates a publish and a subscribe ZMQ socket. The devices publish on their pub socket while another device subscribes to these messages by connecting to the publish socket and subscribing to the topic. So the application with ID = 1 publishes to the application with ID = 2 and so on. The client reads from its topic. These topics have the name “roundtrip” with the ID behind it. So the topic with ID = 1 is “roundtrip1”.

The application with ID = 1 initiates the round trip. Therefore, it starts with publishing to topic “roundtrip2”. The application with ID = 2 then receives the message of the application with ID = 1 and sends a message to the next application. The last application always publishes to the first, making the round trip complete.

## Read

The read implementation continuously executes the `try_consume_message` function to get the latest message of its topic. `try_consume_message` tries to read the next message from the queue without blocking. When a message is received, the application uses `publish` to publish to the next application.

## Callback

The callback implementation uses a callback function for receiving the latest message. This callback sets a message flag which will trigger the publishing of a message to the next application.

## Flood

This chapter contains information about the flood application. This application can be found in the directory: `src/demonstrators/ComparedToOtherProtocols/ZMQ/RoundTrip/C++/Flood`.

## Building

First, go to the flood C++ directory (see [Flood](#)).

Execute the following commands:

```
mkdir build && cd build
cmake ..
make
```

These commands will compile the source code and generate an executable.

## Execution

The application takes several parameters for its execution. It can be executed using the following command in the build folder:

```
./Flood <device ID> <Total devices> <Messages> <Pub address> <Sub address>
```

A more thorough description of the parameters can be found when executing the application without parameters or can be read from the source of main.cpp. The following example starts 4 devices for the flood using ZMQ. The slaves of the flood are started in the background. There are a total of 1000 messages sent by the master. For a flood to be finished, these messages should all be correctly received by the master.

```
./Flood 4 4 1000 tcp://127.0.0.1:5554 tcp://127.0.0.1:5553 &
./Flood 3 4 1000 tcp://127.0.0.1:5553 tcp://127.0.0.1:5552 &
./Flood 2 4 1000 tcp://127.0.0.1:5552 tcp://127.0.0.1:5551 &
./Flood 1 4 1000 tcp://127.0.0.1:5551 tcp://127.0.0.1:5554
```

---

**Note:** Execution of the program

The flood is initiated by the device with <device ID> = 1. Therefore, <device ID> = 1 should always be started last.

The devices added with <device ID> must be an increment of the previous one. This function does not dynamically search for the next device! So if there are four devices, device 2 should be started first, then 3, after that 4 and as last device 1 should be started as 1 should always be started last.

The <Pub address> parameter contains the address to which the device publishes. The <Sub address> parameter contains the address to which the device subscribes. Each device should subscribe to the subsequent device. This means that device 2 should subscribe to the publish address of device 1. Device 1 should subscribe to the publish address of the last device to complete the circle.

The <Messages> and <Total devices> parameter should be the same for each application.

---

## Implementation

The flood application creates a publish and a subscribe ZMQ socket. The devices publish their pub socket while another device subscribes to these messages by connecting to the publish socket and subscribing to the topic. So the application with ID = 1 publishes to the application with ID = 2 and so on. The client reads from its topic. These topics have the name “flood” with the ID behind it. So the topic with ID = 1 is “flood1”.

The application with ID = 1 initiates the flood. It does this by publishing to topic “flood2”. The application with ID = 2 then receives the messages of the application with ID = 1 and sends a message to the next application for each message received.

The slaves (ID not equal to 1) just send a message to the next device in the flood loop as soon as they receive a message. The master (ID = 1) sends messages as fast as possible to the next device in the flood loop.

If the master sends a message, a value is incremented to keep track of the number of messages sent. The master keeps sending messages until this value is equal to the `<total messages>` parameter that the user inserted when executing the application. If the master receives a message, another value is incremented. In the end, this value should also be the same as the `<total messages>` parameter that the user inserted when executing the application.

### Slave

The slave implementation uses a callback which sets a message flag on receiving a message. these messages are received on its topic. When a message is received, the application uses `publish()` for publishing a message to the next application.

### Master

The master implementation uses a callback which sets a message flag on receiving a message. these messages are received on its topic. Afterwards, it writes a new message to the next device in the flood loop.

Therefore, in the beginning, the master only writes to the next devices. Afterwards, the loop will eventually be in a stable condition where the master reads a message first and writes a new message afterwards. In the end, the master won't send messages anymore, and will only read the last messages that exist in the flood loop.

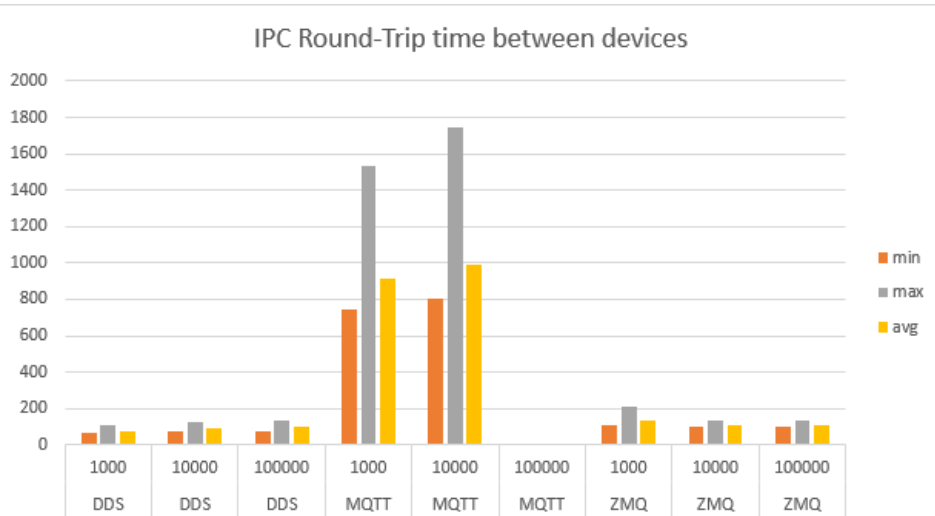
### Findings Round-trip C++

The Round-trip applications for ZMQ and MQTT write their results to a CSV file called `roundtrip.csv`. This CSV file can be used to process the results of experiments done using the application. There were already two experiments done concerning inter-process communication(IPC) and inter-device communication(IDC). The results of these experiments are described below.

#### IPC experiment

The IPC experiment was conducted on the Lenovo Thinkpad T470 20HD000MUK. Three nodes were used to conduct the experiment. The following graph shows the results of the experiment for each protocol for a different number of messages. This was done to put them in direct comparison. No quality of service policies were used, for MQTT this means that level 0 was used.

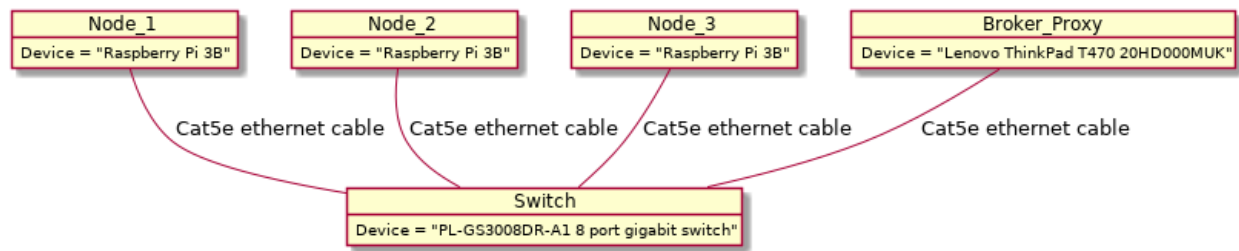




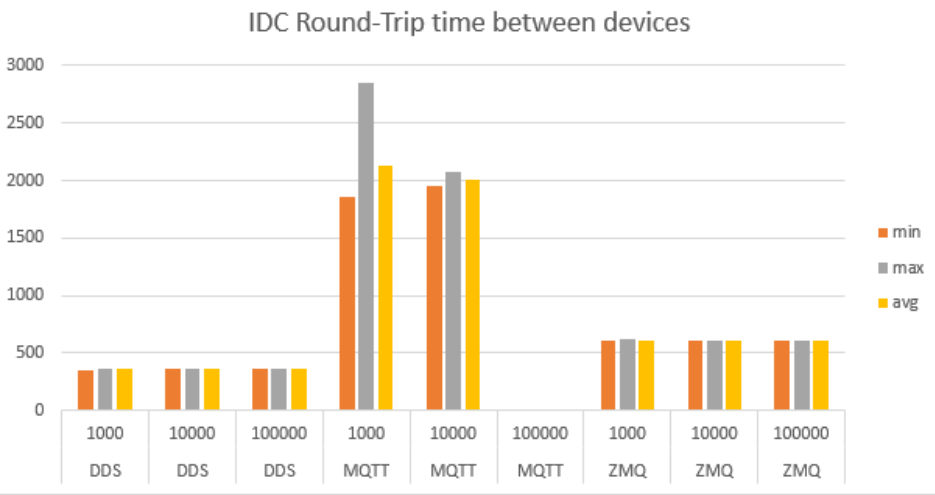
The value on the X-axis is time in microseconds. The values on the Y-axis is the protocol and the number of messages. MQTT gets outperformed by each other, by a factor of about 5 with DDS and a factor of 3 compared to ZMQ. Besides this MQTT also suffered from package loss when sending 100000 messages leaving the roundtrip unable to finish.

IDC experiment

The following setup was used for the IDC experiment:



Three nodes were used to conduct the experiment. The following graph shows the results of the experiment for each protocol for a different number of messages. This was done to put them in direct comparison. No quality of standard policies were used, for MQTT this means that level 0 was used.



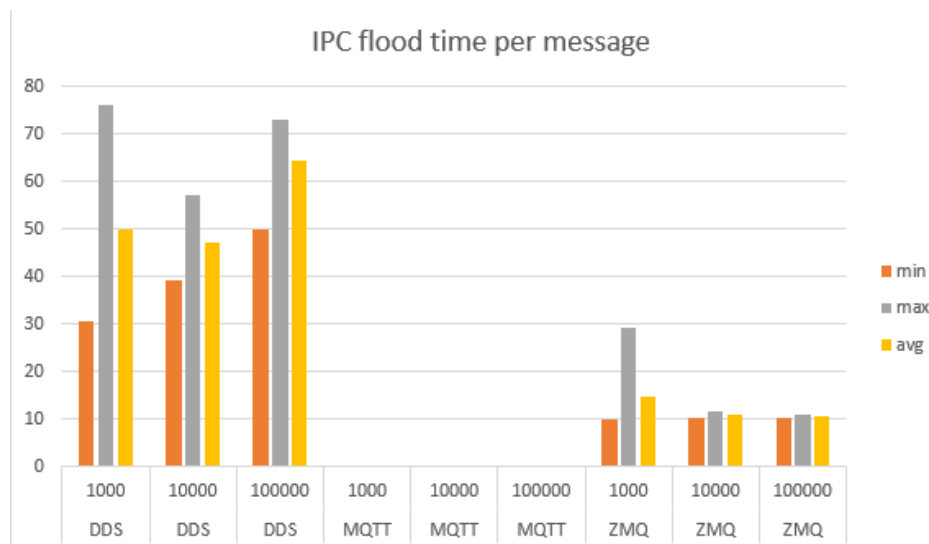
The value on the X-axis is time in microseconds. The values on the Y-axis is the protocol and the number of messages. The graph shows that the amount of messages has a minimal amount of influence on performance for each protocol. Meaning each protocol is stable in the speed of sending and receiving messages. However, MQTT is the least stable of the protocols when looking at the graph. Besides this, MQTT gets outperformed by the same factors as the IPC experiment. Again MQTT suffered from package loss at 100000 messages leaving the application unable to finish.

### Findings Flood C++

The Flood applications write their results to a CSV file called flood.csv. This CSV file can be used to process the results of experiments done using the application. There were already two experiments done concerning inter-process communication(IPC) and inter-device communication(IDC). The results of these experiments are described below.

### IPC experiment

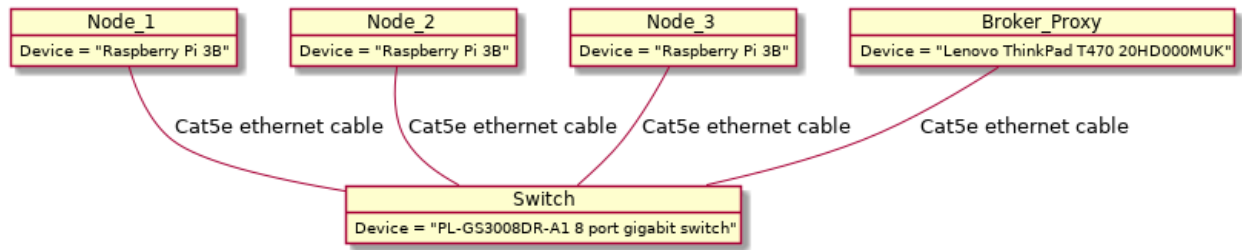
The IPC experiment was conducted on the Lenovo Thinkpad T470 20HD000MUK. Three nodes were used to conduct the experiment. The following graph shows the results of the experiment for each protocol for a different number of messages. This was done to put them in direct comparison. No quality of standard policies were used, for MQTT this means that level 0 was used.



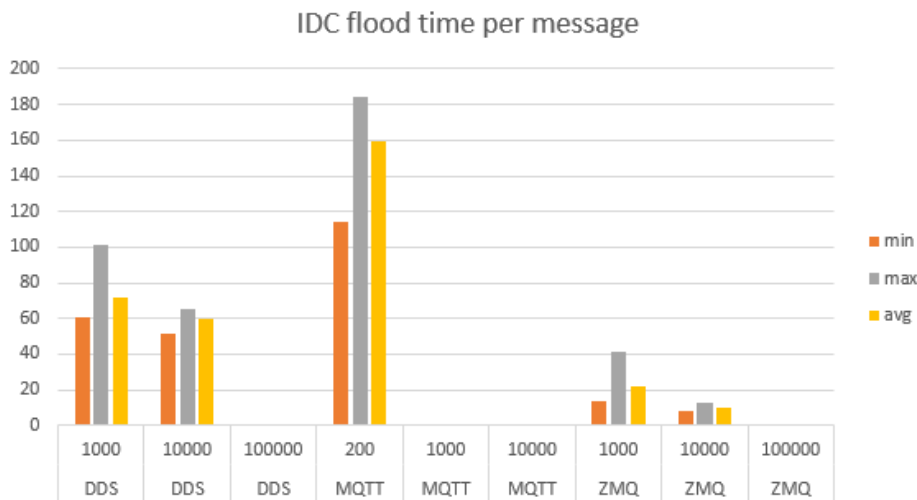
The value on the X-axis is time in microseconds. The values on the Y-axis is the protocol and the number of messages. MQTT would get a buffer overflow when sending messages in the flood experiment. Therefore no results could be extracted. The application would work to a maximum of about 150 to 200 messages. A greater number will trigger the overflow error and crash the application. For the other protocols it can be seen that ZMQ outperforms DDS. While in the round-trip it was the other way around. This probably has something to do with the fact that ZMQ uses a message queue while DDS uses a different mechanism. The message queue principle is faster for the flood case while in the round-trip it's the other way around.

### IDC experiment

The following setup was used for the IDC experiment:



Three nodes were used to conduct the experiment. The following graph shows the results of the experiment for each protocol for a different number of messages. This was done to put them in direct comparison. No quality of service policies were used, for MQTT this means that level 0 was used.



The value on the X-axis is time in microseconds. The values on the Y-axis is the protocol and the number of messages. MQTT would get a buffer overflow when sending messages in the flood experiment. Therefore no results could be extracted. The application would work to a maximum of about 150 to 200 messages. A greater number will trigger the overflow error and crash the application. The graph shows that MQTT is far slower than DDS and ZMQ. The maximum time of a message was 186 microseconds while DDS has a maximum time of 100 microseconds and ZMQ of 41. For some amount of messages no results could be gathered. This is due to package loss leaving the application unable to finish.

### Cyclone DDS Python binding

This page describes the Cyclone DDS Python binding, and gives a tutorial of its usage. There are currently different versions available, but only the latest version is for production, the rest is temporarily saved for development purposes.

Content:

#### Version1

**Warning:** This version is deprecated. It describes an older version of the library, which is no longer in use. This page is only kept for development purposes, and will be removed shortly.

---

**Note:** The code for this version is located in /poc1

---

### Description

Demonstrator for the new Python binding. Main features are (once finished): - Full feature set (all functionalities of the underlying DDS library available) - Able to handle idl files properly (the atolab implementation converts everything to a string, and disregards idl files). - Pythonic implementation, to ensure the easy of use for developers.

Currently there is a minimal version available. It is possible to send and receive basic messages, using the json method. IDL support will be added in the near future.

### Assumptions

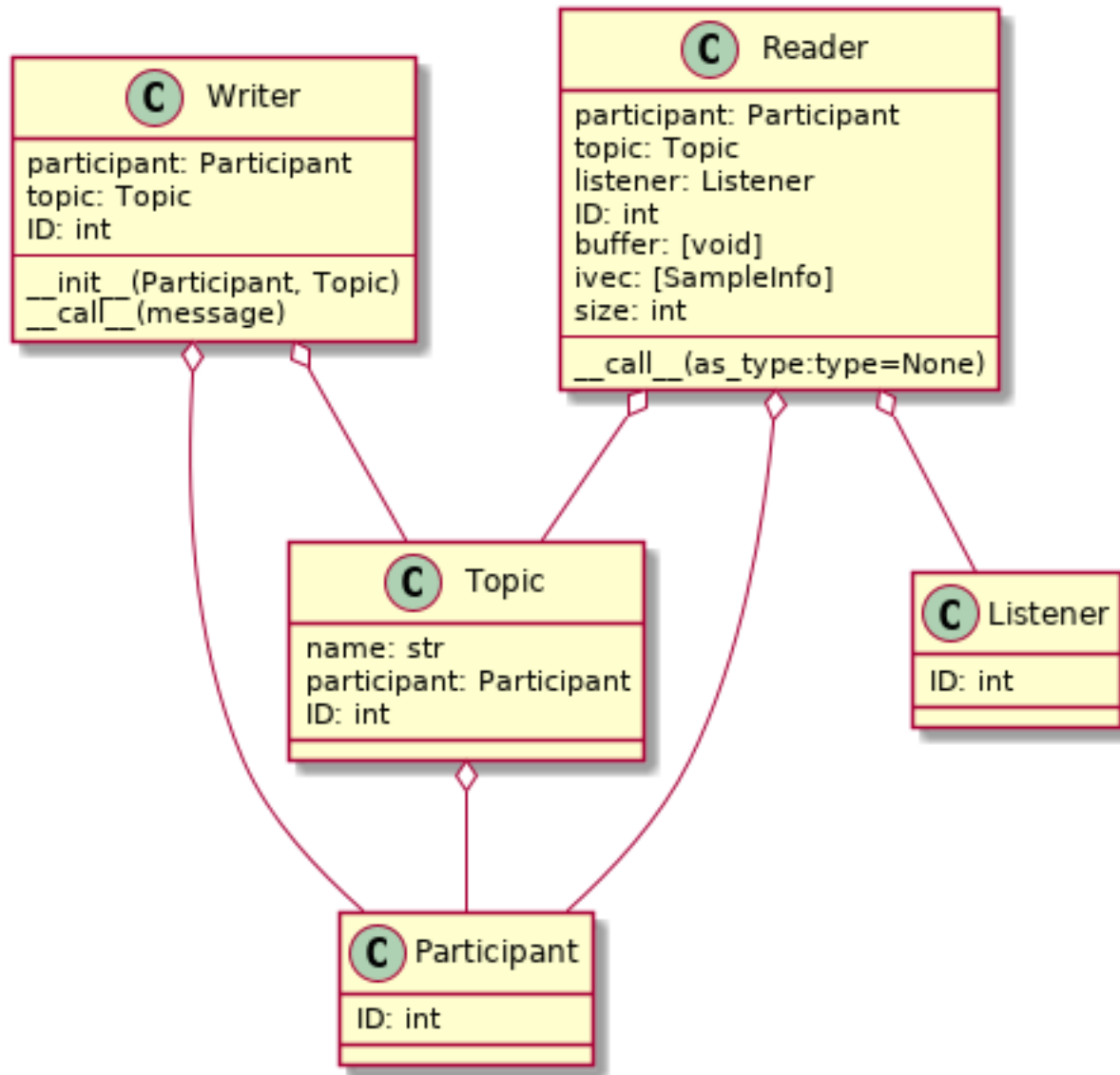
The following library files need to be in the stated directory:

- libddsc.so at “/usr/local/lib”
- libddstubs.so at “/usr/local/lib”

The latter is the one coming from the atolab implementation, and is only needed temporarily. It contains the topic descriptor for the json topic. Once IDL support is added it will be removed.

### Design

The UML of the implementation:



## Examples

The files *sender.py* and *receiver.py* demonstrate the usage of the library. Sender publishes a simple message to the topic “testtopic” every 2 seconds, and logs it to the console.

Receiver reads the same topic, once every 3 seconds, and logs the list of read messages to the console. (It always reads a list of messages, so reading more than one at a time is not a problem.)

The files can be called from the *Examples* folder as follows:

```
python3 sender.py
```

and

```
python3 receiver.py
```

### Version2

### Description

This page describes the Cyclone DDS Python binding that can be found at `src/demonstrators/PythonBinding/poc2`. This package is wrapper on Cyclone's C library, with the aim of providing access to all functionalities, with a pythonic API. It was designed to fulfill a number of requirements:

- **Completeness:** All the functionalities that are implemented within Cyclone DDS should be available from the Python version.
- **DDS wire-standard:** The current version developed by atolab uses pre-defined topics, by converting everything to a string, regardless of the actual data type of the field. This makes it impossible to communicate with different implementations. This library should obey the DDS wire-standard.
- **Pythonic:** The implementation should be done in a Pythonic way, and it should feel natural for the Python developer.

**Warning:** The library is not yet fully implemented. It does not support sequence fields, and nested topic types, and it is not possible to specify quality of service options. These will be implemented soon.

### The role of IDL

In order to understand how the Python binding deals with defining and registering topic types, we give a brief overview of how this is traditionally (both in the C version, as well as in the atolab Python version) approached.

DDS is a topic based protocol. A topic is an abstract entity, that assigns both semantic and syntactic meaning to the bytes that are being sent over the network. It ensures that the receiver can decode the message properly, and also provides a mean of error checking for the sender. There are however a number of other important attributes of a topic other than its byte layout. For example specifying quality of service options, and setting different configuration flags are topic level operations, that need to be stored in the global data space, alongside with the topic definition. There is also an important distinction between keyed and keyless topics: Having no key specified makes all the instances of a topic effectively equal (meaning there is no distinction possible between the instances). So when a field is set to be a topic key, this information needs to be known by the global data space, that stores the messages and manages queues.

To be able to do all this, it is not sufficient to simply define the structure (in the C sense) of the desired topic, but it also becomes necessary to provide some sort of configuration object, that describes the aforementioned attributes. This configuration object (which is not really an object itself, but a C-style structure) is called the topic descriptor. To make life easier for the developer, Cyclone DDS lets the user define the topic structures in a tiny domain specific language, called IDL. A special parser reads this definition, and generates the corresponding C data types for both the topic structure and the topic descriptor. These generated C files can then be used to implement the program.

### How to be Pythonic?

This approach with the IDL file works alright when working with C, or any other compiled language, because the use of an additional static tool to get your topic generated is not of huge inconvenience. When using a script language like Python, it becomes much more of a problem. It suddenly becomes a lot more annoying to let a third-party tool generate part of your code every time you change something. So it is necessary to find a solution that lets the user circumvent the use of IDL definitions.

In the atolab Python library, they decided to get rid of the possibility of defining your own topic altogether. Instead, they limited the user to a single topic (for which IDL is still used by the way), and every possible value that can be sent is reduced to this single topic type. It seems ok on the surface, as there is no need to deal with the static definition of

topic types, but there are two major problems with this approach. Firstly, to ensure that everything can be reduced to a given topic, it is chosen to have a string field that essentially contains all the data. Every time you send something over the network, the library converts the data into a json string. This is of course extremely inefficient. Not only does it cost time and computational resources to convert every single sample into a string, but it also increases the network load by a lot. For example, if you want to send the number 3658, it will not be sent as a single integer, but as four different characters. Another, more obvious problem with this approach is that it entirely disobeys the DDS wire-standard, meaning it is only able to communicate with another atolab client, but not with a different DDS implementation.

At the start of the project there were two possible options for tackling this issue: We can either implement our own IDL parser, that outputs Python run-time objects instead of .c/.h files, or that we leave out the static definition entirely, and do all things in Python. After discussing this issue with the product owner [Albert Mietus] on several occasions, we settled on the second approach, because of being more modern and Pythonic.

## Design

### Domain

Participants in DDS are all part of a **domain**. A domain roughly means that every endpoint that is on domain x can only communicate with other endpoints on domain x, and not with others. Domains are to separate different networks of devices from each other. When creating an application, first you must specify the domain of it, with a domain ID.

```
domain = Domain(1)
```

### Topic

The goal is to define topic structures as ordinary Python classes. In Python, everything is dynamically typed, which means that when a class is defined, it is not clear what the types of its attributes are going to be (and those types do also not necessarily remain the same throughout the execution of the program). This behaviour makes it impossible to generate the proper topic descriptor when creating a new topic. To make it possible we need to introduce a way of fixing the type of the attributes when defining a class. One way of doing this is having a static dictionary, or list of tuples that maps each attribute to their corresponding type. Then we would get something like this:

```
class Data:
    _fields_ = [
        "id": int,
        "name": str
    ]

    def __init__(self):
        self.id = None
        self.name = None
```

It describes the topic `_Data_`, that has two attributes, `id` and `name`. This is however not the most elegant solution, as the user has to add a class level attribute, `fields`, which makes the code less readable, and annoying to write it out if there are many attributes. It is also inconvenient having to initialize them, as their value will be determined run-time. (If you could initialize any of your fields as a constant, it would make no sense to send it over the network, as you could initialize it at the other side to the same constant as well.) Since everything is an object in Python, even types, we chose for a solution that mitigates both issues, and feels a lot more Pythonic. You would then define the class above like this:

```
class Data:
    def __init__(self):
```

(continues on next page)

(continued from previous page)

```
self.id    = int
self.name  = str
```

To turn this into a proper topic, the C struct of this class, and its topic descriptor needs to be generated and passed to DDS. This work is done in the topic class decorator.

---

**Note:** The `_topic_` decorator makes run-time changes on the class definition, adds new attributes to it, and most importantly replaces the `__init__` function. It is important that the original `__init__` **does not** have any arguments other than `self`, and optionally `**kwargs`.

---

As an extra argument, the decorator also takes the domain on which this topic will run.

```
domain = Domain(1)    # '1' is the domain id.

@topic(domain)
class Data:
    def __init__(self):
        self.id    = int
        self.name  = str
```

The topic “Data” is now properly registered by Cyclone, and instances of it can now be sent over the network. It can be instantiated by supplying it with keyword arguments for the attributes.

```
data = Data(name="Kevin", id=27)
```

It is not necessary to initialize all of the fields. There is a default value associated with each type. The uninitialized fields take the default value.

```
>>> data = Data(name="Jan")
>>> data.id
>>> 0
```

It is also possible to be more specific with typing. There are a number of C types included in the library, such as `int16`, `int32`, `uint16` etc.

## Sending and receiving

You can publish using a `Writer` object. The constructor of writer takes a topic that it can write to, and can be used by calling its `__call__` method with the data sample. Below is a minimalistic example of a program that writes a single data point:

```
domain = Domain(1)    # '1' is the domain id.

@topic(domain)
class Data:
    def __init__(self):
        self.id    = int
        self.name  = str

w = Writer(Data)      # Creating writer object.

sample = Data(name="Jaap", id=12)
w(sample)             # Writing the sample data
```



Reading from a topic can be done with the Reader class. Similarly to Writer, when creating a Reader you need to pass it the topic. Additionally you also need to specify the buffer size. It is the number of elements that can be stored unread. So it is the size of the queue for **unread** messages. A call to Reader.\_\_call\_\_ returns a **list** of data points, flushing the buffer. Here is an example of the “other half” of our program:

```
domain = Domain(1)           # '1' is the domain id.

@topic(domain)
class Data:
    def __init__(self):
        self.id = int
        self.name = str

r = Reader(Data, 10)         # Creating reader object.
samples = r()                # samples now contains a (possibly empty) list of Data_
                             ↪ objects.
```

### 1.2.5 Team pages

**author** Albert

**date** Dec 2019

#### Concept

Each team (possible of 1 person) gets an *own* directory (a subdir of this one), to place notes, store know-how, etc. That does not imply they are the only ones who may change those articles; it just a convenient place to start.

So whenever you find out something, that may be interesting for your team, or a future team: write a note, an article, and place it in your team-section. Or update an existing one (even of another team). In all cases, add you name as author or section-author and update the date

This part consists of a lot of ‘notes’. They may be new, maintained, or outdated. Moreover, general quality rules are lowered. This is **on purpose**: it is better to have and share a brainwave, then to have it forgotten. As long a the reader knows it status: by this intro, everybody should know!

#### Talk like pi

Welcome to the ‘talk like pi’ team page. To start right away i would recommend going through the setup chapter first so that all requirements are in place to run everything. Afterwards go to the running part of each of the protocol/library chapters and it will show how to run the programs.



#### contents

#### setup

To make the round trip and bandwidth applications run some sort of setup is needed. There are different ways to make this setup like using virtual

machines. Because there are four raspberries on hand and its more fun they are used to



The image above shows how i do it. four raspberries connected to a router. The DHCP setting can easily be adjusted to give the raspberries static IP's. The following ip addresses are set for the raspberries:

- 192.168.0.200
- 192.168.0.201
- 192.168.0.202
- 192.168.0.203

The raspberry i work from is 202, theres no real reason for it just that its the raspberry that i used first. However it is important to also work from this pi, because the shell scripts that have been made assume this setup and that the pi

from which it is started has ip 202. For these scripts to work each of the raspberries needs a ssh-key without password from the 202 raspberry. Though this is only necessary to use the isc programs.

It is also important to have DDS, MQTT and ZMQ installed on all the raspberries, the repository only needs to be cloned to one of them. How to install all three libraries is shown on the main page of this documentation under IPSC Inter Process & System Communications.

## MQTT

MQTT(Message Queuing Telemetry Transport) uses a pub-sub method to get messages from client to client, more on that in the chapter MQTT workings.

### Running MQTT

before running anything the following ASSUMPTIONS are made here:

- MQTT is installed as per the IPSC manual
- you have four raspberries configured as per the setup chapter
- the repository has been downloaded

Now setting up the first MQTT roundtrip is very easy:

```
sh <techpush-sourcedir>/demonstrators/ComparedToOtherProtocols/MQTT/RoundTrip/Python/
↪ipc/run.sh
```

now the local roundtrip is being setup up and run. Be sure to run cleanup.sh after cancelling the script. note: Some of the shell scripts automatically run cleanup.sh when ctrl-c is pressed

running the other MQTT setups is just as easy. For example the isc version of the roundtrpp:

```
sh <techpush-sourcedir>/demonstrators/ComparedToOtherProtocols/MQTT/RoundTrip/Python/
↪isc/run.sh
```

or for both the bandwidth programs:

```
sh <techpush-sourcedir>/demonstrators/ComparedToOtherProtocols/MQTT/Bandwidth/Python/
↪ipc/run.sh
sh <techpush-sourcedir>/demonstrators/ComparedToOtherProtocols/MQTT/Bandwidth/Python/
↪isc/run.sh
```

The MQTT.py program takes command line inputs in the following order:

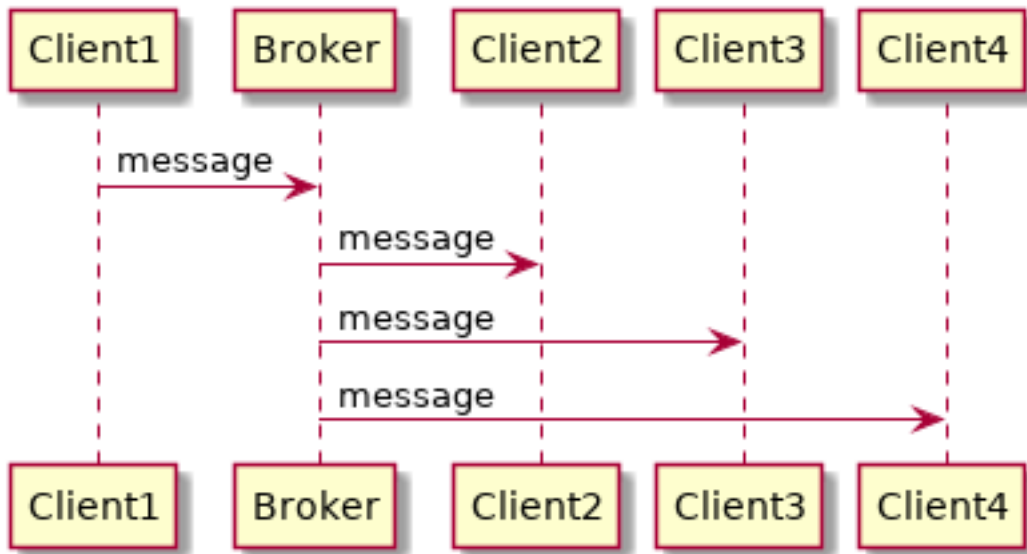
Receiving topic Sending topic MQTT broker IP a 1 or 0 for being a master or hub respectively QOS quality of service so for example:

```
python3 MQTT.py ReceivingTopic SendingTopic 192.168.0.202 1 0
```

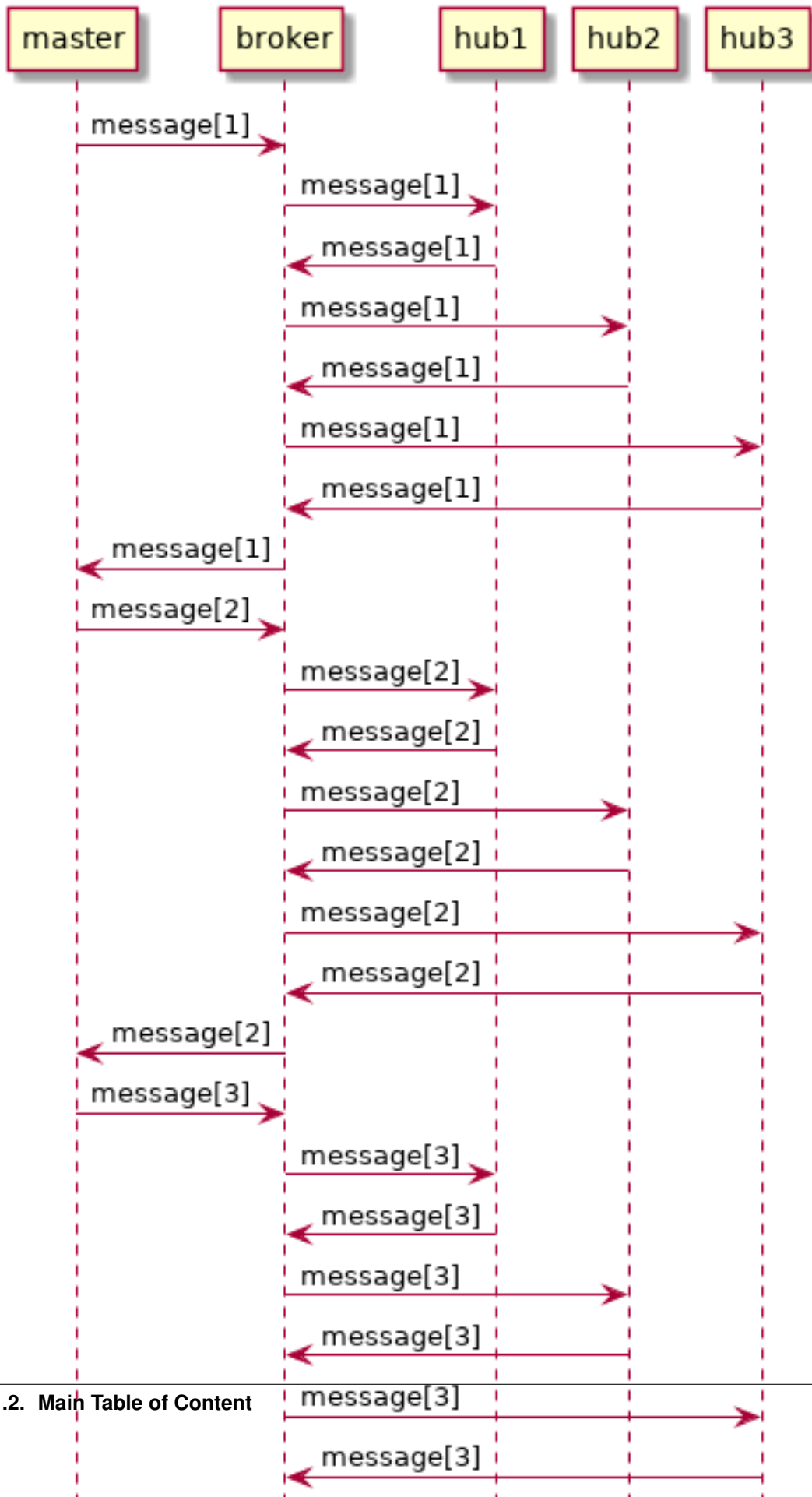
In the example a master(1) is started which sends to SendTopic, receives on ReceiveTopic, works with a broker on 192.168.0.202 and uses a QOS of 0.

### MQTT workings

MQTT works by sending a message to a central broker which then distributes the message to everybody that's listening in. It looks something like:



In the context of the roundtrip this looks like:



In the context of a roundtrip this method might seem a bit inefficient as every message has to go through the broker. Also there is a single point of failure. However if the message needs to go to multiple hubs it costs almost nothing extra for the whole system. Speed should remain the same. This can be useful if for example there is a system where one sensor input needs to go to several different actuators. There are many more downsides and upsides however I will not discuss them here, there are many websites that discuss them like this one: <http://portals.omg.org/dds/features-benefits/>

## ZMQ: ZeroMQ

### Running ZMQ

before running anything the following ASSUMPTIONS are made here:

- ZMQ is installed as per the IPSC manual
- you have four raspberries configured as per the setup chapter
- the repository has been downloaded

<ZMQ roundtrip explanation of how to use them>

For ZMQ the round trip is not as nicely worked out as for MQTT, however the bandwidth programs are easy to start:

```
sh <techpush-sourcedir>/demonstrators/ComparedToOtherProtocols/ZMQ/Bandwidth/Python/  
↪isc/run.sh  
sh <techpush-sourcedir>/demonstrators/ComparedToOtherProtocols/ZMQ/Bandwidth/Python/  
↪ipc/run.sh
```

### About ZMQ

ZMQ is a very versatile library that makes implementing different ways of sending messages possible with one library and still tries to keep it simple. There are four main ways of communicating messages:

- Request-reply: client sends a request to a service and gets information back.
- Publish-subscribe : client publishes message to several other clients whom are listening
- Pipeline
- Exclusive pair

These methods of handling messages can be connected in the following ways:

- PUB and SUB
- REQ and REP
- REQ and ROUTER (take care, REQ inserts an extra null frame)
- DEALER and REP (take care, REP assumes a null frame)
- DEALER and ROUTER
- DEALER and DEALER
- ROUTER and ROUTER
- PUSH and PULL
- PAIR and PAIR

## DDS

DDS(Dynamic Data Distribution)

### Running DDS

Like for MQTT and ZMQ, DDS also needs some preparations to use:

- ZMQ is installed as per the IPSC manual
- you have four raspberries configured as per the setup chapter
- the repository has been downloaded

It is possible to make a round trip using the python scripts. This can be done like this:

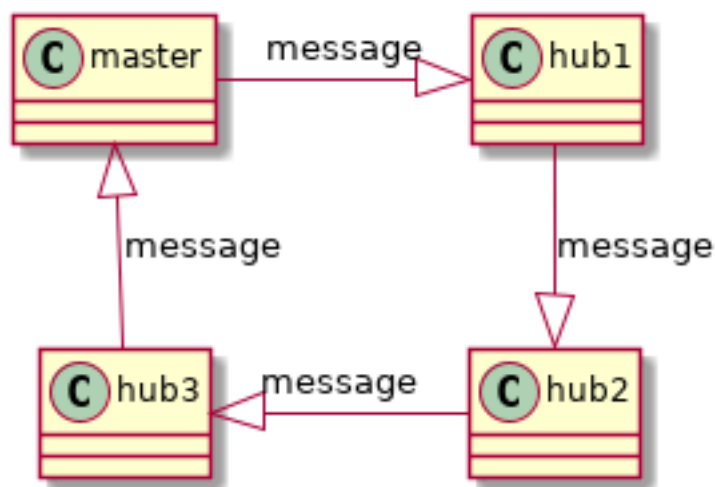
```
python3 <techpush-sourcedir>/demonstrators/RoundTrip/Python/DDShub.py Master_Out hub1
python3 <techpush-sourcedir>/demonstrators/RoundTrip/Python/DDShub.py hub1 hub2
python3 <techpush-sourcedir>/demonstrators/RoundTrip/Python/DDShub.py hub2 hub3
python3 <techpush-sourcedir>/demonstrators/RoundTrip/Python/DDShub.py hub3 Master_in
python3 <techpush-sourcedir>/demonstrators/RoundTrip/Python/RoundTrip.py
```

These commands can be run on the same or separate systems and they still work. DDS will work out how to connect them, of course the systems do have to be on the same network.

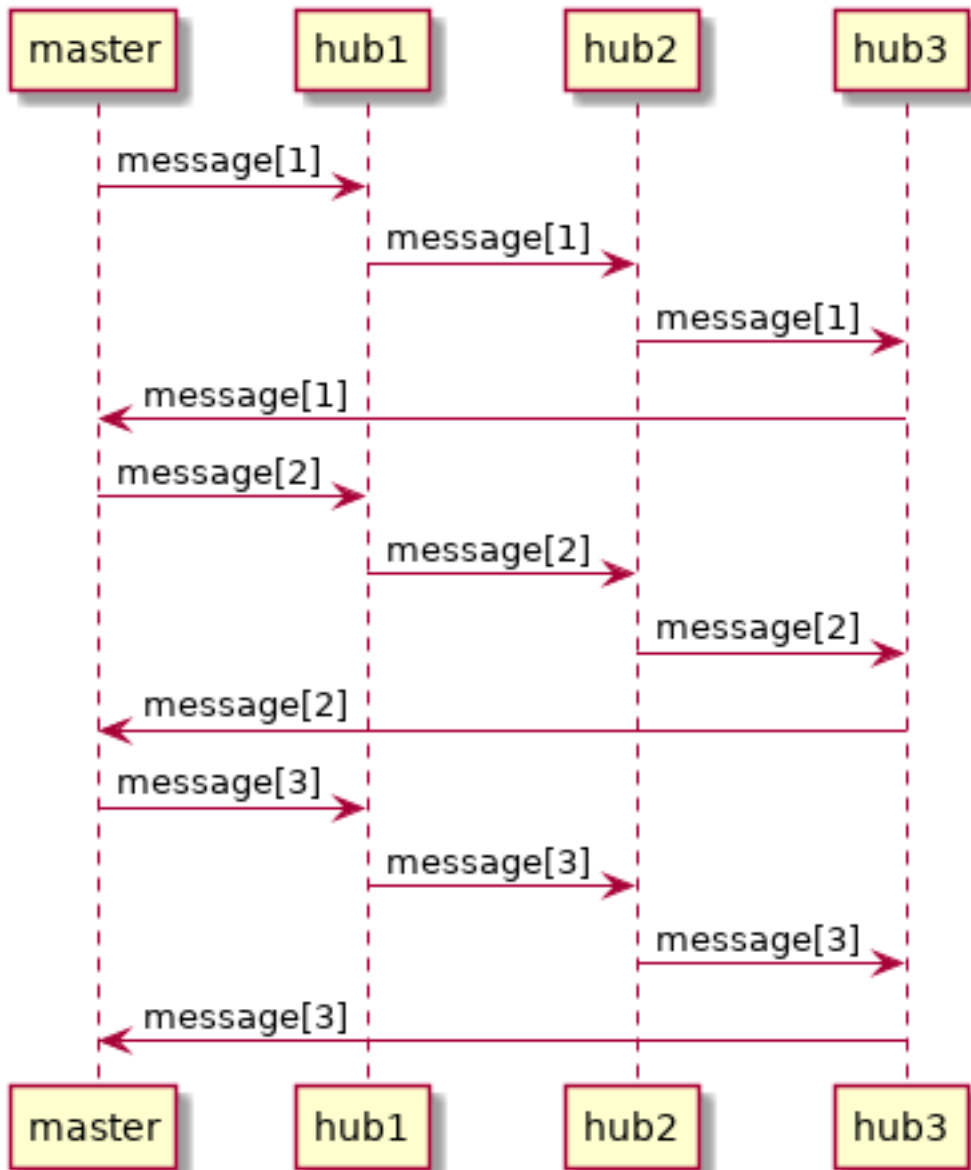
In short what happens here is that first three DDS-hubs are started. the first argument given is the topic the hub is listening to for incoming messages, the second input is the topic to which the received message will be sent. The master functions basically the same way as the hub with only two differences: it adds 1 to the message each time it passes and some time after it started it sends the first message.

### RoundTrip

To test the different ways of communication, what is called a roundtrip is used. In a round trip four different processes or systems send messages to one another. In the round trip there is one master and three hubs it looks a little like this:

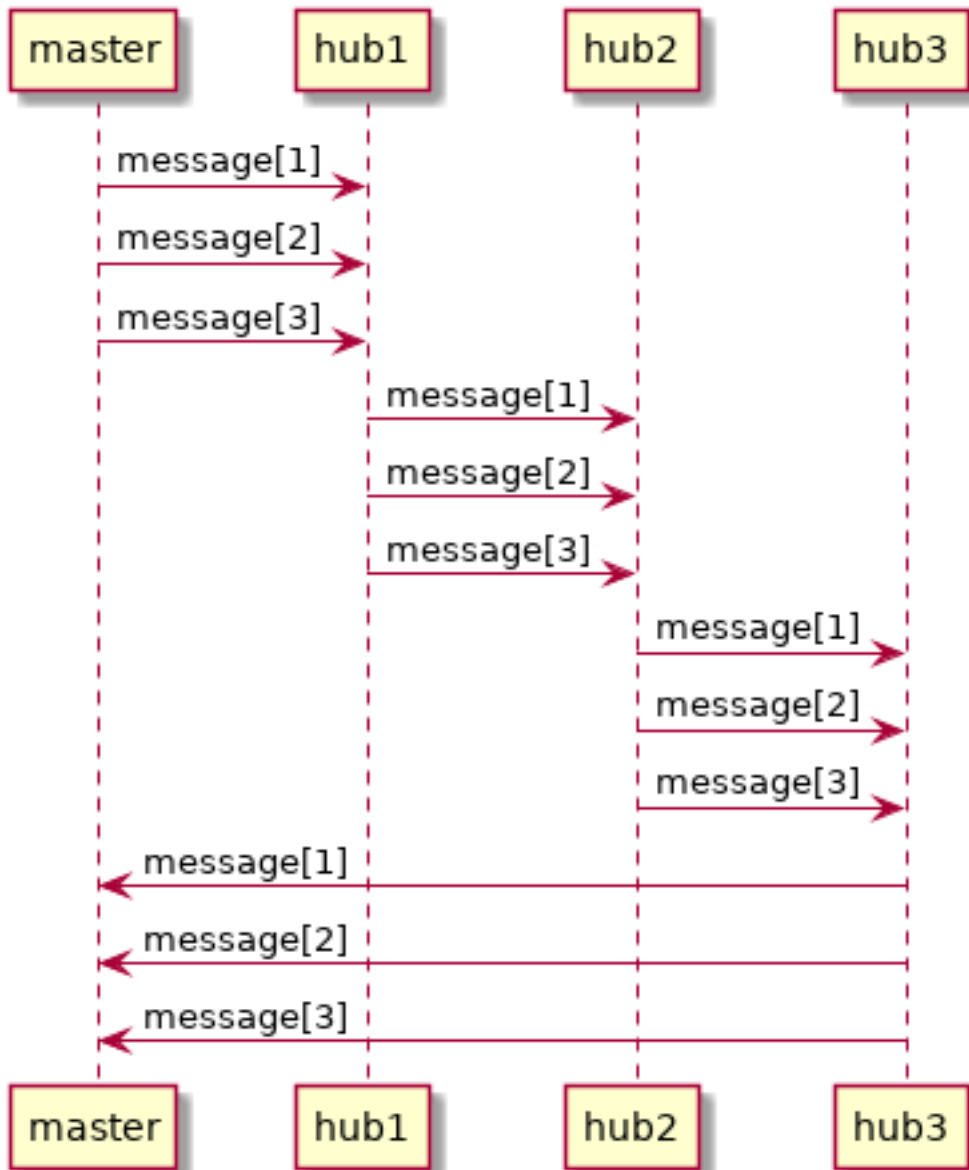


In a time diagram it looks a little like this:



In the example above the message makes three roundtrips. Of course in practice this will be thousands of round trips to accurately determine the amount of time it takes to go around. As shown every time the message passes the master its contents is increased by one. This way the master can calculate how many times a message passes every second or how long it takes for the message to take a round trip. Another way of measuring the methods of communication is sending a bunch of messages and see how long it takes for them to come back. That looks something like:





Both of these tests are interesting as the first test might give an idea whether the communication is fast enough for example in a controlled system. The second is interesting as in practice there may be many systems communicating many different things and this simulates that situation.

## Bandwidth

### Sending a bunch of messages and time how long it takes for them to come back

similar to the Roundtrip there is a different way of testing the speed of a connection. This also results in a time spent per message. In this method the master sends a whole bunch of messages and then waits until they all come back. Dividing the time it took of the messages to come back by the amount of messages also gives an amount of time.

## Tips and tricks

### Sphinx continuous build

Instead of writing in sphinx pressing build to see if it comes out correctly its much more efficient to make a script that continuously builds so you just have to push refresh in your browser.

in windows this can be done with:

```
:loop

DEL /q <buildDir>\*

FOR /L %%A IN (1,1,20) DO (
sphinx-build -b html <techpushDir>\mess.techpush-training\docs\doc <buildDir>
)

goto loop
```

This will keep building sphinx forever and once every 20 builds it does a clean build.

### Use shell scripts

They might not be the most intuitive thing to work with but shell scripts are very usefull. Take for example the shell script used for the isc version of ZMQ bandwidth:

```
BASEPORT="888"
HUB="0"
MASTER="1"

BASESEND1="tcp://"
BASESEND2=":8881"
RECEIVEURL="tcp://*:8881"

Pi0="192.168.0.200"
Pi1="192.168.0.201"
Pi2="192.168.0.202"
Pi3="192.168.0.203"

# copy MQTT.py the raspberries
scp ZMQbisc.py pi@${Pi0}:~/ZMQbisc.py
scp ZMQbisc.py pi@${Pi1}:~/ZMQbisc.py
scp ZMQbisc.py pi@${Pi3}:~/ZMQbisc.py

ssh pi@${Pi0} 'mkdir /tmp/feeds'
ssh pi@${Pi1} 'mkdir /tmp/feeds'
ssh pi@${Pi3} 'mkdir /tmp/feeds'

SOCKETLOCATION="ipc:///tmp/feeds"

mkdir /tmp/feeds

trap ctrl_c INT

ctrl_c() {
    echo "knock knock, cleaning lady"
```

(continues on next page)

(continued from previous page)

```

    sh cleanup.sh
}

ssh pi@${Pi0} python3 ~/ZMQbisc.py ${RECEIVEURL} ${BASESEND1}${Pi1}${BASESEND2} 0 &
sleep 1
ssh pi@${Pi1} python3 ~/ZMQbisc.py ${RECEIVEURL} ${BASESEND1}${Pi3}${BASESEND2} 0 &
sleep 1
ssh pi@${Pi3} python3 ~/ZMQbisc.py ${RECEIVEURL} ${BASESEND1}${Pi2}${BASESEND2} 0 &

python3 ZMQbisc.py ${RECEIVEURL} ${BASESEND1}${Pi0}${BASESEND2} 1

sh cleanup.sh

```

This script makes it easy to start the round trip even though it has scripts running on four different raspberry pi's. Also the cleanup script makes sure that everything the shell script makes is removed when it stops. This makes sure that there is no python program running in the background messing things up. Doing this all manually might not seem like much work but doing it a hundred times on one day is.

Conclusion: USE SHELL SCRIPTS

## sprints

### sprint1

#### sprint1 results

In this sprint the amount it takes for a message to make a round trip has been measured. To understand what the times mean in the tables, see the theory about roundtrip in [RoundTrip](#) The measurements using the round trip programs have given the following results:

IPC measurements	
Language: python3	
protocol	ms/msg
MQTT	4.694029412
ZMQ	0.540805556
DDS	under construction

ms/msg represents the amount of time it takes for the message to make a round trip.

ISC measurements	
Language: python3	
protocol	ms/msg
MQTT	6.052571429
ZMQ	1.617088235
DDS	under construction

### sprint2

The first sprint has passed its time for the second sprint.

## sprint results 1

Using the method of sending a whole bunch of message at the same time and timing how long it takes before they get back the following milliseconds per message have been measured using the protocols isc capabilities( dont understand what is actually being measured? look at the bandwidth chapter in *RoundTrip*).

ISC measurements bandwidth	
Language: python3	
protocol	ms/msg
MQTT	6.5
ZMQ	7.3
DDS	under construction

If the processes communicate in an interprocess way the following times are measured:

IPC measurements bandwidth	
Language: python3	
protocol	ms/msg
MQTT	9.5
ZMQ	0.4
DDS	under construction

## sprint features

Total amount of feature points to spend: 40

Below all options with their tasks have been laid out that can be taken up for this sprint.

### sprint options:

- **Compare 10 ZMQ variations in python [15 fp] (hp)**
  - research ten possible configurations for ZMQ [2 fp]
  - first ping pong [2 fp]
  - first round trip [2 fp]
  - second ping pong [1 fp]
  - second round trip [1 fp]
  - next added ping pong [0.5 fp]
  - next added round trip [0.5 fp]
- **Compare 3 different MQTT setups in python [2 fp] (hp)**
  - research three possible MQTT setups [1 fp]
  - measuring round trip time [1 fp]
- **Change DDS to cyclonedds from eclipse and timing ISC [2 fp] (hp)**
  - install cyclonedds [1 fp]
  - test cyclonedds [1 fp]
- **Round trip time MQTT in C++ IPC and ISC [4 fp]**

- install library [1 fp]
  - working pingpong [2 fp]
  - round trip time [1 fp] This ^ is the same for each of the following round trip time
- Round trip time DDS in C++ IPC and ISC [4 fp]
- Round trip time ZMQ in C++ IPC and ISC [4 fp]
- Round trip time MQTT in java IPC and ISC [4 fp]
- Round trip time DDS in java IPC and ISC [4 fp]
- Round trip time ZMQ in java IPC and ISC [4 fp]
- **bandwidth measurement 2 fp per language and protocol [18 fp]**
  - write program that measures bandwidth [2 fp]
- Documentation for sprint 1 [4 fp]
  - planned features and finished futures [2 fp]
  - sprint results in sphinx table format [2 fp]
- **Documentation for sprint 2 [8 fp]**
  - sprint results [2 fp]
  - software installation and execution [4 fp]
  - ZMQ, MQTT and DDS workings [2 fp]
- Pathways setup and protocol integration [24 fp]

**chosen sprint goals:**

- Compare 10 ZMQ variations in python [15 fp] (hp)
- Compare 3 different MQTT setups in python [2 fp] (hp)
- Change DDS to cyclonedds from eclipse and timing ISC [2 fp] (hp)
- Documentation for sprint 2 [8 fp] (hp)
- Documentation for sprint 1 [4 fp] (mp)
- bandwidth measures for python ZMQ, MQTT and DDS [6 fp] (mp)
- Round trip time DDS in C++ IPC and ISC [4 fp] (lp)
- Round trip time ZMQ in C++ IPC and ISC [4 fp] (lp)

### Team Hurricane



### Setup Guide for CycloneDDS

#### Getting Started

The following are required before building Cyclone DDS:

- C Compiler (i.e. GCC).
- GIT.
- CMAKE (3.7 or later).
- OpenSSL (1.1 or later)
- Java JDK (8 or later).
- Apache Maven (3.5 or later).

#### CycloneDDS Installation

To download CycloneDDS on Linux/Raspberry Pi:

```
$ git clone https://github.com/eclipse-cyclonedds/cyclonedds.git
$ cd cyclonedds
$ mkdir build
```

To build and install CycloneDDS on Linux/Raspberry Pi:

```
$ cd build
$ cmake -DCMAKE_INSTALL_PREFIX=<install-location> ..
$ cmake --build .
$ cmake --build . --target install
```

**Note:** The two `cmake --build` commands might require being executed with `sudo` depending on the `<install-location>`.

---

**Warning:** Apache Maven requires internet access to update the list of artifacts. This will be required when executing `cmake --build ..`. Since the Raspberry Pi is not allowed to connect to the Sogeti network this was achieved using a mobile hotspot.

## Testing CycloneDDS Installation

In order to test the CycloneDDS installation on that particular device we are going to run the Hello World example that is provided. Open two terminals and move to the `~/cyclonedds/build/bin/` directory and in each terminal run:

```
$ ./HelloworldSubscriber
$ ./HelloworldPublisher
```

## Notes

**Note:** In order to customize the configuration of Cyclone DDS create a `.xml` file and point to it (in Linux) by:

```
$ export CYCLONEDDS_URI=file://$PWD/cyclonedds.xml
```

For example the following configuration file was utilized to choose the `wifi0` network interface over all others.

```
<?xml version="1.0" encoding="UTF-8" ?>
<CycloneDDS xmlns="https://cdds.io/config" xmlns:xsi="http://www.w3.org/2001/
→XMLSchema-instance" xsi:schemaLocation="https://cdds.io/config https://raw.
→githubusercontent.com/eclipse-cyclonedds/cyclonedds/master/etc/cyclonedds.xsd">
  <Domain id="any">
    <General>
      <NetworkInterfaceAddress>wifi0</NetworkInterfaceAddress>
    </General>
  </Domain>
</CycloneDDS>
```

## Tips

**Tip:** Windows installation of CycloneDDS proved somehow problematic, instead a Linux subsystem using Ubuntu was utilized from within Windows.

**Todo:** Install CycloneDDS on a Windows machine.

---

### Links

- CycloneDDS Repository: <https://github.com/eclipse-cyclonedds/cyclonedds>
- Introducing Eclipse Cyclone DDS: <https://www.youtube.com/watch?v=-qVHbTTXRKs&t=209s>
- Eclipse Cyclone DDS: Data Sharing in the IoT Age: <https://www.youtube.com/watch?v=2q86azIbSvA>

### Setup Guide for Cyclone DDS Python API

#### Getting Started

The Python DDS binding depends on:

- jsonpickle
- CycloneDDS

#### Installation

To install jsonpickle via Pip on Linux/Raspberry Pi:

```
$ pip3 install jsonpickle
```

To download, build and install the Python DDS binding on Linux/Raspberry Pi run:

```
$ git clone https://github.com/atolab/cdds-python
$ cd python-cdds
$ ./configure
$ python3 setup.py install
```

Since two library files are required for the Python binding after executing the first `./configure` go into `/cdds-python/bit/build/` and copy the `libddstubs.so` into `/usr/local/lib/`.

Edit the following line in the `configure` script to be as follows:

```
BIT_LIB=libddsc.so
```

Also edit the following two lines in `CMakeLists.txt` to look as follows:

```
add_library(ddsc SHARED dds_stubs.c)
target_link_libraries(ddsc ddsbit CycloneDDS::ddsc)
```

Run again:

```
$ cd python-cdds
$ ./configure
$ python3 setup.py install
```

Finally copy the `libddsc.so` file from `/cdds-python/bit/build/` into `/usr/local/lib/`.

**Warning:** When executing `./configure` an error might occur stating that `idlc_generate` was not found. Simply edit the `CMakeLists.txt` inside `/cdds-python/bit/` to look for the Cyclone DDS package in the correct location:



```
find_package(CycloneDDS REQUIRED COMPONENTS idlc PATHS "${CMAKE_CURRENT_SOURCE_DIR}/
→ ../../../../../../")
```

**Warning:** Running the command `python3 setup.py install` requires internet connection in order to configure the jsonpickle package.

## Links

- Cyclone DDS Python API: <https://github.com/atolab/cdds-python>

## An Introduction to DDS with Examples

The following guide aims to present some of the core functionalities and features of DDS that were used by Team Hurricane when implementing the Matrix Board Communication demonstrator in C. The main goal of this article is to present a naive and simple approach to the complete process of creating a DDS application.

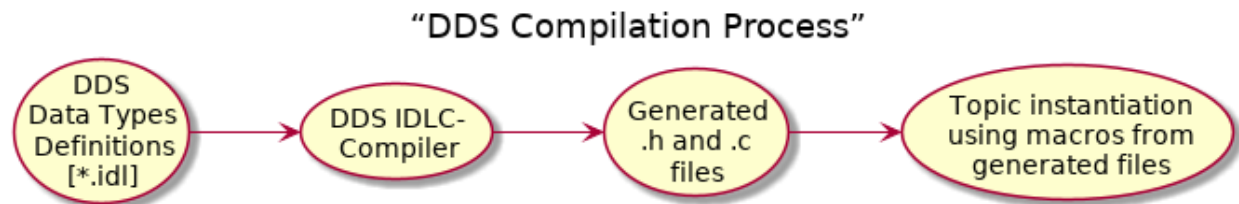
## Topics in DDS

Topics in DDS are data objects for which values will be published/received to/from. Each topic consists of the following:

- The topic type.
- The topic name.
- A set of Quality of Service (QoS) properties.

The topic type defines the data type that will be stored in that particular topic, while the topic name is a string that identifies that topic within that particular communication domain. A DDS topic type contains an IDL structure and a key. The QoS policies of the topic control the behavior of the DataReaders and DataWriters of that topic.

Topics are defined using the IDL specification language enabling the topics to retain independence from a certain programming language. [Mapping of IDL data types](#) presents the mapping between various IDL data types to data types of various programming languages. The figure below shows how a typical DDS application is built.



In the case of the Matrix Board Communication the topic type `Msg` was defined and it includes a structure with two `long` and one `string` IDL primitives. The `userID` defines the identity number of the matrix board writing to that topic while the `traffic` defines the traffic that was sensed by that matrix board. Finally the `message` string is simply the name of that particular topic. The following snippets of code show the mapping from the IDL to C language of the topic type `Msg` defined for the communication of the matrix boards.

```
struct Msg
{
    long userID;
    long traffic;
    string message;
};
#pragma keylist Msg userID
```

```
typedef struct MBCData_Msg
{
    int32_t userID;
    int32_t traffic;
    char * message;
} MBCData_Msg;
```

Within our application before initializing a topic, we must create a participant on a given domain (in this case the default domain) using:

```
participant = dds_create_participant(DDS_DOMAIN_DEFAULT, NULL, NULL);
```

In the context of the matrix board communication, a matrix board can initialize a topic using the following function:

```
topic = dds_create_topic(participant, &MBCData_Msg_desc, topic_name, NULL, NULL);
```

In the function above, the `&MBCData_Msg_desc` refers to the data type that the topic is initialized to, while `topic_name` is a string containing the name of the topic. Topics with the same data type but different names are considered different topics.

### Data Writers in DDS

Data writers are DDS entities which have the ability to write to a given topic.

After initializing a participant on a certain domain as well as a topic, one can create a data writer on that particular topic using:

```
writer = dds_create_writer(participant, topic, NULL, NULL);
```

It is possible to add another layer of abstraction which is a publisher. A publisher includes multiple writers and its main purpose is to manage his group of writers.

In the context of the matrix board communication, each matrix board has only one data writer which writes data only to the topic of that particular board.

### Writing Data to a Topic

In order to demonstrate how data can be written to a particular topic let us look again at the example of the matrix board communication. First initialize the data that needs to be written:

```
MBCData_Msg msg;
/* Create a message to write. */
msg.userID = MBC_id;
msg.traffic = traffic_number;
msg.message = topic_name;
```

And then simply write the data using the writer and initialized data using:

```
dds_write (writer, &msg);
```

Also, DDS provides event-based activation. Meaning that we can perform a write operation depending on a particular event, (e.g. a data reader coming online etc.). In the case of the matrix board communication, each board updates its own topic every second (polling based write).

## Data Readers in DDS

Data readers are DDS entities which have the ability to read from a given topic.

After initializing a participant on a certain domain as well as a topic, one can create a data reader on that particular topic using:

```
reader = dds_create_reader(participant, topic, NULL, NULL);
```

Similarly to publishers, it is possible to add another layer of abstraction which is a subscriber. A subscriber includes multiple readers and its main purpose is to manage his group of readers.

In the context of the matrix board communication, each matrix board has an array of readers starting from its own ID number and up to a certain predefined ID number. It is obvious that it has the respective array of topics since they are required to create each reader. This enables the scanning and detection of two of the closest matrix boards which are “alive”.

## Reading Data from a Topic

In order to read data from a topic, one has to perform some additional steps when compared to a write operation. First of all, we must initialize a `sample` array using the macros created from the compilation of our `.idl` file (see resulting `.c` and `.h` files from *Topics in DDS* chapter):

```
samples[0] = MBCData_Msg__alloc ();
```

Then, for the actual data read we have to options. The first option is to use the following function:

```
dds_read(reader, samples, infos, MAX_SAMPLES, MAX_SAMPLES);
```

Using `dds_read` will return a copy of the read samples, leaving them available in the reader’s history cache so they can be read again later (marking it as “read” along the way). While the second option is to utilize:

```
dds_take(reader, samples, infos, MAX_SAMPLES, MAX_SAMPLES);
```

Using `dds_take` will return the samples but also removes them from the reader’s history cache (but not from any other readers’ history caches). In the case of the matrix board the `dds_take` function was used since we do not use previous samples for anything and it is more efficient to remove them from our cache. After reading the data one has to also free the data location that was initialized, again using the provided macros. An example from the MBC is the following:

```
MBCData_Msg_free (samples[0], DDS_FREE_ALL);
```

## Links

- Mapping of IDL data types: <https://tinyurl.com/qt28c5s>
- DDS Tutorial by A. Corsaro: [http://www.laas.fr/files/SLides-A\\_Corsaro.pdf](http://www.laas.fr/files/SLides-A_Corsaro.pdf)

## DDS Performance Study

In this part we look at important design choices around DDS and discover how different choices influence the performance of DDS. The design choices in question can be summarized to the following:

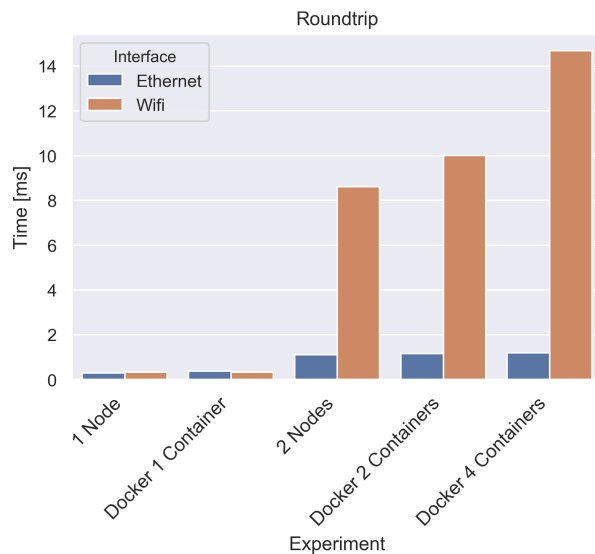
1. Docker deployment and its effect on performance.
2. Ethernet vs. WiFi networking.
3. Router effects on performance.
4. CPU frequency scaling and its effect on performance.

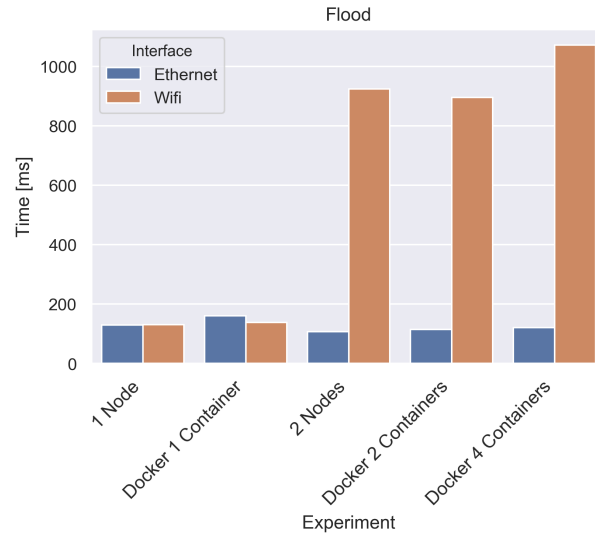
In order to answer all these questions we choose the Roundtrip and Flood demonstrators since the former gives a good estimate on how quickly a message can move through our DDS network and the latter also saturates our network as well as the processor usage. In each case we look at a network of 4 hubs with 1 master hub and 3 slave hubs and a Flood with size of 1000 messages. For more information on the setup refer to *Round trip and Flood in C*.

### Docker and Ethernet vs. WiFi Performance

In this section we are going to look at the performance of DDS when deployed using standalone executables on a machine compared to deployment of DDS through containers. In each scenario we also look at both wired and wireless networking. We define the following scenarios:

1. **1 Node:** all 4 hubs run on a single machine (Inter-Process Communication).
2. **Docker 1 Container:** all 4 hubs run inside one container on one machine (Inter-Container Communication).
3. **2 Nodes:** Master hub runs on machine A, Hub 1 runs on Machine B, Hub 2 runs on machine A and Hub 3 runs on machine B (Inter-System Communication).
4. **Docker 2 Containers:** Master hub runs in container 1A of machine A, Hub 1 runs in container 1B of Machine B, Hub 2 runs in container 1A of machine A and Hub 3 runs in container 1B of machine B.
5. **Docker 4 Containers:** Master hub runs in container 1A of machine A, Hub 1 runs in container 1B of Machine B, Hub 2 runs in container 2A of machine A and Hub 3 runs in container 2B of machine B.





The results for both the Roundtrip and Flood experiments can be seen in the figures above. We can conclude that there is no performance penalty for using DDS inside a Docker container, this can be seen by comparing the **1 Node** to the **Docker 1 Container** for both demonstrators. Furthermore, we can deduct that over Ethernet Docker container deployment performs in an identical manner to the normal standalone deployment (i.e. comparing **2 Nodes** to **Docker 2 Containers**). This is an expected result since we use host networking which means that the container sees the host's network as its own. In the case we would use bridged networking mode we would expect the performance to be worse due to the added overhead of going through the host's internal bridge connection. In addition, we also see that for the Flood experiment over Ethernet all experiments yield the same results due to the fact that we saturate our network bandwidth. In the case of WiFi, it is logical to have larger execution times when moving to 2 separate nodes.

In conclusion, we expected the wireless setup to perform much slower than a wired connection. The most important take away from this study is that Docker containers perform as good as a native implementation. More on this matter can be found in an interesting paper by IBM with the title 'An Updated Performance Comparison of Virtual Machines and Linux Containers' <<https://tinyurl.com/hmj4vex>>'. Finally, we note that inter-container communication results in the same connection penalty as communication between separate nodes.

## Router Effect on Performance

In this section, we look at the effect different routers will have on the performance. This research is conducted in order to determine the difference between the following categories of routers:

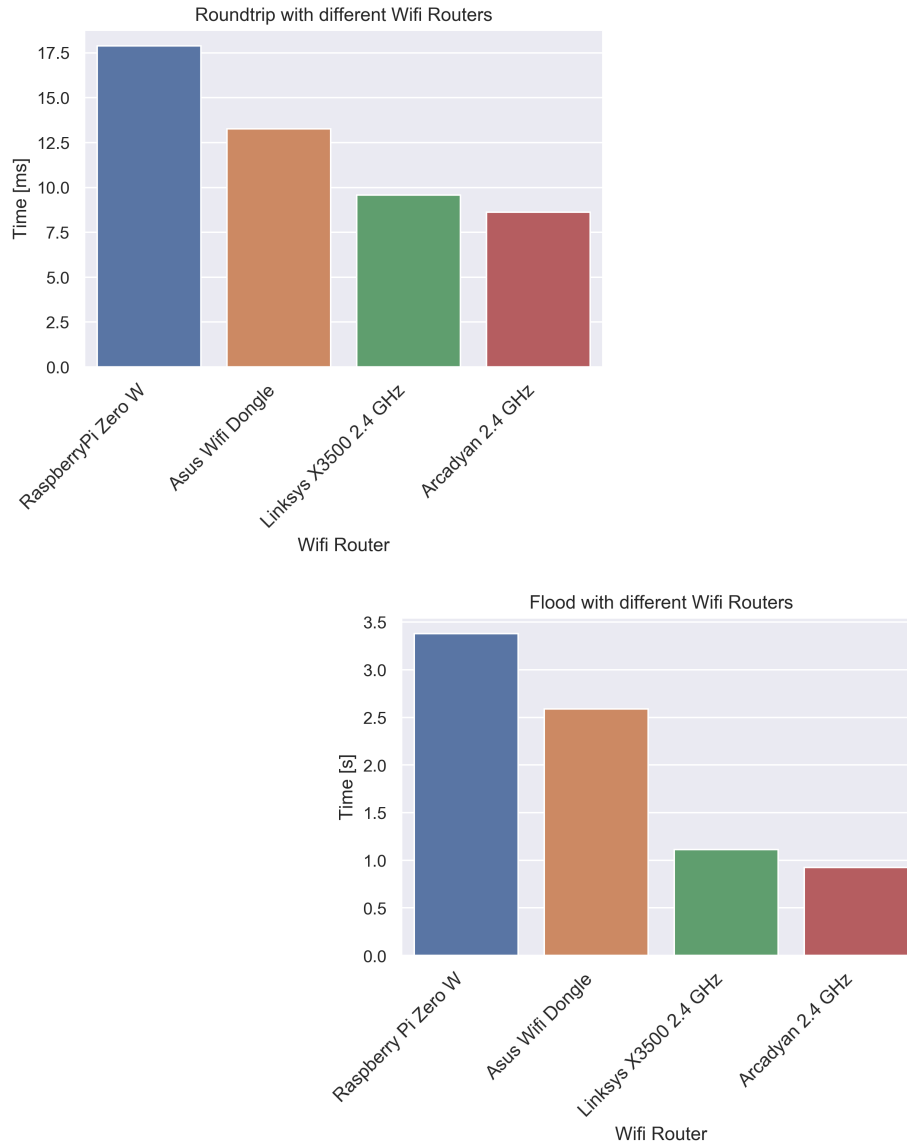
1. **Cheap Embedded Router Solution**, such as the Raspberry Pi Zero W.
2. **Modular lightweight and low cost Solution**, such as a USB WiFi dongle.
3. **Dedicated mid price point router Solution**.
4. **Dedicated high price point router Solution**.

The routers under investigation are the following:

1. **Raspberry Pi Zero W**.
2. **Asus USB-AC53 dongle**.
3. **Linksys X3500 router**.
4. **Arcadyan AC2200 router**.

Both Roundtrip and Flood demonstrators were setup up with the 2 node configuration:

- **2 Nodes: Master hub runs on machine A, Hub 1 runs on Machine B, Hub 2 runs on machine A and Hub 3 runs on machine B (Inter-System Communication).**

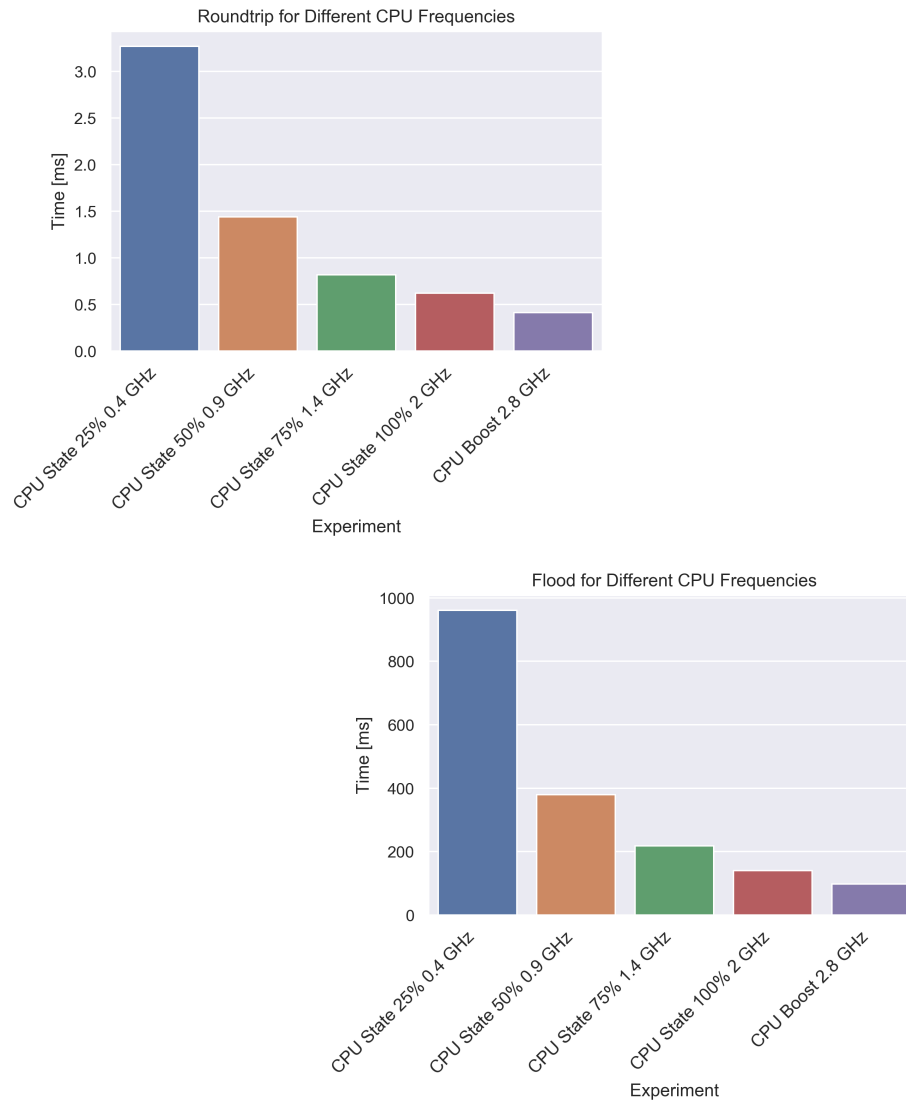


It is important to note at this point that all configurations were positioned in close proximity from the wireless router in order to alleviate any differences caused by different antenna sizes at higher distances (i.e. dedicated routers larger antenna arrays). The results show a clear advantage for the dedicated routers. In the case of the Roundtrip demo we see that the expensive router provides a 2 time performance boost over the cheap embedded router in the Raspberry Pi Zero W, while in the case of Flood where the network is saturated this difference is in the order of 3. We can already draw our first conclusion, which is that for applications where higher bandwidth is required, a dedicated router is the clear winner. At the same time, if we look at the cost effectiveness of our solution the price difference is almost 10x as high for the expensive dedicated router over a Raspberry Pi Zero. Similarly, the USB WiFi dongle also gives a performance boost over the embedded router but the difference in performance does not justify the difference in price which is 2x more just for a dongle compared to a full computer. Finally, it is also interesting to compare the two dedicated routers that come at different price points (50 euros vs. 100 euros). The more expensive router is by a small margin faster. It is interesting to note that the two dedicated routers were also tested in the same experimental setup over Ethernet connection, with identical performance results. In conclusion, the choice of router really depends on the application. An application that requires high bandwidth and/or across an increasing number of nodes will definitely

benefit vastly from a dedicated high performance router. On the other hand, one can easily see the benefits of using an embedded device or a dongle as a router.

## Effect of CPU Frequency Scaling on Performance

In this section we investigate the effect of a CPU frequency scaling on the performance of our two DDS demonstrators (i.e. Roundtrip and Flood). For this setup we choose to utilize the Inter-process Communication on a single machine. Furthermore, we choose to utilize the Sogeti laptop that utilizes an Intel i5-8350U with 4 cores and 8 threads. This choice was made mainly due to the ease of scaling the frequency on a Windows machine through different maximum CPU states.



The results show an exponential decay in execution time with respect to CPU frequency, for both Roundtrip and Flood. As a result, we can conclude that Flood is bandwidth bound and the communication speed is more important than the per node calculation time.

### Docker Guide: How to create and run a Docker Image

The following guide will discuss all the necessary steps in order to create a Docker image and also how to run a container from an image.

#### Installing Docker

The first step is to install Docker on our machine of interest. For Windows/MacOS we require Docker Desktop. This can be downloaded from [here](#). An important note is that we prefer the Edge channel rather than the Stable channel, since the Edge channel enables us to build on x86 machines images for other architectures such as ARM (see [Building Multi-Arch Images](#) chapter).

For Linux Systems such as the Raspberry Pi:

```
$ sudo apt-get update && sudo apt-get upgrade
$ curl -fsSL https://get.docker.com -o get-docker.sh
$ sudo sh get-docker.sh
```

Add a Non-Root User to the Docker Group:

```
$ sudo usermod -aG docker [user_name]
```

Check Docker version:

```
$ docker version
```

#### Docker Workflow

We begin by describing the general Docker workflow. Usually, we begin by creating a Dockerfile. In a nutshell, a Dockerfile is a script which describes the commands we would like to execute within a container. This container is then saved as a new Docker image. Let's provide an example for the sake of understanding:

```
FROM alpine:latest

RUN apk update && apk add git && apk add bash

WORKDIR /src/

RUN git clone https://github.com/eclipse-cyclonedds/cyclonedds.git
```

This example clones the latest Alpine Linux image using the `FROM` command, updates and installs `git` and `bash` from the `apk` repo using the `RUN` command. Then creates a folder `src` moves into it and clones the CycloneDDS repo. The result is an image that runs Alpine Linux, has `git` and `bash` installed and the repo CycloneDDS in a `src` directory. The above described workflow can be seen in the following diagram.





The following link presents some of the [Best practices for writing Dockerfiles](#). In addition, the following link is the [Dockerfile reference guide](#).

Similarly to any other code hosting service (e.g. github, bitbucket etc.) there is a Docker Hub that hosts Docker images. As a result, in some cases the workflow from above can be changed to skip building the image from a Dockerfile to simply cloning a pre-existing image that satisfies that particular need.

## Multi-stage Builds

Multi-stage builds refer to Dockerfiles that use various images throughout the building process. It is easier to explain this with an example, consider the following:

```
FROM alpine:latest AS build

RUN apk update && apk add git && apk add cmake && apk add --no-cache build-base && \
    apk add maven && apk fetch openjdk8 && apk add openjdk8 && apk add linux-headers &
    & \
    apk add bash

WORKDIR /src/

RUN git clone https://github.com/eclipse-cyclonedds/cyclonedds.git

WORKDIR /dds/

WORKDIR /src/cyclonedds/build

RUN cmake -DCMAKE_INSTALL_PREFIX=/dds/ .. && \
    cmake --build . && cmake --build . --target install

FROM alpine:latest

RUN apk update && apk add bash

WORKDIR /dds/

COPY --from=build /dds/ ./
```

The first stage named BUILD will start from the latest Alpine Linux image and will install all the dependencies required by CycloneDDS (more on these dependencies can be found [Setup Guide for CycloneDDS](#)). Subsequently, it clones the CycloneDDS repo and builds it. Then we move to the second and final stage, to which we will refer to as the DEPLOY stage (separation of the two stages can be seen from the different FROM commands). In the DEPLOY stage, we start from the clean latest Alpine Linux image and we copy only the executables we compiled in the previous stage. This separation enables us to provide to the end user only the executables required for our application in a minimal Linux environment. Also, the resulting image is drastically smaller in size.

## Building Docker Images

After creating our Dockerfile, it is time to build our image which can be done by navigating to the folder of our Dockerfile and executing:

```
$ docker build -t <image name> .
```

Similarly, we can also build images for each of the different stages we created, using:

```
$ docker build --target <stage name> -t <image name> .
```

Finally, using `$ docker images` we can see all the images in our system.

The build process can be seen in the following video.

### Running Docker Containers

Containers are running instances of images. In order to create a new container we run the following command:

```
$ docker run --network="host" -ti --name <container name> <image name> bash
```

We provide a meaningful name for our new container and the name of an already existing image. Also, note that there are many different options, but in the case of DDS we want our container to have the IP address of the host, so that it can communicate over the host's network.

**Warning:** Host networking does not work on Docker Desktop for Windows/MacOS. Basically, the command can be used as is but the IP address will not be the same as the one the host receives from its network. Nevertheless, containers can still communicate between each other and within the same container, just not with the outside network.

Furthermore, we can open another terminal on an already running container by using:

```
$ docker exec -it <existing container name> bash
```

**Note:** In all of the above mentioned commands we execute `bash` since it is installed on our images. If you do not have `bash` installed use `/bin/sh` instead.

### Building Multi-Arch Images

Finally we describe how we can actually build images for different architectures than the one on the machine we are currently building on. This is useful in the case we want to build an image for the Raspebrry Pi on a x86 Windows Machine which has more processing power. A pre-requisite for this is to have the Edge channel of Docker Desktop installed. First we check what `buildx` builders are available and create a new builder using:

```
$ docker buildx ls
$ docker buildx create --name <builder name>
```

We select that specific builder using:

```
$ docker buildx use <mybuilder>
$ docker buildx inspect --bootstrap
```

Finally, we build and push our new image for the selected architecture to a new Docker Hub repo using:

```
$ docker buildx build --platform linux/arm/v7 -t <username/repo name>:<image name> --
➔push .
```

## Building Demo Image

The Dockerfile created for the C implementations of the Round trip as well as the different versions of the Matrix Board Communication can be found in the `~/src/docker/Build_C_MBC_Roundtrip/` directory of this repository. First, we need to build this image by navigating to the above mentioned directory and using:

```
$ docker build -t <image name> .
```

Then we can create a container of this image using:

```
$ docker run -it --network="host" --name <container name> <image name>
```

For more information on how to execute the experiments please refer to the docker sections of each respective *Round trip and Flood in C* and *Matrix Board Communication in C* demonstrator.

## Links

- Shipping C++ Programs in Docker: A Journey Begins: <https://tinyurl.com/yxn4gr3s>
- Building Multi-Arch images: <https://www.docker.com/blog/multi-arch-images/>

## Deploying DDS Over WAN

In this section we are going to discuss how we can deploy Cyclone DDS over WAN. We can think of many interesting use cases in which we would like our DDS applications to be able to communicate with each other across the wider network. Although many of the commercial DDS implementations include extensive documentation on how to configure the DDS library through a `.xml` configuration file in order for applications to be able to communicate over the wider network, the Cyclone DDS documentation is still lacking the explanation required for us to implement such a scenario. Furthermore, some features are still not implemented in Cyclone DDS since there is no official release of this library. As a result, performing the across-network discovery, which is the main issue in our case, can prove quite difficult without utilizing some sort of common networking method.

In order to make sure that both remote nodes, which we wanted to connect, could discover each other it was decided to create a VPN network using just a simple online service. This VPN service focuses on IoT lightweight networking solutions. By connecting both nodes (in this case 2 Raspberry Pis with different internet connections) to this VPN network we could move to configuring our discovery protocol. The discovery can be done in a similar manner to the way we choose our DDS network interface, namely through the same `cyclonedds.xml` file. First, we find the IPv6 addresses provided by our VPN network to each individual node and we add them in our configuration by adding the following block.

```
<Discovery>
  <Peers>
    <Peer address="[IPv6-address of local machine]"/>
    <Peer address="[IPv6-address of remote machine]"/>
  </Peers>
  <ParticipantIndex>auto</ParticipantIndex>
</Discovery>
```

We add both IPv6 address to make sure that our configuration file can actually link our local machine to the remote. We can also add the IPv4 addresses, but Cyclone DDS does not allow IPv4 and IPv6 address mixing. Finally, we disable multicast since it is not supported by the VPN service and add all these blocks to our final configuration file.

```
<?xml version="1.0" encoding="UTF-8" ?>
<CycloneDDS xmlns="https://cdds.io/config" xmlns:xsi="http://www.w3.org/2001/
→XMLSchema-instance" xsi:schemaLocation="https://cdds.io/config https://raw.
→githubusercontent.com/eclipse-cyclonedds/cyclonedds/master/etc/cyclonedds.xsd">
  <Domain id="any">
    <General>
      <NetworkInterfaceAddress>auto</NetworkInterfaceAddress>
      <AllowMulticast>false</AllowMulticast>
      <MaxMessageSize>65500B</MaxMessageSize>
      <FragmentSize>4000B</FragmentSize>
      <Transport>udp6</Transport>
    </General>
    <Discovery>
      <Peers>
        <Peer address="[IPV6-address]" />
        <Peer address="[IPV6-address]" />
      </Peers>
      <ParticipantIndex>auto</ParticipantIndex>
    </Discovery>
    <Internal>
      <Watermarks>
        <WhcHigh>500kB</WhcHigh>
      </Watermarks>
    </Internal>
    <Tracing>
      <Verbosity>severe</Verbosity>
      <OutputFile>stdout</OutputFile>
    </Tracing>
  </Domain>
</CycloneDDS>
```

An important conclusion that was drawn from the WAN experiment was that we need to add softer QoS constraints into our application in order for the communication to function properly. For example, poc3 of the Matrix Board Communication demonstrator does not function consistently well, mainly due to the fact that the communication is extremely fast and expects that the samples require deletion after a few milliseconds. In the WAN case, those milliseconds are not enough for the receiving node to actually read the message. In all other demonstrators where the communication is much slower this issue was not noticed. In conclusion, adding such functionality requires the developer to re-think his DDS timing constraints and add more relaxed QoS standards.

### Links

- Using Cyclone DDS with Husarnet on ROS2: <https://husarion.com/tutorials/other-tutorials/husarnet-cyclone-dds/>

### Setting up Raspberry Pi network

In order to create a Raspberry Pi network, one of the Raspberry Pis will be used as a wifi router where all the other devices will connect to.

The first step is to install DNSMasq and HostAPD using:

```
$ sudo apt install dnsmasq hostapd
```

Turn these two processes off:

```
$ sudo systemctl stop dnsmasq
$ sudo systemctl stop hostapd
```

## Configuring a static IP

To configure the static IP address, edit the dhcpcd configuration file with:

```
$ sudo nano /etc/dhcpcd.conf
```

Go to the end of the file and edit it as follows:

```
interface wlan0
    static ip_address=192.168.4.1/24
    nohook wpa_supplicant
```

Restart the dhcpcd daemon using:

```
$ sudo service dhcpcd restart
```

## Configuring the DHCP server

First move the old configuration file into the .orig file and start a new one:

```
$ sudo mv /etc/dnsmasq.conf /etc/dnsmasq.conf.orig
$ sudo nano /etc/dnsmasq.conf
```

Add the following two lines

```
interface=wlan0      # Use the require wireless interface - usually wlan0
dhcp-range=192.168.4.2,192.168.4.20,255.255.255.0,24h
```

Interface wlan0 will provide IP addresses between 192.168.4.2 and 192.168.4.20.

Start the dnsmasq:

```
$ sudo systemctl start dnsmasq
```

## Configuring the access point

Run the following command to edit the hostapd configuration file:

```
$ sudo nano /etc/hostapd/hostapd.conf
```

In case the file does not exist simply create it at the above mentioned location and add the following lines:

```
interface=wlan0
driver=nl80211
ssid=PID3
hw_mode=g
channel=7
wmm_enabled=0
macaddr_acl=0
```

(continues on next page)

(continued from previous page)

```
auth_algs=1
ignore_broadcast_ssid=0
wpa=2
wpa_passphrase=Sogeti2020
wpa_key_mgmt=WPA-PSK
wpa_pairwise=TKIP
rsn_pairwise=CCMP
```

Now we need to tell the system where to find the above mentioned file by:

```
$ sudo nano /etc/default/hostapd
```

By changing the following line to:

```
DAEMON_CONF="/etc/hostapd/hostapd.conf"
```

### Start up

Finally start all the required services by running:

```
$ sudo systemctl unmask hostapd
$ sudo systemctl enable hostapd
$ sudo systemctl start hostapd
```

### Routing and masquerade

Edit `/etc/sysctl.conf` and uncomment this line:

```
net.ipv4.ip_forward=1
```

Add a masquerade for outbound traffic on `eth0` and save the iptables rule:

```
$ sudo iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
$ sudo sh -c "iptables-save > /etc/iptables.ipv4.nat"
```

Edit `/etc/rc.local` and add this just above “exit 0” to install these rules on boot:

```
$ iptables-restore < /etc/iptables.ipv4.nat
```

Reboot and check if it works by connecting any device with wifi to this new network with the SSID and password provided in the `hostapd.conf` file.

Now it is possible to use `ssh` on that network to connect to any of the devices on the network including the router node.

### IP addresses

The following IP addresses or hostnames with the corresponding passwords can be used to `ssh` into the Raspberry Pis that connect to the wifi network with SSID = PID3 on raspberrypi-202.

Hostname	IP address	Password
raspberrypi-200	192.168.4.3	200
raspberrypi-201	192.168.4.6	201
raspberrypi-202	192.168.4.1	202
raspberrypi-203	192.168.4.20	203
raspberrypi-204	192.168.4.12	204
raspberrypi-205	192.168.4.13	205
raspberrypi-206	192.168.4.14	206

## Links

- Raspberry Pi as an access point (NAT): <https://tinyurl.com/y3byvmjz>

## Setup Guide for PlantUML

- 1) Download the latest `plantuml.jar` [here](#).
- 2) Save the following script as `plantuml` (without the `.sh` extension) in some directory, like `~/bin`.

```
#!/bin/sh

PLANTUML=/path/to/plantuml.jar

if [ -n "${JAVA_HOME}" ] && [ -x "${JAVA_HOME}/bin/java" ] ; then
    JAVA="${JAVA_HOME}/bin/java"
elif [ -x /usr/bin/java ] ; then
    JAVA=/usr/bin/java
else
    echo Cannot find JVM
    exit 1
fi

$JAVA -jar ${PLANTUML} ${@}
```

- 3) Finally, Add `~/bin` to your `PATH`, with:

```
$ export PATH=$PATH:~/bin
```

**Note:** Add this last command in your `.bashrc` in order to make it persistent.

## How-to keep your forked repo up-to-date

In order to keep our forked repository up to date with the original repository as well as with the repositories of each of the collaborators, we add upstream repositories from which we can merge the changes into our repository.

First run:

```
$ git remote add upstream <url_to_repo>
```

**Note:** In the above command `upstream` refers to the name so give it a significant name if you are going to add multiple remote repositories.

List all the remote repositories with:

```
$ git remote -v
```

If the remote has been added successfully do the following to merge the changes into your own master branch:

```
$ git fetch upstream
$ git checkout master
$ git merge upstream/master
```

If you want to push the changes to your own remote repository do:

```
$ git push origin master
```

Finally, in order to remove the `upstream` remote repository simply:

```
$ git remote rm upstream
```

## Links

- Keep forked repository current: <https://dev.to/ranewallin/how-to-keep-your-forked-repository-current-38mn>

## Merging Repositories

Question: Can we merge a repository into another without losing history? For merging two existing repositories a lot can be found on the internet. Eventually I merged two found methods into one method. The second part of the first method did not result in the wanted behavior. For that the second method was used.

The found methods:

- [method 1](#)
- [method 2](#)

**Tip:** Within the following steps repository A will be merged into repository B. It is wise to create a temporary directory in which both repositories will be cloned, so when a certain step does not have the wanted outcome it will be fairly easy to start all over again.

**Note:** Most of the steps below are shown from a script used to execute the merge between two repositories, see `merge_git_repo_a_to_repo_b`.



## Steps

Create temporary directory:

```
mkdir $HOME/tmp/merge_git_repos
cd $HOME/tmp/merge_git_repos
```

Clone repository A into a temporary name tmp\_repo\_a:

```
git clone <url of repo a> tmp_repo_a
```

Clone repository B into a temporary name tmp\_repo\_b:

```
git clone <url of repo b> tmp_repo_b
```

Execute script for merging repo a into repo b:

```
merge_git_repo_a_to_repo_b
```

**Note:** The script can also be used with the locations of the two repositories (e.g. merge\_git\_repo\_a\_to\_repo\_b <location repo a> <location repo b>). If the location are omitted the defaults tmp\_repo\_a and tmp\_repo\_b are used.

Executed steps from the shell script:

Checkout all the branches you want to copy from repository A:

```
cd $REPO_A_PATH
git branch -a > $MERGE_PATH/repo_a_branches.txt
for branch in `cat $MERGE_PATH/repo_a_branches.txt`
do
    git checkout $branch
done
```

Fetch all the tags from repository A:

```
git fetch --tags
```

Check if you have all the tags and branches:

```
git tag
git branch -a
```

Clear the link to the remote of repository A:

```
git remote rm origin
```

Do some filtering for rewriting the history:

```
git filter-branch -f --tree-filter 'mkdir -p toDelete;git mv -k docs/Makefile_
↳toDelete/.;git mv -k docs/_external_templates/conf/* toDelete/.;git mv -k docs/_
↳external_templates/static/* toDelete/.;git mv -k docs/requirements.txt toDelete/.;
↳git mv -k docs/doc/_generic.inc toDelete/.;git mv -k docs/doc/conf.py toDelete/.;
↳git mv -k docs/doc/index.rst toDelete/docs_index.rst;git mv -k README.md toDelete/.
↳' HEAD
git filter-branch -f --tree-filter 'mkdir -p docs/doc/Teams/0.Talk_like_pi;git mv -k_
↳docs/doc/teams/2018.1_Talk_like_pi/* docs/doc/Teams/0.Talk_like_pi;git mv -k docs/
↳doc/teams/index.rst toDelete/teams_index.rst' HEAD
```

(continues on next page)

(continued from previous page)

```

git filter-branch -f --tree-filter 'mkdir -p docs/doc/Demonstrators;git mv -k docs/
↳ doc/IPSC/* docs/doc/Demonstrators/' HEAD
git filter-branch -f --tree-filter 'mkdir -p src/demonstrators/
↳ ComparedToOtherProtocols/MQTT/Bandwidth/Python;mkdir -p docs/doc/Demonstrators/
↳ ComparedToOtherProtocols/Bandwidth/MQTT;git mv -k IPSC/MQTT/py/Bandwidth/* src/
↳ demonstrators/ComparedToOtherProtocols/MQTT/Bandwidth/Python; git mv -k src/
↳ demonstrators/ComparedToOtherProtocols/MQTT/Bandwidth/Python/readme.rst docs/doc/
↳ Demonstrators/ComparedToOtherProtocols/Bandwidth/MQTT/index.rst;git mv -k src/
↳ demonstrators/ComparedToOtherProtocols/MQTT/Bandwidth/Python/ipc/readme.rst docs/
↳ doc/Demonstrators/ComparedToOtherProtocols/Bandwidth/MQTT/ipc.rst;git mv -k src/
↳ demonstrators/ComparedToOtherProtocols/MQTT/Bandwidth/Python/isc/readme.rst docs/
↳ doc/Demonstrators/ComparedToOtherProtocols/Bandwidth/MQTT/isc.rst' HEAD
git filter-branch -f --tree-filter 'mkdir -p src/demonstrators/
↳ ComparedToOtherProtocols/MQTT/RoundTrip/Python;mkdir -p docs/doc/Demonstrators/
↳ ComparedToOtherProtocols/RoundTrip/MQTT;git mv -k IPSC/MQTT/py/RoundTrip/* src/
↳ demonstrators/ComparedToOtherProtocols/MQTT/RoundTrip/Python;git mv -k src/
↳ demonstrators/ComparedToOtherProtocols/MQTT/RoundTrip/Python/readme.rst docs/doc/
↳ Demonstrators/ComparedToOtherProtocols/RoundTrip/MQTT/index.rst;git mv -k src/
↳ demonstrators/ComparedToOtherProtocols/MQTT/RoundTrip/Python/ipc/readme.rst docs/
↳ doc/Demonstrators/ComparedToOtherProtocols/RoundTrip/MQTT/ipc.rst;git mv -k src/
↳ demonstrators/ComparedToOtherProtocols/MQTT/RoundTrip/Python/isc/readme.rst docs/
↳ doc/Demonstrators/ComparedToOtherProtocols/RoundTrip/MQTT/isc.rst' HEAD
git filter-branch -f --tree-filter 'mkdir -p src/demonstrators/
↳ ComparedToOtherProtocols/MQTT/PingPong/Python;mkdir -p docs/doc/Demonstrators/
↳ ComparedToOtherProtocols/PingPong/MQTT;git mv -k IPSC/MQTT/py/* src/demonstrators/
↳ ComparedToOtherProtocols/MQTT/PingPong/Python;git mv -k src/demonstrators/
↳ ComparedToOtherProtocols/MQTT/PingPong/Python/readme.rst docs/doc/Demonstrators/
↳ ComparedToOtherProtocols/PingPong/MQTT/index.rst' HEAD
git filter-branch -f --tree-filter 'mkdir -p src/demonstrators/
↳ ComparedToOtherProtocols/ZMQ/Bandwidth/Python;mkdir -p docs/doc/Demonstrators/
↳ ComparedToOtherProtocols/Bandwidth/ZMQ;git mv -k IPSC/ZMQ/py/Bandwidth/* src/
↳ demonstrators/ComparedToOtherProtocols/ZMQ/Bandwidth/Python;git mv -k src/
↳ demonstrators/ComparedToOtherProtocols/ZMQ/Bandwidth/Python/readme.rst docs/doc/
↳ Demonstrators/ComparedToOtherProtocols/Bandwidth/ZMQ/index.rst' HEAD
git filter-branch -f --tree-filter 'mkdir -p src/demonstrators/
↳ ComparedToOtherProtocols/ZMQ/RoundTrip/Python;git mv -k IPSC/ZMQ/py/RoundTrip/* src/
↳ demonstrators/ComparedToOtherProtocols/ZMQ/RoundTrip/Python' HEAD
git filter-branch -f --tree-filter 'mkdir -p src/demonstrators/
↳ ComparedToOtherProtocols/ZMQ/Push-Pull/Python;git mv -k IPSC/ZMQ/py/Push-Pull/* src/
↳ demonstrators/ComparedToOtherProtocols/ZMQ/Push-Pull/Python' HEAD
git filter-branch -f --tree-filter 'mkdir -p src/demonstrators/
↳ ComparedToOtherProtocols/ZMQ/Request-Reply/PingPong/Python;git mv -k IPSC/ZMQ/py/
↳ Request-Reply/PingPong/* src/demonstrators/ComparedToOtherProtocols/ZMQ/Request-
↳ Reply/PingPong/Python' HEAD
git filter-branch -f --tree-filter 'mkdir -p src/demonstrators/
↳ ComparedToOtherProtocols/ZMQ/Request-Reply/RoundTrip/Python;git mv -k IPSC/ZMQ/py/
↳ Request-Reply/Roundtrip/* src/demonstrators/ComparedToOtherProtocols/ZMQ/Request-
↳ Reply/RoundTrip/Python' HEAD
git filter-branch -f --tree-filter 'mkdir -p src/demonstrators/
↳ ComparedToOtherProtocols/ZMQ/PingPong/Python;mkdir -p docs/doc/Demonstrators/
↳ ComparedToOtherProtocols/PingPong/ZMQ;git mv -k IPSC/ZMQ/py/* src/demonstrators/
↳ ComparedToOtherProtocols/ZMQ/PingPong/Python;git mv -k src/demonstrators/
↳ ComparedToOtherProtocols/ZMQ/PingPong/Python/readme.rst docs/doc/Demonstrators/
↳ ComparedToOtherProtocols/PingPong/ZMQ/index.rst' HEAD
git filter-branch -f --tree-filter 'mkdir -p src/demonstrators/PingPong/Python;mkdir -
↳ p docs/doc/Demonstrators/PingPong;git mv -k IPSC/DDS/CycloneDDS/py/PingPong/* src/
↳ demonstrators/PingPong/Python;git mv -k src/demonstrators/PingPong/Python/readme.
↳ rst docs/doc/Demonstrators/PingPong/index.rst' HEAD

```

(continues on next page)

(continued from previous page)

```
git filter-branch -f --tree-filter 'mkdir -p src/demonstrators/RoundTrip/Python;mkdir
↳ -p docs/doc/Demonstrators/RoundTrip;git mv -k IPSC/DDS/CycloneDDS/py/Loop/* src/
↳ demonstrators/RoundTrip/Python;git mv -k src/demonstrators/RoundTrip/Python/readme.
↳ rst docs/doc/Demonstrators/RoundTrip/index.rst' HEAD
git filter-branch -f --index-filter 'git rm -r --cached --ignore-unmatch toDelete'
↳ HEAD
```

**Tip:** The index-filter takes less time in comparison to the tree-filter and can be used for removing files/directories. Also sometimes removing files/directories using the tree-filter does not always rewrite history.

For changing the directory structure the following commands can be used within a tree-filter

```
> mkdir -p <new directory>;git mv -k <file to move> <new directory>
```

For removing a file/directory the following command can be used within a index-filter

```
> git rm -r --cached --ignore-unmatch <file or directory>
```

**Note:** The order of the actions within the commands or executed filtering is very important. It is best to first move sources and finally removing files/directories, since the second filtering must use option -f (force) to be executed.

Removing garbage from filtering:

```
git reset --hard
git gc --aggressive
git prune
git clean -fd
```

Perfrom any necessary changes and commit these:

```
git status
git add .
git commit
```

Create a new branch in repository B:

```
cd $REPO_B_PATH
# create a new branch
git checkout -b feature/merge_git_repos
```

Create a remote connection to repository A as a branch in repository B:

```
git remote add repo-a $REPO_A_PATH
```

**Note:** repo-a can be anything - it's just a random name

Pull files and history from branches into repository B:

```
git pull repo-a master --allow-unrelated-histories
```

Remove the remote connection to repository A:

```
git remote rm repo-a
```

Finally, push the changes:

```
git push origin <new branch name>
```

### Migrating mercurial to git

Question: Can we merge a mercurial repository into existing git repository without losing history? For the migration of a mercurial to a git repository not much can be found. There are only a few different methods. I found a url where several migration paths to git have been defined (see [url](#)).

---

**Tip:** Within the following steps repository A will be migrated from a mercurial to a git repository. It is wise to create a temporary directory in which the mercurial repository will be cloned and migrated to a git repository.

---

---

**Note:** Most of the steps below are shown from a script used to execute the migration of a mercurial repository to a git repository, see `migrate_mercurial_to_git`. For the execution of the script the url of the mercurial repository needs to be provided as an argument.

```
usage() {  
    echo "migrate_mercurial_to_git <url mercurial repo>"  
    echo ""  
}
```

### Steps

Needed tooling for migration:

Clone tooling repository:

```
mkdir -p $MIGRATE_PATH  
cd $MIGRATE_PATH
```

```
git clone https://github.com/frej/fast-export.git  
cd fast-export  
git checkout v180317
```

---

**Tip:** During the first attempts I was getting some errors (unexpected "(" in `hg-fast-export.sh`, `hg-fast-export.py` cannot import name `revsymbol`). Eventually found the resolution to checkout a specific version of the tool.

---

Mercurial repository:

Clone the mercurial repository into temporary name `hg_repo` and collect committing users:

```
cd ..  
hg clone $1 $HG_REPO  
hg log | grep user: | sort | uniq | sed 's/user: *//' > $AUTHORS_FILE
```

Fix the authors file to reflect the correct syntax for users in git:

---

**Note:** Example for fixing the authors file: "bob"="Bob Jones <bob@company.com>" "bob@localhost"="Bob Jones <bob@company.com>" "bob <bob@company.com>"="Bob Jones <bob@company.com>" "bob jones <bob <AT> company <DOT> com>"="Bob Jones <bob@company.com>"

---

Create initial git repository and migrate mercurial repository to this:

```
git init $CONVERTED_PATH
cd $CONVERTED_PATH
$MIGRATE_PATH/fast-export/hg-fast-export.sh -r $HG_REPO_PATH -A $AUTHORS_FILE
```

```
git checkout HEAD
```

---

**Note:** Any next steps depend on what should be the actual result. If this is a new git repository the appropriate action for this need to be executed. In our case it had to be merged into an existing git repository. For that already a procedure has been created (*Merging Repositories*). This only needs to be adapted since repository A does not need to be cloned. For repository A the location <tmp\_dir>/converted can be used.

---

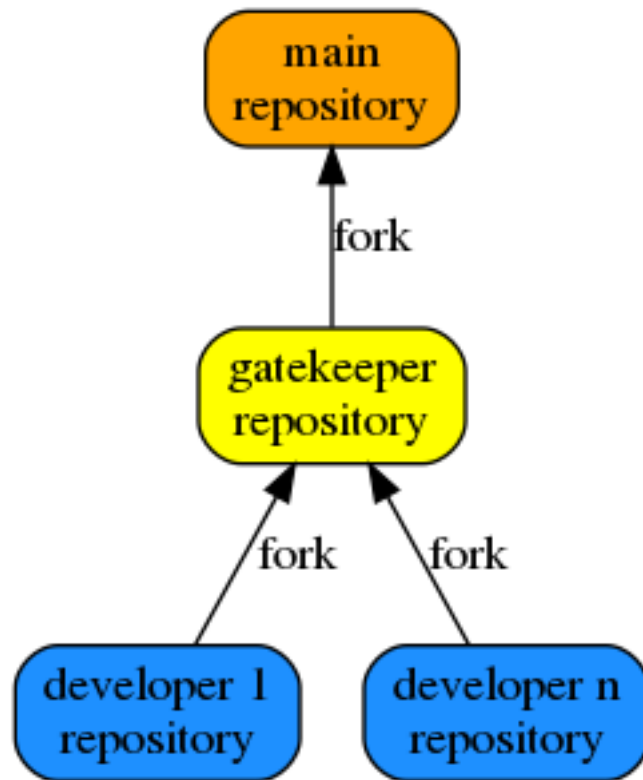
## Git workflow DDS-Demonstrators

There are several workflows for using git. We started with a centralized workflow. However a bottleneck was encountered in the time to be spent on the pull-requests towards the central repository. Any subsequent changes to the 'local' master influenced the outstanding pull-requests. This needed to be addressed, by finding a different workflow to be used where all the parties could continue.

The following workflows have been examined, [link](#):

- Integration-manager workflow
- Dictator and lieutenant workflow

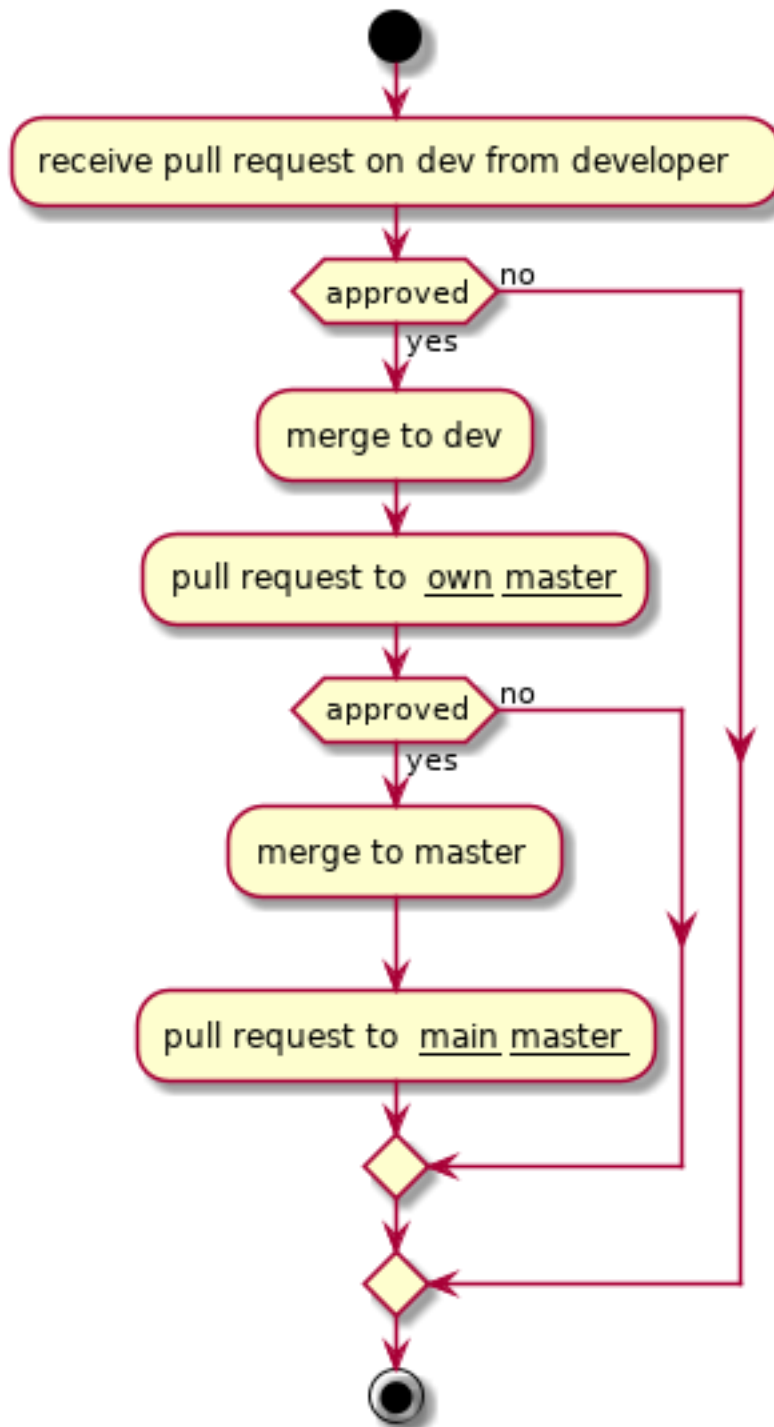
The result is a variant of both.

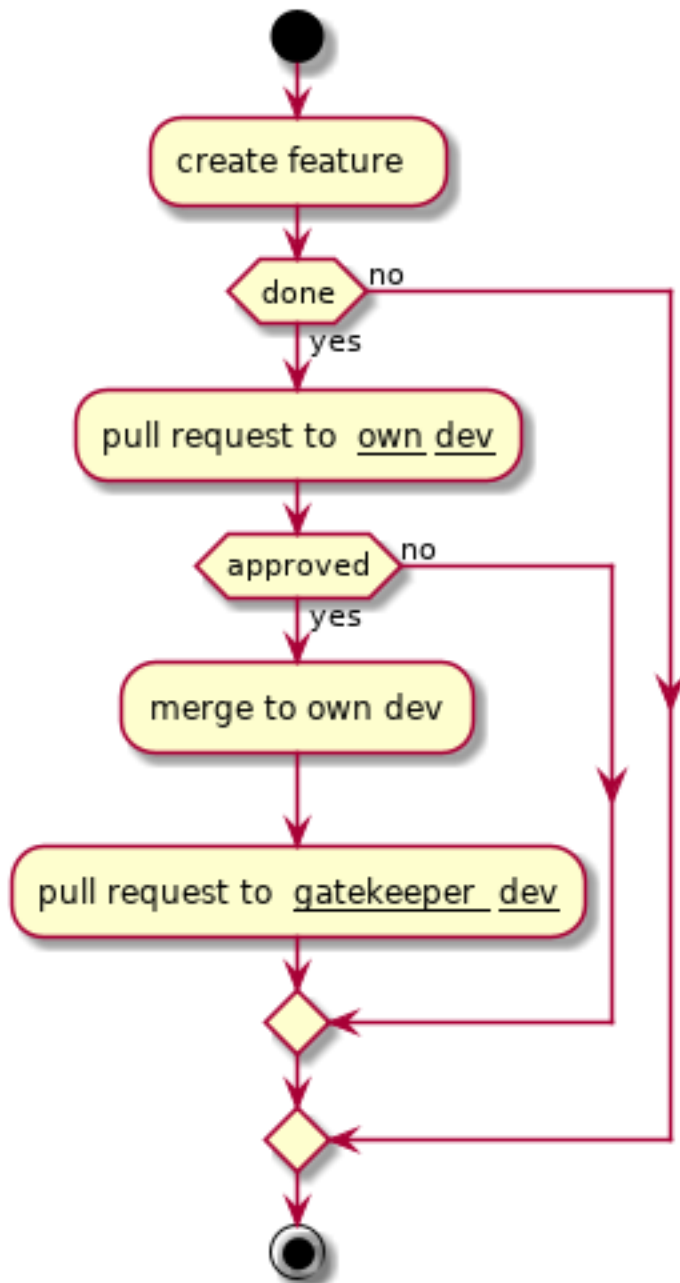


The gatekeeper forks the central repository and the developers fork the gatekeeper repository. For the gatekeeper and developers repository the **dev** branch is used for pushing/pulling changes using pull-requests. Only the gatekeeper will address the **master** branch on the central repository by means of pull-requests.

So, for all changes to be merged (independent of the repository) a pull-request is needed to uphold the needed quality for documentation and source. Because all developers fork the gatekeeper, they can benefit from each other and there would be no need to interchange documentation or source between their repositories and avoids any merge conflicts.

Some workflows:





### Team Demonstra-Tors

**authors** Sam Laan, Furkan Ali Yurdakul, Bendeguz Toth and Joost Baars

**date** Feb 2020





These team pages contain our **findings** during the project.

## DDS on bare-metal embedded hardware devices

**authors** Furkan Ali Yurdakul

**date** Feb 2020

### Possible solutions

Research was done concerning different solutions for using the DDS communication protocol in an embedded environment. Each result will be compared to the requirements listed underneath, these were discussed with the project owner.

Requirements:

- It must be easily accessible for an average user. For example: a known website, in stock and doesn't take more than a week to get it home.
- This solution will have to make use of the TCP/IP communication since this is the base of the DDS communication protocol.
- The hardware must be relatively cheap (max €50). For example: affordable for the average student.

To be able to do this, each solution will have a small summary to create an insight about it. Afterwards there will be a small hypothesis to see if the solution can be used to finish the project within a reasonable timeframe and fits the requirements. The estimated time to fully implement the solution will be given in days and is estimated by experience, knowledge and the amount of work it will need.

### SLIP/PPP

#### Summary of the solution

SLIP and PPP are known for implementing TCP/IP protocol in embedded hardware to communicate over serial ports. These protocols have been used a lot in the past and implementation enabled communication over TCP/IP in most cases. Because this is just about the communication and not the implementation of the DDS communication protocol, this would mean that research has to be performed about the calls that are used to send or receive within the DDS communication protocol.

PPP is a newer version of SLIP but also more complex to use. The complexity will need a lot more storage on your embedded hardware compared to SLIP. PPP also has more configurable options like authentication, compression and encryption. However, in this project all that's needed is a TCP/IP connection, so SLIP will be preferable in this case.

### Hypothesis

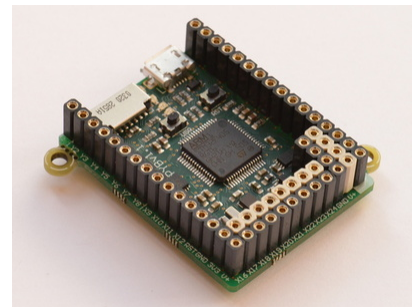
#### Reference

[SLIP SLIP Arduino implementation PPP](#)

This solution would be great to implement since it would make it possible to use DDS communication protocol in a wide variety of embedded hardware. These protocols are made to be used by everyone and the specifications are easy to find. Since it's a protocol which exists over a longer period, people can find many implementations for known embedded hardware like Arduino. These protocols on their own don't have any costs, only the hardware will be calculated in the cost. The estimated amount of days it will cost to finish this project using this solution is 60 days.

### MicroPython/Pyboard

#### Summary of the solution



MicroPython is an efficient implementation of the programming language Python 3 with the standard libraries and is optimized to run on embedded hardware, specifically for the Pyboard. It also contains hardware-specific modules to make use of different low-level hardware, for example: I2C and GPIO.

The Pyboard is a compact electronic circuit board that runs MicroPython on the bare metal, giving you a low-level Python operating system that can be used to control all kinds of electronic projects.

### Hypothesis

#### Reference

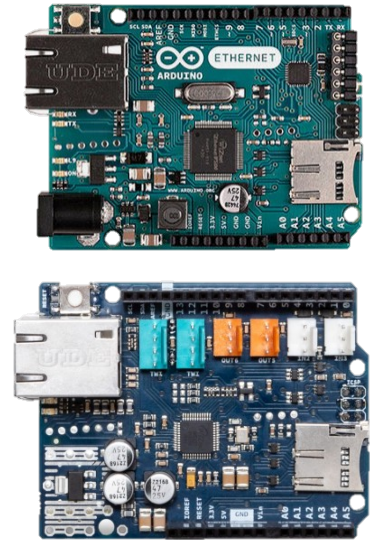
[Micropython/Pyboard ESP8266 Micropython](#)

It's already known that DDS can be implemented with different kind of programming languages including Python. In this case the Pyboard only misses a connection made by the TCP/IP protocol. The Pyboard is available in online stores and has a lot of resources to fully understand how it works. The Pyboard itself costs €59 which makes it slightly more expensive than the cost requirements. The original Pyboard can be used with SLIP/PPP or the Pyboard "ESP8266

Micropython” can be used. The estimated amount of days it will cost to finish this project using this solution is 70 days.

## Arduino + Ethernet

### Summary of the solution



Arduino has many products in-house. For this project, only the most relevant will be considered. First, there is the standard Arduino UNO Rev3. It is very likely that a programmer has this kind of board and therefore familiar with the programming language C/C++. The disadvantage of this board is that there is no Ethernet port. It can use the SLIP protocol as mentioned earlier, or an Arduino Ethernet Shield 2 can be used. This is an add-on for the standard Arduino, with which you can use an Ethernet port for communication.

Honorable mentions; Arduino Ethernet w/o PoE and Arduino Yun Rev 2. Arduino Ethernet is a retired product and Arduino Yun is a Linux based product. Therefore, these are not suitable for this project.

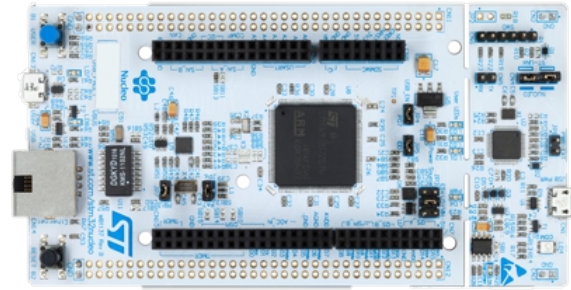
### Hypothesis

#### Reference

[Arduino Uno Rev3](#) [Arduino Ethernet Shield](#) [Arduino Ethernet w/o PoE](#) [Arduino Yun Rev 2](#)

On the forums of Arduino, it shows that SLIP has been implemented on Arduino. This makes it possible to use the standard Arduino without the ethernet shield. However, when using the ethernet shield time can be saved to investigate SLIP and its limitations. The Arduino Uno Rev3 costs 20 euro and the Ethernet Shield costs 25 euro, so it will cost 45 euros in total. The expected time for achieving the project with this solution is 60 days.

### Embedded RTPS - STM32



#### Summary of the solution

This is the only solution for which there is evidence that the DDS communication protocol has worked. This solution uses the microcontroller: STM32Nucleo-F767ZI. This Microcontroller has an Ethernet port at its disposal so there is no need for SLIP/PPP. According to the GitHub page, this microcontroller has the minimum to create a communication with a Linux device. According to this page, the 2 devices can send at least 10 messages back and forth.

#### Hypothesis

##### Reference

[GitHub STM32Nucleo-F767ZI](#)

Because this is a ready-made software package, it must first be investigated whether licenses are required for its use within this project. This concept solution will speed up the time to a result and more time can be spent on a well-made guide. The cost of this controller is 25 euros. It is expected that this can be achieved in 45 days.

#### Conclusion

When looking at the requirements, one is less noticeable but affects a project that is time-limited.

For hardware solutions that can't directly use TCP/IP communication, SLIP or PPP can be used. Because implementing SLIP or PPP will take too much time, according to the hypothesis, a separate Arduino or standard PyBoard can't be used. The Arduino + the Ethernet add on would be applicable, but is too expensive.

We also need to look at how realistic these solutions are. The sources indicate that the "Embedded RTPS - STM32" solution should be able to use DDS, which can be seen from the sources. After a short discussion, it was decided to use the Nucleo for the final result.

Check "*Nucleo Setup*" to learn how to set up the environment for this device.

#### Interface Description Language (IDL) explanation

**authors** Joost Baars

**date** Feb 2020

The IDL language is used for the general data types between different programming languages. This language is the layer between different programming languages and makes sure that the received message is formatted correctly for the wanted programming language. In the DDS applications, a message is described using IDL. An example for an IDL file with a message can be seen underneath.

```

module MatrixBoardData
{
    struct Msg
    {
        long userID; //@key
        long traffic;
        string message;
    };
    #pragma keylist Msg userID
};

```

This example creates a module where the wanted data types can be stored in. A struct is used for the message within the module. This message can be send using DDS. Three different variables are being stored in this message, two long types and one string type. These are general types which will be converted to a language specific type (string will for example be converted to a `char *` in C).

The `#pragma keylist Msg userID` defines the key of the message. A key groups the message into a certain category. In the DDS matrix board application, the key is the ID of a unique matrix board. This `#pragma` method is used for defining a key value within an OMG-DDS type structure. `long userID` also has `//@key` behind the variable. This is not used in the DDS application but is added for interoperability with the RTI definition.

## Links

- Information about defining the key within an IDL header: <https://stackoverflow.com/questions/31548067/what-does-pragma-keylist-keyword-do>

## Quality Of Service

**authors** Joost Baars

**date** Feb 2020

## Description

The Quality of Service (QoS) policies are a set of configurable parameters that control the behavior of a DDS system (how and when data is distributed between applications). Some of the parameters can alter the resource consumption, fault tolerance, or the reliability of the communication.

Each entity (reader, writer, publisher, subscriber, topic, and participant) of DDS has associated QoS policies to it. Some policies are only for one entity, others can be used on multiple entities.

More information about the QoS policies as well as a list of existing DDS policies can be found in the [Links](#).

## QoS findings

Different tests and measurements are done regarding the Quality of Service policies. The “General findings” explain some findings that were found while experimenting with the Quality of Service policies. There are some quick performance tests executed on the round trip and flood demonstrator (roundtrip demonstrator explanation: [RoundTrip](#)). The results (“Quick performance tests”) and the software of the application (“Quick performance tests software”) can be found in the list below. The “Matrix board performance measurements” are measurements with different QoS policies in the matrix board demonstrator (matrix board explanation: [Matrix Board Communication](#)). These measurements contain the connect/disconnect duration and resource usage. See the page below for more information.

### General findings

**authors** Joost Baars

**date** Feb 2020

### Description

This page shows the general findings regarding the QoS policies.

### Performance

The `dds_write` function is around twice as fast with a static message (a message that stays the same) compared to a message that is changing after each write. A struct was sent using DDS containing 3 parameters. It did not seem to matter how much data was changed in the struct. If the struct was changed (even only one byte), the execution time of the write function would be around twice as slow. The execution time of one write action was measured in this setup (the average of 10k measurements). The average result was around 8500 nanoseconds for a static message and around 16000 nanoseconds for a changing message. These measurements are not reliable though but it shows that there is a clear difference between the two configurations.

### Liveliness of DDS objects

The liveliness on a topic can be requested using a function call `dds_get_liveliness_changed_status()` if the liveliness callback QoS is configured with `NULL`. The liveliness can also be configured to call a callback function as soon as the liveliness changes. This liveliness callback is called when a writer connects to the topic (also if the writer is part of the same participant). The liveliness callback is not called for readers. Therefore, the number of active readers on a topic cannot be detected.

### CycloneDDS missing features

- The DDS extension for the QoS Database policy
- Asynchronous publisher does not seem to be implemented (possibly done automatically for you)

### Quick performance tests

**authors** Joost Baars

**date** Mar 2020

### Description

This page shows the results of the performance tests for QoS policies within DDS. The setup of the tests is added. The tests are not 100% reliable. These tests are not performed on a real-time OS.

Most of the tests are executed on the round trip application. The most thorough test is also executed on the flood application.

## Round trip

This chapter contains measurements of the QuickQoSMeasurements for different QoS configurations with the round trip application. The result of each measurement is in microseconds. This time is the time needed for one ping in the round trip. So if there are 3 devices in the round trip, and the round trip takes 600 microseconds. Then the result of the test would be  $600 / 3 = 200$  microseconds (per ping).

### Setup 1

Setup: 2 Raspberry Pi's communicating using DDS through a modem. The Pi's are connected using ethernet. The modem is not connected to the internet. A laptop starts the applications using SSH.

These results are in microseconds per ping. The settings are applied on the writer and the reader.

Settings	TEST 1	TEST 2
default	400	389
deadline	395	392
destination order reception	387	385
destination order source	386	387
durability persistent	389	385
durability transient	386	384
durability transient local	382	380
durability volatile	387	390
lifespan 10ms	398	399
lifespan 100ms	398	394
liveliness automatic	392	390
liveliness manual by participant	390	386
liveliness manual by topic	388	385
ownership shared	387	387
ownership exclusive	389	385
data lifecycle qos id	387	389
data lifecycle qos id	392	385
reliability best effort	358	355
reliability reliable	386	389
durability transient with durability service 1s	386	
durability transient with durability service 10ms	392	
history all	387	
history keep last	386	

Test 1: Default QoS settings with the tests Test 2: Reliability reliable default on

There is definitely a difference between the reliability configurations. There is a small overhead with the lifespan. Everything seems to have a small overhead. The durability transient does not have overhead compared to this one. The others do.

### Setup 2

These next measurements are performed with a laptop containing an Intel i5 8th generation processor with a Raspberry Pi 3 model B containing a fresh Raspbian Buster Lite (13-03-2020) install. The laptop was connected to a power outlet and did not have active background processes.

Both devices were connected to the network with an ethernet cable. The laptop had an USB 2.0 to Ethernet converter in between. There is around 17 meter of ethernet cable between the devices. Additionally, a router is also in between the 2 devices. The router is connected to the internet. Additionally, the router is used by other devices for internet too. Therefore, it could cause some overhead.

The test that was executed: the “slave” of the round trip was running on the Raspberry Pi. The “master” was running on the laptop. There only were 2 devices in the round trip. Each measurement was the average of 500 round trips.

<b>settings</b>	<b>TEST 1</b>	<b>TEST 2</b>	<b>TEST 3</b>
default	512	521	508
deadline	540	542	531
destination order reception	525	538	525
destination order source	521	534	533
durability persistent	514	543	537
durability transient	534	536	538
durability transient local	513	539	519
durability volatile	524	521	539
lifespan 10ms	522	540	535
lifespan 100ms	534	535	508
liveliness automatic	534	528	542
liveliness manual by participant	518	543	536
liveliness manual by topic	519	529	514
ownership shared	532	536	540
ownership exclusive	535	523	540
data lifecycle qos id	535	540	540
data lifecycle qos id	499	533	540
reliability best effort	521	540	529
reliability reliable	535	534	537
durability transient with durability service 1s	523	530	519
durability transient with durability service 10ms	535	533	536
history all	535	536	536
history keep last	531	534	509

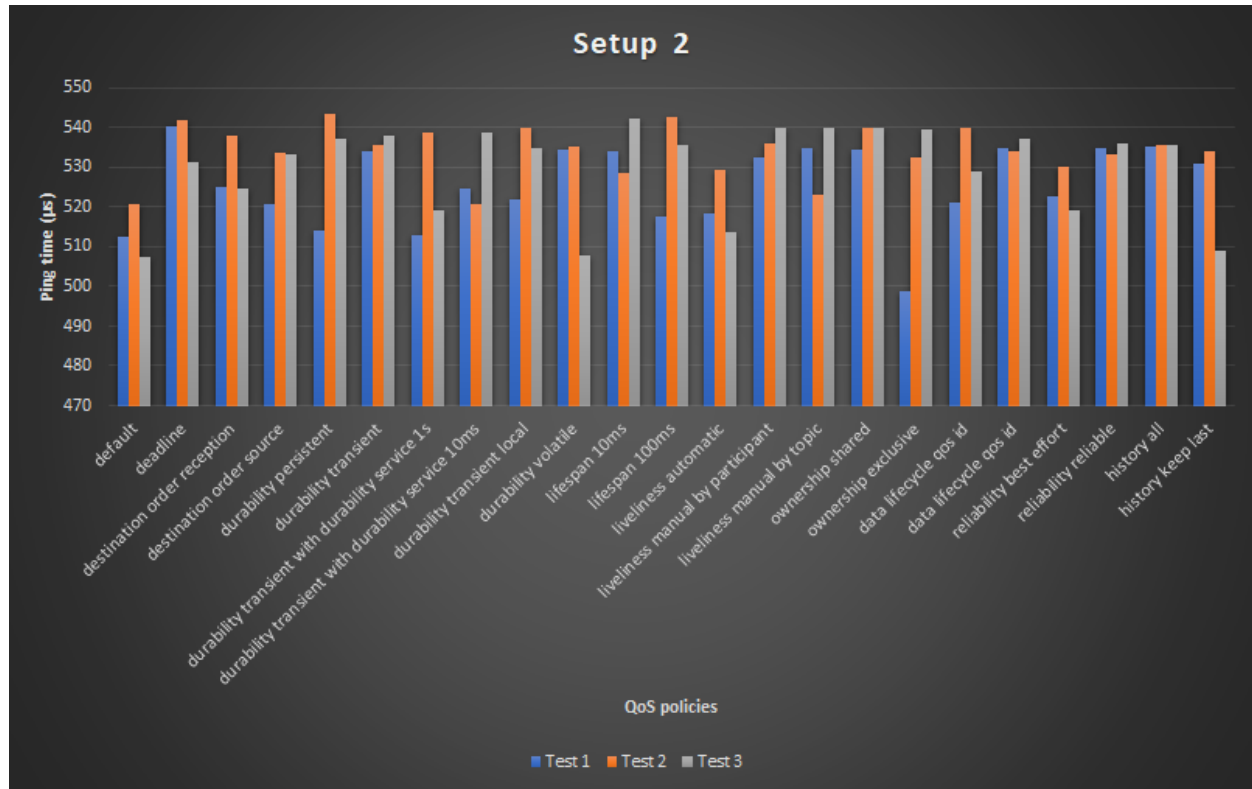
Test 1: Default configuration

Test 2: Every test has RELIABILITY\_BEST\_EFFORT except for the reliability reliable

Test 3: Every test has RELIABILITY\_RELIABLE except for the reliability best effort

The data from the table is visualized in a graph. The graph below shows the three test columns from the table.





Interesting results: Between test 2 and 3, the differences are not noticeable. Some tests even went noticeably faster on test 3. Although, I would expect that test 3 would be noticeably slower compared to test 2. Because reliability reliable should be slower and is mostly slower when directly compared. This difference could be due to background processes that interfered with the tests.

Just for reference, I compared the performance on ethernet with the performance on WiFi. Time with both devices on ethernet: ~1200 (lot of deviation, measurement of ~1700 and ~1000) microseconds. Time with the RPi on ethernet and the laptop on WiFi: ~525 microseconds (measurement of ~500 and ~540, much less deviation)

### Setup 3

This setup is the same as setup 2. But this setup contains (much) more measurements.

settings	TEST 1	TEST 2	TEST 3	TEST 4	TEST 5	TEST 6	TEST 7	AV
default	539	534	508	528	526	525	515	525
deadline	537	540	508	532	502	530	535	526
destination order reception	538	518	525	521	500	500	525	518
destination order source	531	534	508	519	526	518	503	520
durability persistent	533	529	525	527	491	533	509	521
durability transient	535	514	531	528	520	533	479	520
durability transient with durability service 1s	535	508	523	531	529	531	510	524
durability transient with durability service 10ms	524	532	525	527	530	502	506	521
durability transient local	531	502	519	520	520	533	528	522
durability volatile	537	530	498	522	524	529	501	520
lifespan 10ms	520	536	506	522	529	538	528	525
lifespan 100ms	538	519	522	511	532	517	545	520

Table 1 – continued from previous page

liveliness automatic	529	518	506	506	507	512	513	513
liveliness manual by participant	538	534	527	524	522	531	530	529
liveliness manual by topic	539	534	521	504	528	528	527	520
ownership shared	563	531	523	524	528	529	503	529
ownership exclusive	537	537	527	523	528	532	519	529
data lifecycle qos id	533	508	500	524	528	523	533	521
data lifecycle qos id	512	467	502	520	526	529	531	512
reliability best effort	508	498	506	518	528	518	524	514
reliability reliable	533	532	506	506	533	526	500	519
history all	532	505	517	506	493	510	516	511
history keep last	536	522	525	505	526	526	507	521
latency budget 1000ms	539	524	525	524	500	534	527	525
latency budget 10ms	534	508	527	520	500	524	526	520
latency budget 10ms	535	534	504	502	494	524	528	517
presentation topic	525	534	526	512	507	499	529	519
presentation group	535	535	500	517	509	525	520	520
presentation instance	537	505	530	514	500	505	526	516
resource limits	528	532	525	488	522	516	531	520
time based filter 1s	538	534	499	525	528	525	533	520
time based filter 10ms	531	526	515	524	495	528	528	521
transport priority of 1	536	498	521	524	529	518	531	522
transport priority of 100	517	511	524	525	511	526	528	520
writer data lifecycle true	506	535	529	509	529	526	530	523
writer data lifecycle false	537	529	525	522	520	525	519	525
partition	537	511	521	521	527	537	534	527
<b>AVERAGE</b>	<b>532</b>	<b>522</b>	<b>517</b>	<b>518</b>	<b>517</b>	<b>523</b>	<b>521</b>	<b>AV</b>
<b>AVERAGE 2</b>	<b>523</b>	<b>527</b>	<b>520</b>	<b>510</b>	<b>517</b>	<b>522</b>	<b>520</b>	

Test 1: Default configuration

Test 2: Every test has reliability reliable

Test 3: Every test has reliability best effort

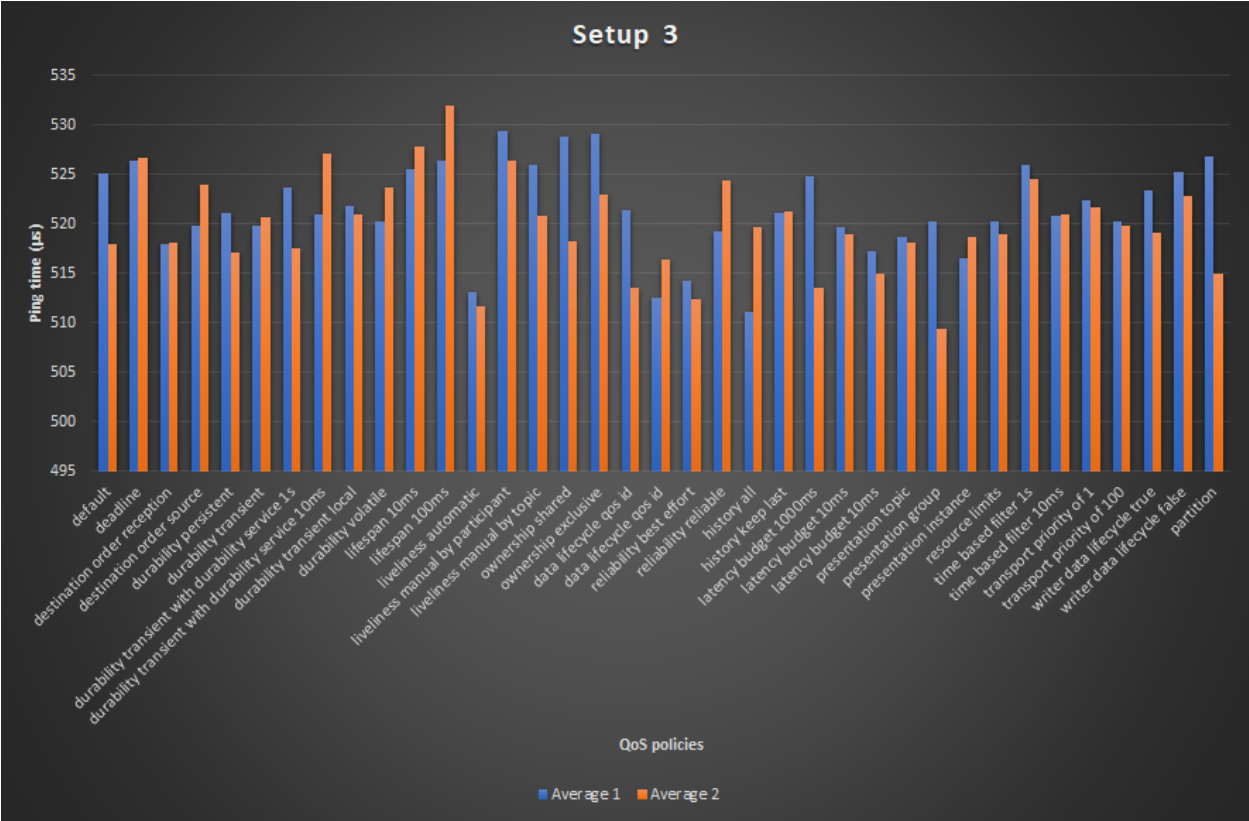
Test 4: Every test has reliability best effort with DDS durability transient local

Test 5: Every test has reliability best effort with DDS durability volatile

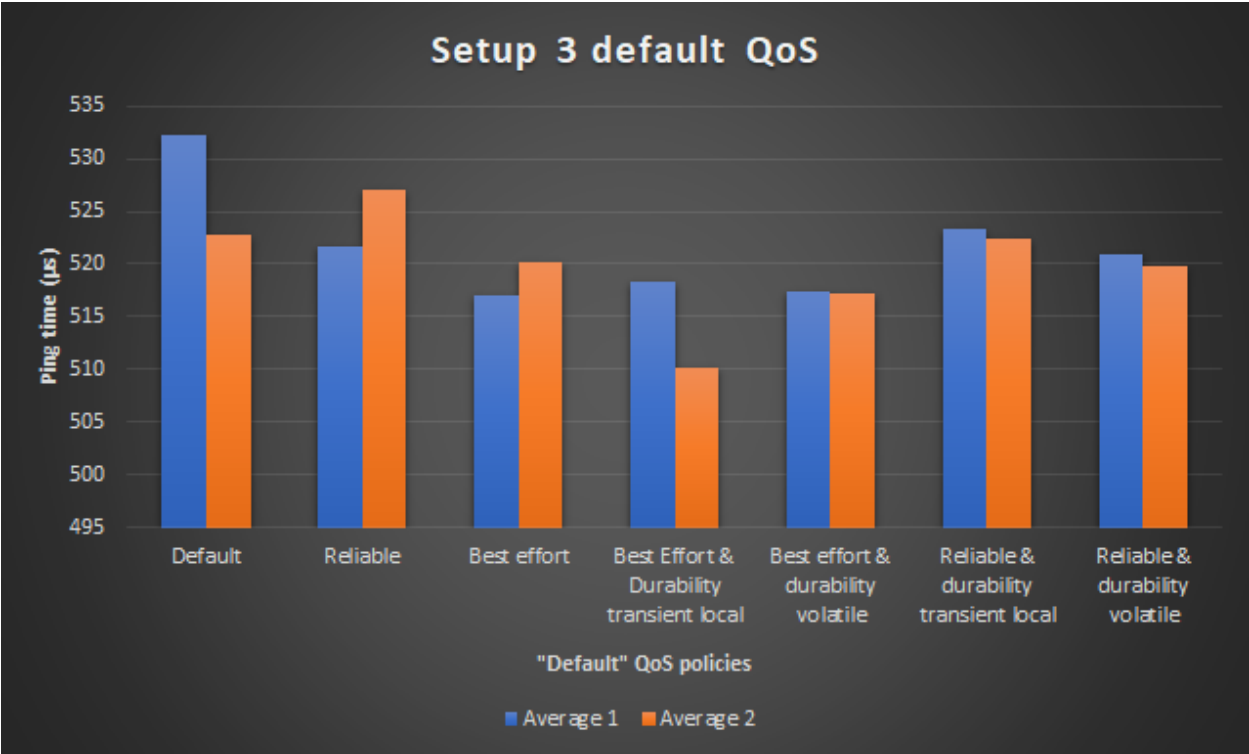
Test 6: Every test has reliability reliable with DDS durability transient local

Test 7: Every test has reliability reliable with DDS durability volatile

The data from the table is visualized in graphs. The graph below shows the two average columns from the table.



The graph below shows the two average rows from the table.



The AVERAGE 2 are the average measurements of a second execution of the same measurements. These are included for comparing accuracy.

The graphs clearly show a lot of deviation between the average measurements. The average of the first test should be about the same as the average of the second test. The biggest deviation is about 2.3% or 12 microseconds.

Conclusions between the tests:

The reliability reliable is clearly a bit slower compared to the reliability best effort. The durability kinds do not seem to differ a lot. It seems that durability transient local is a bit slower. This can also be because the tests are not accurate enough (if you compare the results with `AVERAGE 2`).

Conclusion between different QoS policies:

- Lifespan causes some noticable overhead.
- Liveliness automatic seems to be faster compared to manual liveliness.
- Ownership also seems to give some overhead.
- History keep all/keep last, keep last seems to be slower.

For the rest, I don't see really interesting data.

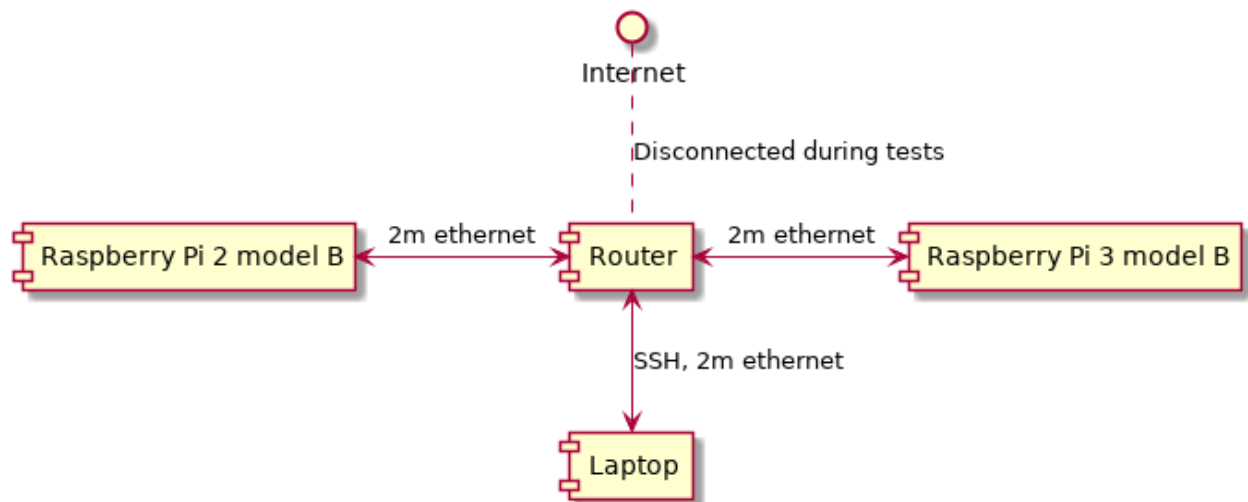
### Setup 4

This setup focusses on a reliable test without much deviation. This is not 100% possible because the OS used is not a real-time OS.

This setup uses 2 Raspberry Pi's with each a clean install of Raspbian Buster 13-03-2020. The Raspberry Pi's used are a Raspberry Pi 2 model B and a Raspberry Pi 3 model B.

These Raspberry Pi's only have the necessary dependencies installed.

The setup can be found in the image below. All devices are connected using ethernet cables.



This setup contains a controlled network with no influence of other devices. The laptop is connected to both devices using SSH. This laptop also executed the application on the Raspberry Pi's (by SSH).

The Raspberry Pi 3 model B plus was the master with ID = 1.

settings	1	2	3	4	5	6	7	8	9	10	11
default	355	324	327	310	332	347	328	340	333	334	0

Continued on next page

Table 2 – continued from previous page

deadline 10s	346	346	338	334	337	345	364	346	<b>344</b>	<b>341</b>	<b>11</b>
deadline 10ms	344	348	340	329	343	346	346	346	<b>343</b>	<b>343</b>	<b>10</b>
destination order reception	338	328	327	312	328	325	324	349	<b>329</b>	<b>330</b>	<b>-4</b>
destination order source	339	334	334	318	329	325	334	342	<b>332</b>	<b>332</b>	<b>-1</b>
durability persistent	337	329	333	332	329	324	329	347	<b>332</b>	<b>333</b>	<b>-1</b>
durability transient	344	329	336	329	334	329	336	344	<b>335</b>	<b>334</b>	<b>2</b>
durability transient with durability service 1s	343	328	330	332	327	327	327	348	<b>333</b>	<b>331</b>	<b>0</b>
durability transient with durability service 10ms	343	335	333	335	323	333	333	342	<b>335</b>	<b>334</b>	<b>2</b>
durability transient local	334	329	323	323	317	327	330	333	<b>327</b>	<b>323</b>	<b>-6</b>
durability volatile	344	334	336	331	337	323	328	337	<b>334</b>	<b>331</b>	<b>1</b>
lifespan 10ms	357	349	346	329	343	349	347	354	<b>347</b>	<b>346</b>	<b>14</b>
lifespan 100ms	347	345	339	331	346	345	351	352	<b>344</b>	<b>349</b>	<b>11</b>
liveliness automatic 10ms	352	334	339	328	341	333	331	345	<b>338</b>	<b>335</b>	<b>5</b>
liveliness manual by participant 10ms	343	333	335	320	339	333	327	343	<b>334</b>	<b>333</b>	<b>1</b>
liveliness manual by topic 10ms	339	326	338	316	337	331	338	346	<b>334</b>	<b>334</b>	<b>1</b>
ownership shared	345	328	328	312	333	323	326	343	<b>330</b>	<b>333</b>	<b>-3</b>
ownership exclusive	342	334	337	311	331	323	330	347	<b>332</b>	<b>331</b>	<b>-1</b>
data lifecycle qos id 1s	341	324	333	315	334	327	329	349	<b>332</b>	<b>331</b>	<b>-1</b>
data lifecycle qos id 10ms	334	328	327	317	330	327	328	341	<b>329</b>	<b>331</b>	<b>-4</b>
reliability best effort	335	330	329	322	332	323	333	333	<b>329</b>	<b>330</b>	<b>-4</b>
reliability reliable	334	334	333	333	327	329	335	338	<b>333</b>	<b>328</b>	<b>0</b>
history all	346	326	325	319	335	350	329	340	<b>334</b>	<b>331</b>	<b>1</b>
history keep last	343	331	332	316	329	328	333	343	<b>332</b>	<b>333</b>	<b>-1</b>
latency budget 1000ms	336	326	324	311	329	327	323	340	<b>327</b>	<b>331</b>	<b>-6</b>
latency budget 10ms	342	326	330	316	336	326	334	347	<b>332</b>	<b>331</b>	<b>-1</b>
presentation topic	336	329	337	323	329	328	331	340	<b>332</b>	<b>335</b>	<b>-1</b>
presentation group	346	334	331	315	333	335	345	340	<b>335</b>	<b>331</b>	<b>2</b>
presentation instance	344	324	333	316	331	327	332	339	<b>331</b>	<b>329</b>	<b>-2</b>
resource limits	347	332	334	319	334	324	333	339	<b>333</b>	<b>334</b>	<b>0</b>
time based filter 1s	336	324	324	316	326	327	329	348	<b>329</b>	<b>332</b>	<b>-4</b>
time based filter 10ms	346	335	329	319	330	328	326	340	<b>331</b>	<b>331</b>	<b>-2</b>
transport priority of 1	345	335	334	317	328	334	350	338	<b>335</b>	<b>333</b>	<b>2</b>
transport priority of 100	343	334	323	320	324	322	336	344	<b>331</b>	<b>331</b>	<b>-2</b>
writer data lifecycle true	343	328	326	312	329	326	335	347	<b>331</b>	<b>331</b>	<b>-2</b>
writer data lifecycle false	342	327	337	312	327	328	338	348	<b>332</b>	<b>333</b>	<b>-1</b>
partition	345	327	328	311	333	324	328	340	<b>330</b>	<b>332</b>	<b>-3</b>
high performance combination	318	311	323	317	316	316	316	315	<b>316</b>	<b>317</b>	<b>-17</b>
high overhead combination	354	357	358	355	354	353	357	358	<b>356</b>	<b>352</b>	<b>23</b>
<b>AVERAGE</b>	<b>342</b>	<b>332</b>	<b>332</b>	<b>321</b>	<b>332</b>	<b>331</b>	<b>334</b>	<b>343</b>			
<b>AVERAGE 2</b>	<b>343</b>	<b>331</b>	<b>332</b>	<b>321</b>	<b>332</b>	<b>333</b>	<b>332</b>	<b>342</b>			

- 1: Test 1: Default QoS policies
- 2: Test 2: Reliability reliable
- 3: Test 3: Reliability best effort
- 4: Test 4: Reliability best effort + durability transient local
- 5: Test 5: Reliability best effort + durability volatile
- 6: Test 6: Reliability reliable + durability transient local
- 7: Test 7: Reliability reliable + durability volatile

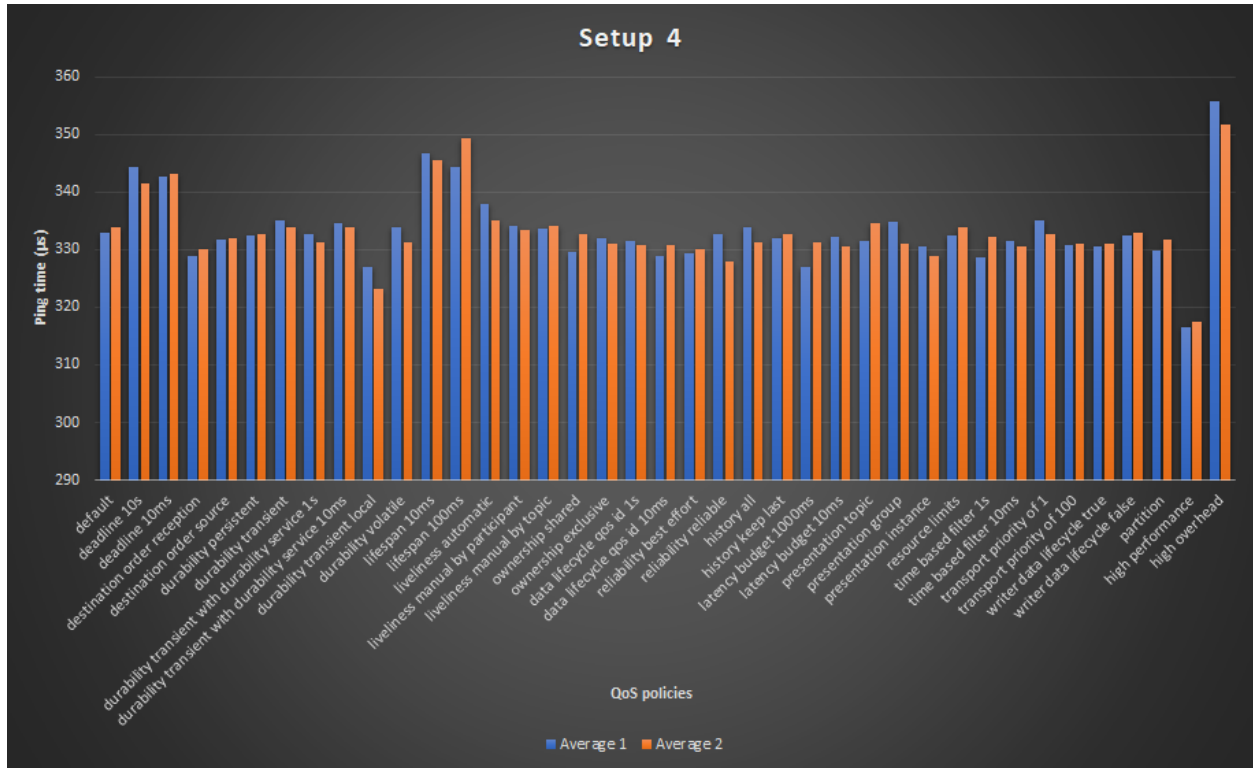
8: Test 8: Default QoS policies

9: Average of the shown measurements

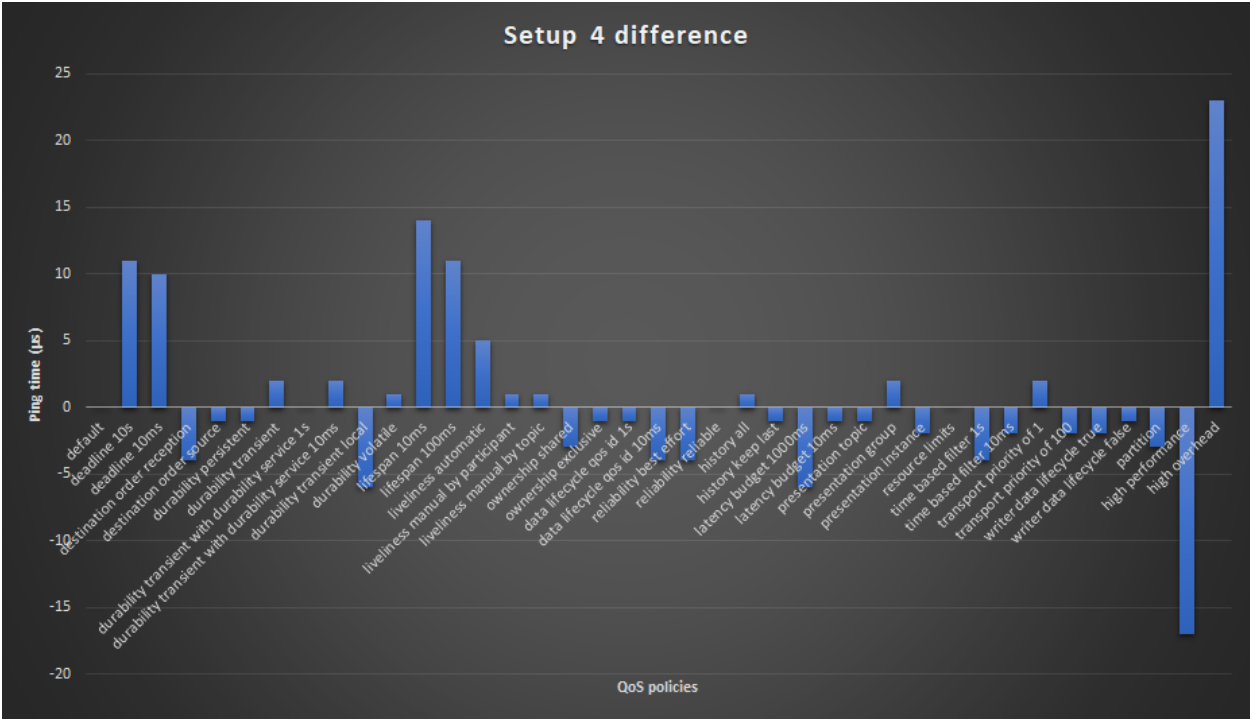
10: Average of a second measurement with the same tests (for accuracy)

11: The difference between the default (top) value and the first average (column 9)

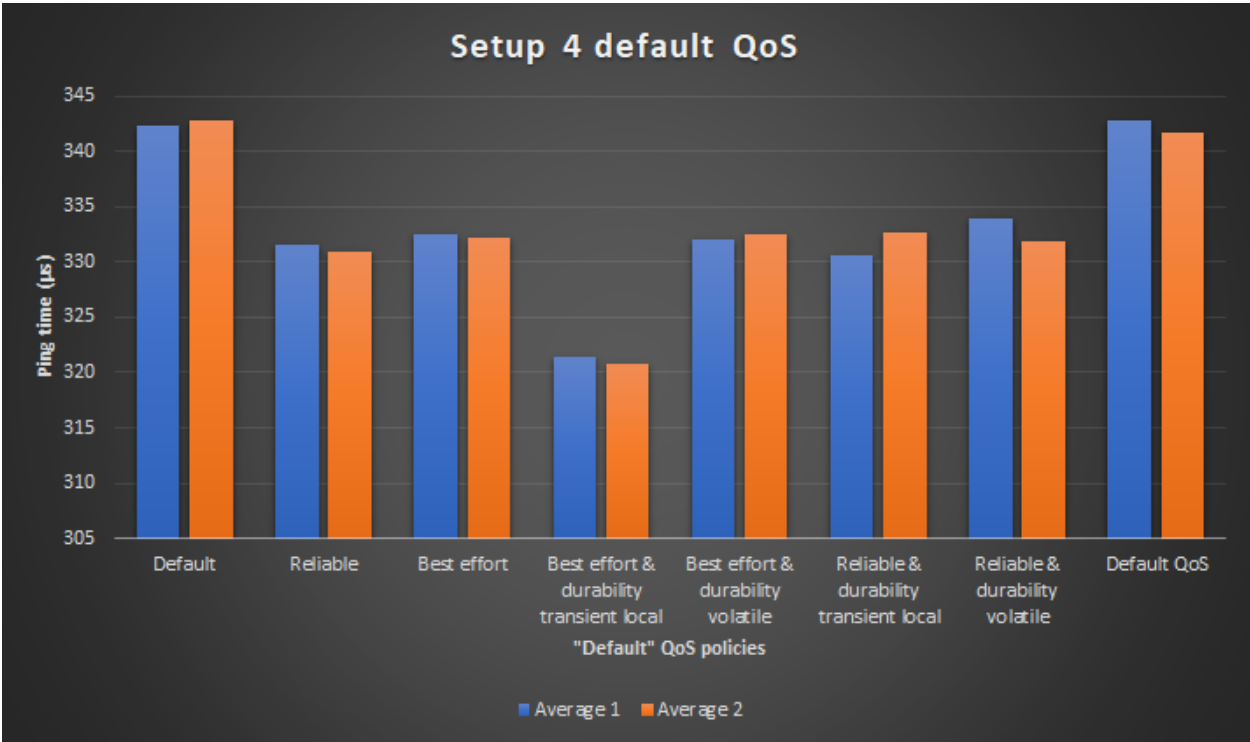
The data from the table is visualized in graphs. The graph below shows the two average columns from the table. The accuracy of the measurements is the difference between Average 1 and Average 2.



The difference row can be seen in the graph below. This graph clearly shows what QoS policies are faster / slower compared to the default QoS.



The graph below shows the two average rows from the table. The accuracy of the measurements is the difference between Average 1 and Average 2.



The AVERAGE 2 are the average measurements of a second execution of the same measurements. These are included for comparing accuracy.

**The high performance combination from the tests:**

- History keep last
- Reliability best effort
- Ownership shared
- Durability transient local
- Presentation instance

### The high overhead combination from the tests:

- History keep all
- Reliability reliable
- Ownership exclusive
- Lifespan 10ms
- Deadline 10ms
- Liveliness automatic 10ms
- Latency budget 1s

The graphs clearly show that there is not a lot of deviation between the average measurements. This is definitely the case if you compare it with [Setup 3](#). The worst deviation is 5 microseconds or 1.4%. Which is around 40% accuracy gain compared to [Setup 3](#).

Most of the measurements don't have interesting results. There are a few measurements with noticable overhead.

### Interesting results from column 9 and 10:

- deadline 10s (noticable overhead)
- lifespan (10 and 100ms) (noticable overhead)
- durability transient local (better performance)
- Liveliness automatic (small overhead)

### Interesting results from the AVERAGE rows:

- Test 1 and 8: Much overhead for the default QoS policies
- Test 4: Performance gain

It would be interesting to see why the default QoS policies cause more overhead compared to other configurations.

The **liveliness** has a minor overhead compared to the other measurements. I would expect more overhead because DDS instances must ping every 10ms that they are alive.

The **lifespan** is the lifespan of a message. Therefore, messages that exceed the lifespan are removed from the DDS network. These checks cause noticable overhead. 10 or 100ms lifespan does not seem to matter a lot. RTI documentation says that this QoS policy should not significantly impact the use of resources.

The **deadline** is the maximum time between writing a new value for the writer and between reading new messages for the reader. These checks seem to cause rather much overhead.

For the **durability**, according to RTI's documentation, the durability volatile should cause the least overhead. That is because this one does not save / deliver old samples. Transient local saves and delivers messages as long as the writer exists. This means that the implementation of saving probably causes less overhead compared to not saving (or removing?).

I would expect that the **reliability** QoS would be noticable. But during these tests, there is no noticable difference. The reliability QoS policy configures if messages should be reliably received. If so, it checks if the message is correctly received by the reader. If the message is not correctly received, then it is resend. Therefore, I would expect



that this would cause noticable overhead like *Setup 1*. *Setup 1* is less reliable though because of less measurements. Additionally, it could differ because of the set-up (different routers, different OS).

I think the other QoS policies don't have to cause overhead. So those results are all as expected.

## Flood

The same test is performed with flood instead of round trip. This test was a bit harder to execute because some QoS configurations could not reliably be executed.

The numbers that are given are in microseconds per message for the flood. This test was executed with 2 different devices in the flood loop.

**The following tests were removed compared to the round trip:**

- Reliability best effort (unreliable communication what resulted in lost messages)
- Lifespan 10ms (Some messages took longer to be received and therefore were removed, what resulted in lost messages)
- History keep last (This one would probably still work, but was removed because messages may not be lost) (this one was still active in the high performance test)
- Resource limits (the resource limits was set-up so only the last 5 messages in the buffer exist. The rest was removed. This resulted in lost messages)

The test setup was the same as *Setup 4*. The only difference was that the Raspberry Pi 2 model B was the master instead of the Raspberry Pi 3 model B.

settings	TEST 1	TEST 2	TEST 3	AVERAGE 1	AVERAGE 2	AVERAGE 3
default	154.3	104.3	154.8	<b>137.8</b>	<b>166.4</b>	<b>134.2</b>
deadline 10s	105	107.3	107.9	<b>106.8</b>	<b>120.9</b>	<b>104</b>
deadline 10ms	124.1	127.8	126	<b>126</b>	<b>148.9</b>	<b>145.5</b>
destination order reception	103.7	177.4	116.6	<b>132.6</b>	<b>101</b>	<b>105.1</b>
destination order source	106.4	105.4	106.8	<b>106.2</b>	<b>109.8</b>	<b>105.5</b>
durability persistent	168.5	103.8	152.3	<b>141.6</b>	<b>117.9</b>	<b>133.3</b>
durability transient	106	106.4	104	<b>105.5</b>	<b>116.5</b>	<b>106.3</b>
durability transient with durability service 1s	105.6	105.7	106.3	<b>105.9</b>	<b>121</b>	<b>104.6</b>
durability transient with durability service 10ms	105.8	104.3	102.9	<b>104.3</b>	<b>118.7</b>	<b>120.5</b>
durability transient local	105.1	175.7	105.5	<b>128.8</b>	<b>101.5</b>	<b>101.6</b>
durability volatile	105.7	106.1	105.5	<b>105.8</b>	<b>140</b>	<b>133</b>
lifespan 100ms	162.9	107.7	178.5	<b>149.7</b>	<b>103.6</b>	<b>122.8</b>
liveliness automatic 10ms	105.2	178.5	103.4	<b>129.1</b>	<b>118.4</b>	<b>108.4</b>
liveliness manual by participant 10ms	107.3	109.2	107.5	<b>108</b>	<b>121.8</b>	<b>119.2</b>
liveliness manual by topic 10ms	107.8	108.4	107.1	<b>107.7</b>	<b>103.6</b>	<b>102.8</b>
ownership shared	105.5	114.8	104.8	<b>108.4</b>	<b>116.6</b>	<b>117.2</b>
ownership exclusive	105.2	110.8	105.8	<b>107.3</b>	<b>129.4</b>	<b>102.1</b>
data lifecycle qos id 1s	154.2	155.4	154.3	<b>154.6</b>	<b>108.1</b>	<b>122.7</b>
data lifecycle qos id 10ms	177.8	106.6	151.8	<b>145.4</b>	<b>125.4</b>	<b>114.5</b>
reliability reliable	114.7	106.5	111.8	<b>111</b>	<b>101</b>	<b>104.1</b>
history all	104.4	114.3	108.2	<b>109</b>	<b>116.5</b>	<b>113.1</b>
latency budget 1000ms	112.7	115.9	106.7	<b>111.8</b>	<b>116</b>	<b>101.3</b>
latency budget 10ms	225.3	157	120.1	<b>167.5</b>	<b>100.3</b>	<b>128.6</b>
presentation topic	105.5	117.1	154.4	<b>125.7</b>	<b>104.1</b>	<b>132.4</b>
presentation group	112.9	105.8	103.3	<b>107.3</b>	<b>125.1</b>	<b>100.3</b>

Continued on next page

Table 3 – continued from previous page

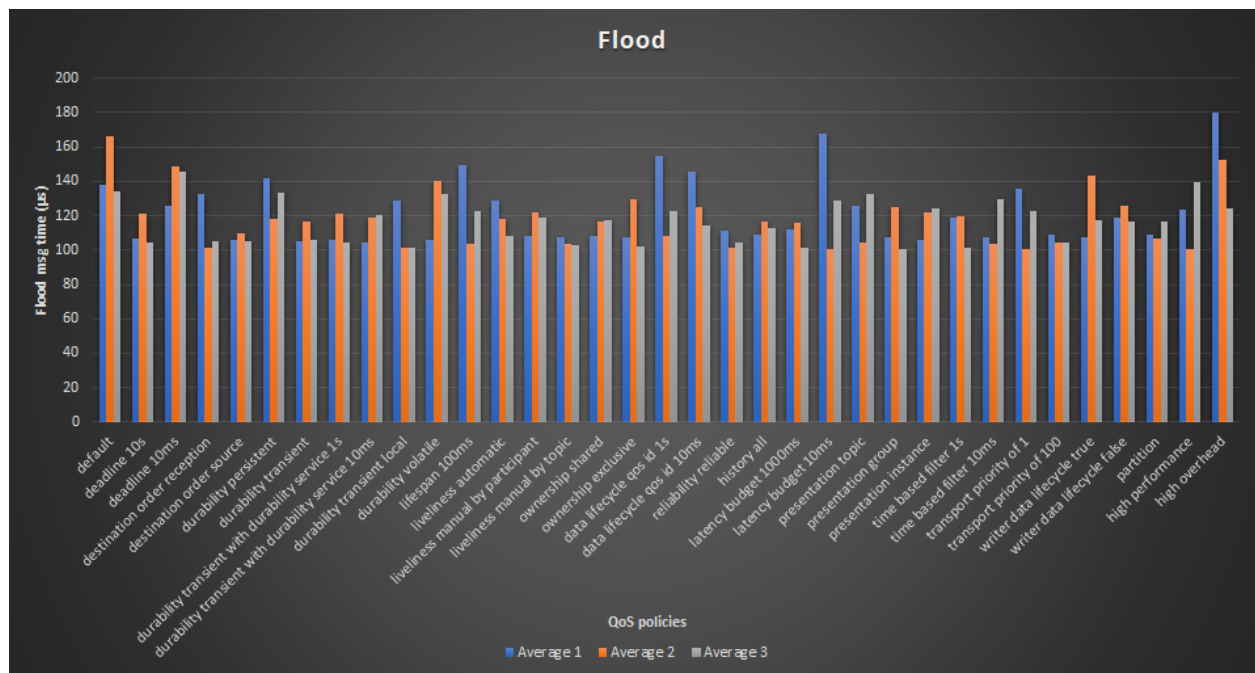
presentation instance	105.7	105.7	105.6	<b>105.7</b>	<b>121.8</b>	<b>124.3</b>
time based filter 1s	105.5	146.4	105.5	<b>119.1</b>	<b>119.3</b>	<b>101.7</b>
time based filter 10ms	114.9	104.4	104.2	<b>107.8</b>	<b>103.8</b>	<b>129.9</b>
transport priority of 1	153.2	105.3	149.7	<b>136.1</b>	<b>100.4</b>	<b>122.5</b>
transport priority of 100	105.9	106.5	113.5	<b>108.6</b>	<b>104.4</b>	<b>104.3</b>
writer data lifecycle true	104.7	112.4	104.8	<b>107.3</b>	<b>143.4</b>	<b>117.4</b>
writer data lifecycle false	105.3	145.7	105.7	<b>118.9</b>	<b>125.8</b>	<b>116.6</b>
partition	112.7	107.8	106.7	<b>109.1</b>	<b>106.5</b>	<b>116.4</b>
high performance combination	112.4	112.5	145.5	<b>123.4</b>	<b>100.4</b>	<b>139.5</b>
high overhead combination	255.3	146	139.5	<b>180.3</b>	<b>152.6</b>	<b>124.6</b>
<b>AVERAGE 1</b>	<b>124.8</b>	<b>120.7</b>	<b>119.6</b>			
<b>AVERAGE 2</b>	<b>120.6</b>	<b>116.5</b>	<b>117</b>			
<b>AVERAGE 3</b>	<b>116.2</b>	<b>116.6</b>	<b>117</b>			

Test 1: Reliability reliable

Test 2: Reliability reliable with durability transient local

Test 3: Reliability reliable with durability volatile

The average columns are visualized in the graph below.



**AVERAGE 1** is the average of the non-average values that are in the table.

**AVERAGE 2 and 3** are results of different executions of the same test. They are added for accuracy comparison.

There is a lot of deviation between the average results. Mainly for the accuracy of the rows. This can easily be seen in the graph (the Average 1, Average 2 and Average 3 should be around the same height). The biggest deviation is about 67 microseconds or about 60%! For that reason, these measurements are difficult to reliably analyze.

## Quick performance tests software

**authors** Joost Baars

**date** Mar 2020

### Description

This page contains information about the quick performance tests that were implemented for the round trip and flood in C++. The goal of these measurements was to analyze the performance of different QoS policies in various configurations.

The software for the quick performance tests can be found in the directory `src/demonstrators/RoundTrip/C++/PerformanceMeasurements/QuickQoSMeasurements/`.

### Dependencies

The dependencies necessary for succesful compilation of the quick performance measurements in C++ are the following:

- CMake
- C++17 compiler (Gcc 7+ for example)
- Cyclone DDS library

### Configuration

The performance tests implementation contains an implementation for the round trip application and one for the flood application. One of these implementations can be chosen before building the project.

The implementation can be configured by altering the following line in the `CMakeLists.txt`:

```
# Choose the wanted configuration
## Config parameter: can be configured for building the wanted performance test
## This can be "roundtrip" or "flood"
set(PROJECT roundtrip)
```

This line can be found at the top of the `CMakeLists` file. The `roundtrip` can be changed to `flood` for the flood implementation.

### Building

The software can be build by executing the following commands in the command line (you should be in the main directory of the application! See [Description](#) for the directory)

```
mkdir build && cd build
cmake ..
make
```

These commands compile and create the executable for the performance measurements.

### Execution

The application must be executed using various parameters. The application can be executed using the following command:

```
./<Application name> <device ID> <number of devices> <total messages / round_
↳trips> <filename for storing measurements>
```

<Application name>: The application you want to run, this can be

performance\_measurements\_flood or performance\_measurements\_roundtrip

<device ID>: The ID of the device, must be unique, > 0 and <= total devices

<Total devices>: The amount of devices in the loop (the amount of applications running)

<Total msg / round trips>: For the flood application: Total messages send. For the round trip: Number of round trips to execute

<filename>: Optional, can be configured to change the default measurements file name. By default: measurements.csv.

The results are comma separated and appended to the file. Therefore, the measurements are not nicely formatted in the file.

The following example starts 4 nodes for the round trip. The slaves of the round trip are started in the background. Only the master is started in the foreground in this example. Each device pings a total of 1000 times. Therefore, there are 1000 round trips. The measurements of this test are stored in the file test1.csv

```
./performance_measurements_roundtrip 2 4 1000 & ./performance_measurements_
↳roundtrip 3 4 1000 & ./performance_measurements_roundtrip 4 4 1000 &
./performance_measurements_roundtrip 1 4 1000 test1.csv
```

A description of the parameters can also be found when executing the application with no or incorrect parameters.

---

**Note:** Execution of the program

The round trip is initiated by the device with <device ID> = 1. Therefore, <device ID> = 1 should always be started after the other ID's are started.

The devices added with <device ID> must be an increment of the previous one. This function does not dynamically search for the next device! So if there are three devices, device 2 and 3 should be started first. Afterwards, device 1 can be started.

The <total msg / round trips> parameter should be the same for each application.

---

### Software

The application is written with customizability and ease of use in mind. Therefore, it is rather easy to create new tests. This chapter explains how tests can easily be added to the software. The round trip won't be explained, because it is already explained in the *Round trip in C++*.

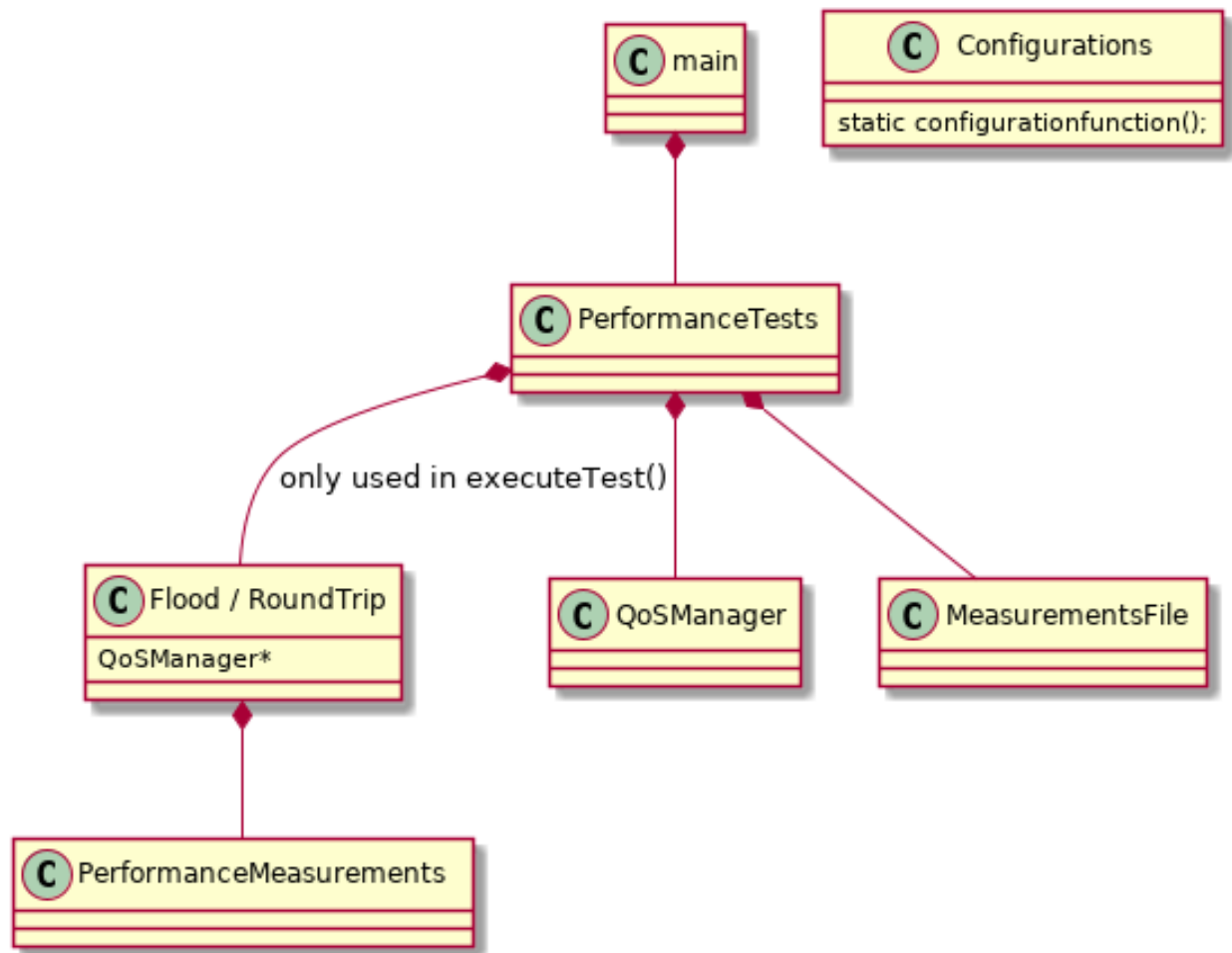
## Class structure

The class structure is shown in the image below. The `PerformanceTests` class executes the wanted tests. In this class, the correct configuration can be set-up for the flood / round trip application.

The `Flood / RoundTrip` class is only locally created in the `executeTest()` function in the `PerformanceTests` class. This is done so the tests are performed in a clean `Flood / RoundTrip` object. The `QoSManager` object is given to the `Flood / RoundTrip` class as a pointer.

The `Configurations` class is a fully static class with static configuration functions inside. These static functions are configured in the `PerformanceTests` class and are executed in the `Flood / RoundTrip` class.

The performance measurements are only executed in the `Flood / RoundTrip` class. These results are requested by the `PerformanceTests` class (the `executeTest()` function) and stored in a file by the `MeasurementsFile` class.



## Configurations

In the configurations class, new configurations regarding QoS policies can be added. They can be implemented with one configuration parameter.

The function must have the following structure in the header file:

```
static void <function name>(dds_qos_t *qos, dds_listener_t *listener, const_
↪int64_t configuration = 0);
```

This is a function structure that is used for every configuration. This makes it easy to select different configurations from the `PerformanceTests` class.

The `<function name>` tag should of course be changed to the wanted function name.

The C++ file must contain the configuration itself. This is an example of the reliability QoS policy in the `Configurations` class:

```
void Configurations::reliability(dds_qos_t *qos, dds_listener_t *listener, const_
↪int64_t configuration) {
    // DDS_RELIABILITY_BEST_EFFORT or DDS_RELIABILITY_RELIABLE
    dds_qoset_reliability(qos, static_cast<dds_reliability_kind_t>(configuration), DDS_
↪SECS(10));
}
```

The configuration is an `int64_t` because almost all parameters can be safely converted to this type. Because of this design choice, the configuration should first be converted to the correct enum type for the reliability.

### Adding performance measurements

The QoS policies are configured using a `QoSRoundtrip` class. This class manages the default and custom QoS policies for each test.

A QoS policy can be configured using the following code:

```
QoSRoundtrip::QoSFunction configDefault;
configDefault(Configurations::reliability, DDS_RELIABILITY_BEST_EFFORT);
```

This is a struct containing the reliability test from the `Configurations` class with the configuration for that function. In this case, the configuration is `DDS_RELIABILITY_BEST_EFFORT`.

You can create as many of these configurations as you want.

The `QoSRoundtrip` class manages the QoS policies. More information can be found in [Explanation of QoSRoundtrip](#). The configuration structures above should be configured with the `QoSRoundtrip` class.

### Default configuration

If you want to change the configuration for a set of tests, the default configuration in the `QoSRoundtrip` class must be changed.

An example is given for a default test below. This executes the `qosPolicyTests()` with the reliability set to reliable and the durability to transient local.

These settings will be active with each test in the `qosPolicyTests()` except if the setting is overwritten by a custom configuration.

```
QoSRoundtrip::QoSFunction configDefault, configDefault2;
configDefault(Configurations::reliability, DDS_RELIABILITY_RELIABLE);
```

(continues on next page)

(continued from previous page)

```
configDefault2(Configurations::durability, DDS_DURABILITY_TRANSIENT_LOCAL);

QoSRoundtrip qosConfig; // You must have a qosConfig object
qosConfig.configureDefaultQoS(configDefault, configDefault2);
qosPolicyTests();
```

The function `qosConfig.configureDefaultQoS()` accepts as many configurations as you want.

The default configuration can also be reset to the default configuration. This can be done by the following command:

```
qosConfig.resetDefaultQoS();
```

## Custom Configurations

If you want a configuration to be executed once. You should alter the custom configuration in the `QoSRoundtrip` class.

This next codeblock shows how custom configurations are made. These custom configurations are the “master” configuration. This overwrites the default configuration if they alter the same QoS policy.

The next codeblock assumes that you already have made a `QoSRoundtrip` object with the name: `qosConfig`.

```
QoSRoundtrip::QoSFunction config;
config(Configurations::deadline, DDS_SECS(10));
qosConfig.configure(config);
executeTest("deadline", &_qosConfig);
```

The codeblock above configures the deadline QoS policy with a configuration parameter of `DDS_SECS(10)`.

The `qosConfig` object is configured normally (without using `configureDefaultQoS()`). The `QoSRoundtrip` object containing the wanted QoS policies is given to the `executeTest()` function with a custom name.

The `executeTest()` function needs a string containing the name of the test and the `QoSRoundtrip` object. This function executes the round trip and stores the result in a measurements file.

## Explanation of QoSRoundtrip

The `QoSRoundtrip` class controls the QoS policies for the round trip performance measurements. This class contains all the necessary QoS objects.

The class contains the `dds_qos_t *` for the writer, reader, topic of the writer and the topic of the reader. Additionally, it contains the `dds_listener_t *` for the listener. Each of these has a custom configuration that is used for the performance tests. As well as a default configuration that will be copied into the custom configuration before the custom QoS policies are set-up.

This class also contains the struct that manages the different configurations. This struct is called `QoSFunction`.

This struct contains a function containing a QoS policy. It must be a function with the format:

```
void <function name>(dds_qos_t *qos, dds_listener_t *listener, const int64_t_
↳ configuration);
```

This struct also contains for what object it is meant. It can be configured that the QoS policy is only configured for the writer, or for the reader. The `applicable()` function can be used to set for whom the QoS policies must be configured.

```
// This function is part of the QoSFunction struct
void applicable(const bool writer, const bool reader, const bool writeTopic, const_
↪bool readTopic);
```

The following functions can be executed for configuring the QoS policies:

```
// Resets the default QoS policies in the class
void resetDefaultQoS();

// Configures the default QoS policies.
void configureDefaultQoS(ConfigFunctions... configFunctions);

// Configures the custom QoS policies
void configure(ConfigFunctions... configFunctions);
```

The `ConfigFunctions` must be of type `QoSFunctions`.

The `configure()` function overwrites the previously called `configure()` function. The `configureDefaultQoS()` function does not overwrite the previously called `configureDefaultQoS()` function. This can only be cleared using the `resetDefaultQoS()` function.

## Measurements

The measurements of each test are appended to a CSV file. This is not a human-readable structure. Therefore, this chapter explains how to make the results readable relatively fast.

Open the measurements file with Excel. Column A should contain the measurement names and column B the measurements. The current test executes 39 QoS tests with 8 different default configurations. Therefore, a table must be created with 39 rows and 8 columns.

Place this command in an empty cell: `=INDEX($B:$B, ROW(B1) + ((COLUMN(B1) - 2) * 39))`.

Then select the square on the right bottom of that cell and select 39 rows and 8 columns with it.

## Matrix board performance measurements

**authors** Joost Baars

**date** June 2020

## Description

The matrix board performance measurements execute performance measurements on the matrix board demonstrator (see: [Matrix Board Communication](#)).

In these measurements, different Quality of Service (QoS) policies are compared with each other. The Matrixboard demonstrator mainly shows the time it takes for a new device to connect or disconnect using DDS.

The selected QoS policies are the durability, liveliness, lifespan, reliability and the deadline QoS policies (see [Quick performance tests](#)).



A short description of how the connect and disconnect durations are measured can be seen below (see chapter [Connect duration & Disconnect duration](#))

An application has been designed for measuring the connect and disconnect duration automatically. This application and the results of the measurements are explained on the pages beneath. The system configuration shows how to configure the dependencies of the performance measurements. The performance application page describes the application in detail: how the application can be executed as well as how the application works from a technical point of view. The performance measurement results show the results of the performance measurements of the selected QoS policies.

## Matrix board performance application system setup

**authors** Joost Baars

**date** June 2020

### Description

The matrix board application often runs on multiple devices for the best results. But those devices must have a synchronized clock to have accurate measurements: the clocks of the devices should be synchronized to <1ms difference. The precision time protocol (PTP) was chosen for the synchronization between the devices. See [Precision time protocol \(PTP\)](#) for a description of the PTP protocol.

If the application is executed only on one device, this page can be skipped. This page is not useful if the application is only executed on one device

### Install

The chosen PTP implementation for Linux based system is the PTP daemon (ptpd).

ptpd can be installed using the following command on Debian based systems (for example Ubuntu and Raspbian):

```
sudo apt-get install ptpd
```

This command downloads the necessary dependencies and installs ptpd. ptpd can also be installed manually by following the following install guide: [PTPd install guide](#).

The official GitHub repository of ptpd can be found here: [PTPd GitHub repository](#).

This PTP implementation should also work for FreeBSD, NetBSD, OpenBSD, OpenSolaris and Mac OS X.

### Setup

ptpd must run in the background for it to work. There are different methods for achieving this. The ptpd application can be started manually every time the performance tests are executed. This application must be started on both devices before the performance application is started! This is the recommended method when the performance measurements are only executed once or twice. This method is explained in [manual execution](#).

When the performance application will be started more often (like when measuring different configurations), the method of automatically starting the application could be more convenient. This method only works on certain devices and takes more work to set-up. This method is explained in [systemctl](#).

## manual execution

The `ptpd` application can be started manually. This can be done by executing the command below.

```
sudo /usr/sbin/ptpd -m -i eth0 &
```

The `ptpd` executable is executed with the `-m` parameter to detect the best active master for time synchronization. The `-i` parameter must be configured with the interface where PTP must be active on. In the example, the first ethernet port was chosen (`eth0`). By checking `ifconfig`, the network names can be found. The preferred network should be chosen. The network name for Wi-Fi is often called `wlan0`.

When it is running, the system configuration is completed and the performance application can be started. See [Matrixboard performance application](#) for starting the performance application.

## systemctl

If you have chosen the manual execution, this section can be skipped.

This method uses `systemctl` and `systemd` as the boot manager. This boot manager is not installed on every device and these steps don't always work for every device. This method is checked with a Raspberry Pi running Raspbian Buster Lite 2020-03-13 and works for this software. On some configurations, this method does not work, then the manual execution is recommended.

With `systemctl`, the initialization of a device can be managed. Services can be configured using `systemctl`.

```
sudo nano /lib/systemd/system/ptpd.service
```

`nano` can be replaced with an editor of choice. This `ptpd.service` should not exist on your system, this command, therefore, creates this file. The following must be inserted to the file:

```
[Unit]
Description=Ptpd accurate time synchronization
Wants=network-online.target
After=network.target

[Service]
ExecStart=/usr/sbin/ptpd -m -V -i eth0
Restart=always
User=root

[Install]
WantedBy=multi-user.target
```

This creates a service for `systemctl` called `ptpd` (name of the file). The application is automatically restarted when it crashes.

The `ptpd` executable is executed with the `-m` parameter to detect the best active master for time synchronization. The `-V` parameter places the application in the foreground (necessary for `systemctl`) and enables debugging. The debugging can be seen when executing:

```
systemctl status ptpd
```

The `-i` parameter must be configured with the interface where PTP must be active on. In the example, the first ethernet port was chosen (`eth0`). By checking `ifconfig`, the network names can be found. The preferred network should be chosen. The network name for Wi-Fi is often called `wlan0`.

The application must first be enabled to be executed on boot. This can be done using the command:

```
sudo systemctl enable ptpd
```

With this command, the `ptpd` service is started on boot.

If you don't restart your device, the service is not started yet. Therefore, the first time you must manually start the service. This can be done using the command:

```
sudo systemctl start ptpd
```

With the status command, the status of the service can be requested. This status command is shown below. The third line should note: `Active: active (running)`.

```
systemctl status ptpd
```

If PTP is successfully running on all devices, then the configuration is done.

More information regarding `systemd` and `systemctl` can be found here: [systemctl/systemd basics](#).

---

**Note:** Time zone

Make sure the time zone is the same for both devices because otherwise the time synchronization won't work correctly.

---

---

**Note:** `ptpd` service

This service relies on the network and may only start after the network is configured. Therefore, it could also not work on systems with `systemd` installed. Then, the application should be started after a different service. The `ptpd` service can also be started manually upon boot using `systemctl restart ptpd`. More information regarding `systemd` and `systemctl` can be found here: [systemctl/systemd basics](#).

---

## Precision time protocol (PTP)

PTP is a method to synchronize devices over a local area network (LAN). PTP should be capable of synchronizing multiple clocks to less than 100 nanoseconds on a network. PTP works best with hardware timestamping but a network card capable of hardware timestamping is necessary for this. The device that is used for the performance measurements in [Matrix board performance measurements](#) is the Raspberry Pi. This microcomputer does not have hardware timestamping and is therefore limited to software timestamping. Synchronization of ~10 microseconds should be doable but 10-100 microseconds is typical for software timestamping with PTP (according to [PTP general information](#)).

This data originates from the following source: [PTP general information](#).

## NTP

NTP was also researched for time synchronization between multiple local devices. NTP should also be able to do <1ms synchronizing on local networks. But the setup was more difficult because an NTP server had to be created. NTP was also unstable on the Raspberry Pi (sometimes it stopped running or stopped finding the correct time). This was probably due to that the Raspberry Pi contains `timesyncd` by default for time synchronization, but `ntpd` was necessary to run an NTP server (`timesyncd` is only for NTP clients). `timesyncd` and `ntpd` cannot run together, and that gave issues configuring it.

### Links

- [Hardware timestamping](#)
- [PTP general information](#)
- [PTPd GitHub repository](#)
- [PTPd install guide](#)
- [More information about Systemctl](#)

### Matrixboard performance application

**authors** Joost Baars

**date** June 2020

### Description

This performance application is made for the matrix board demonstrator (see: *Matrix Board Communication*). The C++ matrix board implementation is used within this application (see: *Matrix Board Communication in C++*).

This page describes how the performance application works. The first sections focus on installing and executing the performance application. By following the steps below, the performance measurements can be repeated. After the section for executing the application, the application itself is explained (see *Performance application*). The last section explains how more tests can be added (see *Add/remove performance tests*).

### General information about the application

The performance application executes an external matrix board application. The matrix board application must, therefore, be compiled first. A different matrix board application is created for each different QoS configuration.

The steps necessary to execute the performance application are described on this page.

The different created configurations for the matrix board communication can be found in:

```
src/demonstrators/MatrixBoard/C++/performance_measurements/  
custom_matrixboards/.
```

The performance application can be found in:

```
src/demonstrators/MatrixBoard/C++/performance_measurements/  
performance_application/.
```

### Dependencies

The necessary dependencies for succesful compilation of the matrix board application in C++ are the following:

- CMake
- C++17 compiler (Gcc 7+ for example)
- Cyclone DDS library

Installing Cyclone DDS and the other dependencies can be found in: *Clean Install with internet*.

## Compiling the matrix board applications

Start by going to the custom matrixboards directory:

```
src/demonstrators/MatrixBoard/C++/performance_measurements/  
custom_matrixboards/.
```

This directory contains the different QoS implementations for the matrix board application. These configurations should all be built separately into an application.

Create a build folder for building the custom matrix boards (do this within the `custom_matrixboards` directory) and enter it. This can be achieved using the following command:

```
mkdir build && cd build
```

Each of the custom configurations can then be built using the command:

```
sh ../build_all.sh
```

This is a shell script that builds all the different configurations.

---

### Note: Manual compilation

The shell script builds all the different configurations. Building these configurations can also be done manually by building every matrix board application one by one. Manual compilation is not advised as it takes more time and is more prone to errors.

Manually, each configuration can be built with the commands:

```
cmake -D CUSTOM_MBC=<custom configuration name> .. && make -j4
```

In this command, `<custom configuration name>` should be replaced by the desired configuration. For the default performance application, each configuration in the `custom_matrixboards` directory should be compiled.

The `CUSTOM_MBC` CMake parameter defines what application is being compiled.

---

## Compiling the performance application

Go to the performance application directory:

```
src/demonstrators/MatrixBoard/C++/performance_measurements/  
performance_application/.
```

Then create a build folder for building the performance application and enter it:

```
mkdir build && cd build
```

The performance application can then either be built for the slave or the master. There should only be one master application active at the same time. There may be multiple slaves active.

The master application can be built with the following command:

```
cmake -D PERFORMANCE_ACTOR=Master .. && make -j4
```

The slave application can be built with the following command:

```
cmake -D PERFORMANCE_ACTOR=Slave .. && make -j4
```

### Executing the performance application

The master performance application can be executed with various parameters. These parameters will be explained in this section. The slave application will also be described.

Both these applications give information about their parameters when executed with no parameters. So when you forget the parameters, execute the application without parameters.

The directory where the commands from this section are executed from is:

```
src/demonstrators/MatrixBoard/C++/performance_measurements/  
performance_application/  
unless specified otherwise.
```

The page [Example performance application](#) contains an example for running this application.

---

#### Note: Synchronization

For the application to work correctly on different devices, the time should be synchronized. Synchronizing the time is explained on the page: [Matrix board performance application system setup](#).

---

### Slave

The slave application can be executed using the following command:

```
./Slave <MatrixBoard ID>
```

- MatrixBoard ID: The ID/position of the matrix board

The slave application must be executed before the master application. Multiple slave applications can be executed at once. The connect and disconnect durations are measured for each slave. The lowest, average, and highest values are stored in the logging file.

### Master

Only one parameter for the master application is necessary. That's the matrix board ID. The application can be executed using the following command:

```
./Master <MatrixBoard ID> <Name of logging file> <Execution amount> <Application_  
↪Location> <Application name>
```

- **MatrixBoard ID:** The ID/position of the matrix board
- **Name of logging file:** OPTIONAL: the name of the logging file, by default: `logging.csv`
- **Execution amount:** OPTIONAL: The number of executions each configuration has, by default: 4
- **Application location:** OPTIONAL: the location containing the matrix board applications, by default: `../../custom_matrixboards/build/`
- **Application name:** OPTIONAL: By default, all custom matrix board applications are executed. With this parameter, you can only measure one custom application. For example `default` for the default configuration.

---

**Note:** Execution of the program

The slave applications must be started first!

The <Matrixboard ID> parameter should be unique for every matrix board. Additionally, it may not be less than 1.

The order at which matrix boards and their associated IDs are started does not matter.

If the logging file exists already, the new data is appended to the existing data.

The application location should include all executables for the matrix boards that are used by the `TestsImplementation` class. This is by default all of the applications in the `custom_matrixboards` directory as well as the default configuration. Each of these can be build using the shell script `build_all.sh` in the `custom_matrixboards` directory (see [Compiling the matrix board applications](#)).

---

## Example performance application

An example of starting the performance application is given in this section.

Execute the slave first. Start it with the command below.

```
./Slave 1
```

Now, a master can be started in a new terminal to see the performance application going.

```
./Master 4
```

The default configuration executes each test only four times. When both the master and slave stop running, the performance test is done. The results can be seen by executing:

```
cat logging.csv
```

This shows all the results. See [CSV output](#) for an explanation of how the logging can be interpreted.

## CSV output

The output of the performance application is a CSV (comma-separated value) file. When a custom matrix board application is executed, the name is stored in the CSV file.

A separate page is written for analyzing this data and automatically calculating the average values for each configuration (if the execution amount is >2). See: [Performance measurements data analysis](#).

The paremeters are shown in the following three tables. The first 3 parameters of the measurements:

	1st		2nd		3rd
Lowest connect duration		Average connect duration		Highest connect duration	

These parameters are for the connect duration. The connect duration of all devices that connect to the master's matrix board topic are measured. The lowest, average and highest values are stored (1st, 2nd and 3rd parameter).

The second 3 parameters of the measurements:

	4th		5th		6th
Lowest disconnect duration		Average disconnect duration		Highest disconnect duration	

These parameters are for the disconnect duration. The disconnect duration of all devices that disconnect from the master's matrix board topic are measured. The lowest, average, and highest values are stored (1st, 2nd, and 3rd parameter).

The last 3 parameters of the measurements:

	7th		8th		9th
CPU usage [%]		Physical memory usage [kB]		Virtual memory usage [kB]	

These parameters are for the resource usage. The CPU usage, physical memory usage, and physical memory usage are measured. These measurements are all executed on the master's matrix board application.

---

**Note:** All these parameters are written on the same line!

---

## Matrix board application

The matrix board application is the same as is described in *Matrix Board Communication in C++*. This application has, by default, debugging built-in. The debugging sends a timestamp to the `Debug` topic. At the following moments, a timestamp is sent:

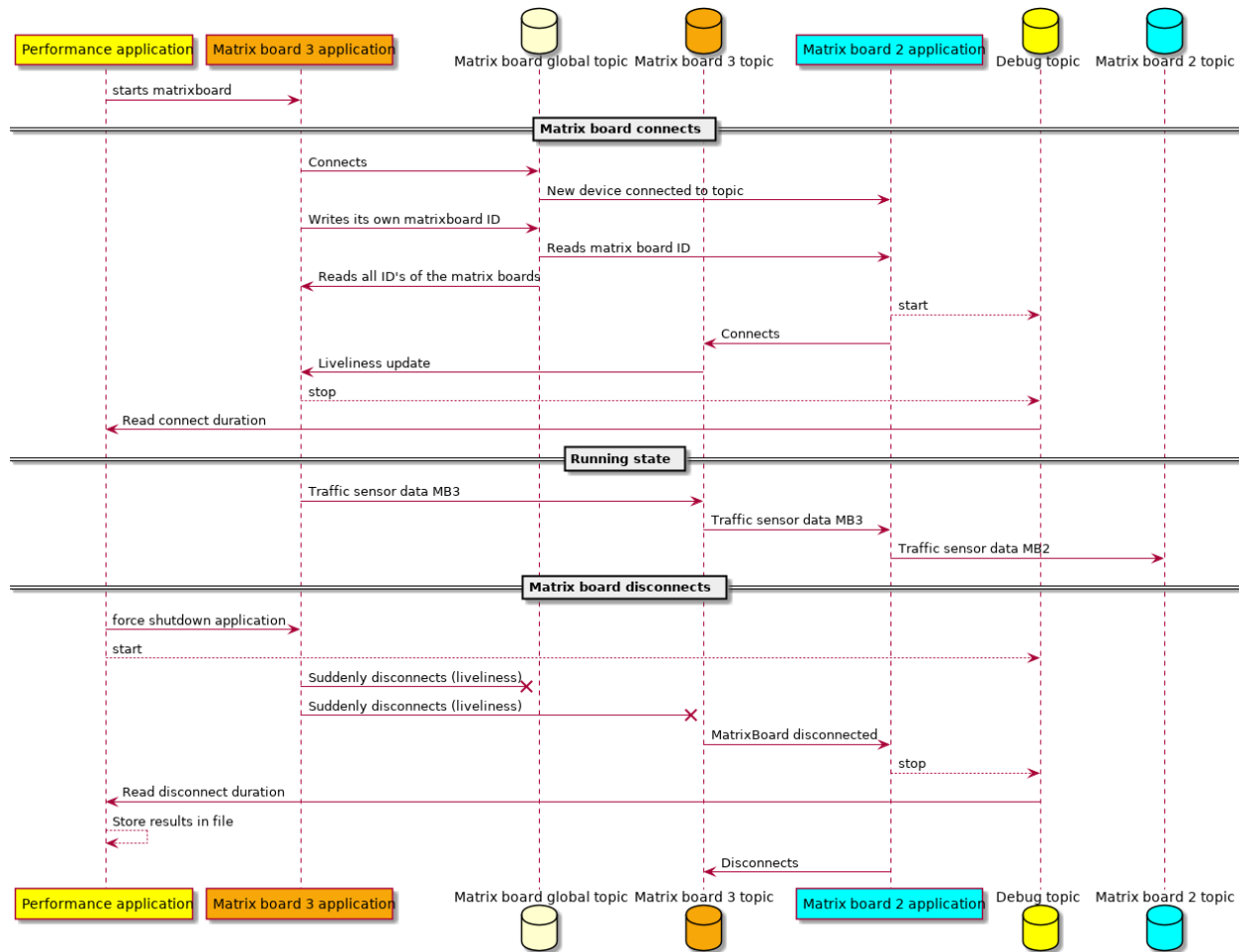
- When the matrix board connects to a matrix board topic
- When a matrix board connects to its own topic
- When a matrix board writer or reader disconnects

This debugging can be used for the performance application to calculate the connect and disconnect duration.

## Sequence diagram

The sequence diagram below shows how communication with DDS is used within the performance application. In this sequence diagram, `Matrix board 2` is already active and `Matrix board 3` is started at the beginning of the sequence diagram.





This sequence diagram is split into three parts. The **Matrix board connects** part shows the communication flow when a new matrix board connects to the network. First, the matrix board ID is sent to the Matrix board global topic. When the ID is received by Matrix board 2 application, it can connect to the Matrix board 3 topic. Before connecting, the starting timestamp is sent towards the Debug topic. When Matrix board 3 application receives a liveliness update that a new reader has connected, the stop timestamp is sent towards the Debug topic.

The connect duration is measured by subtracting the start timestamp from the stop timestamp. This is measured on Matrix board 3 topic instead of the Matrix board global topic because the global topic has certain QoS policies that cannot be altered (else the global topic won't work as expected). The Matrix board 3 topic does not have QoS policies that must be configured. Therefore, Matrix board 3 topic is used for the measurements in the sequence diagram.

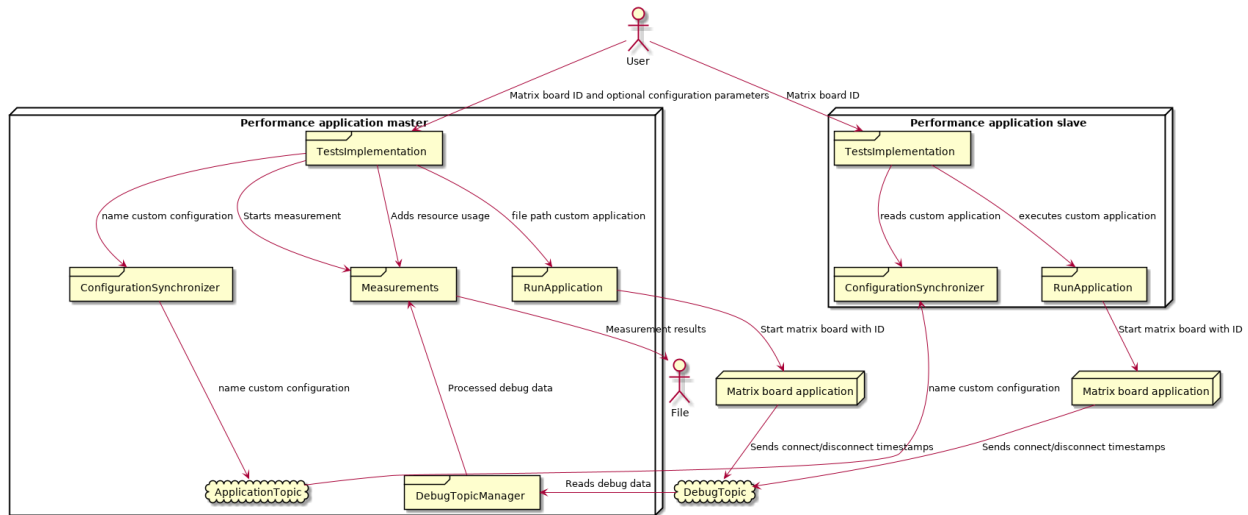
The **Running state** is the stable state. In this state, the matrix board sends the sensor data towards its own topic. This can be read by other matrix boards (in this case Matrix board 2).

The **Matrix board disconnects** state shows the communication flow when a matrix board disconnects. Two topics are notified by this action. Matrix board 2 is notified by both topics, and responds by disconnecting from the Matrix board 3 topic. At the moment Matrix board 3 application is shut down, the start timestamp is sent towards the Debug topic. When Matrix board 2 application is notified by the Matrix board 2 topic, it sends the stop timestamp towards the Debug topic.

## Performance application

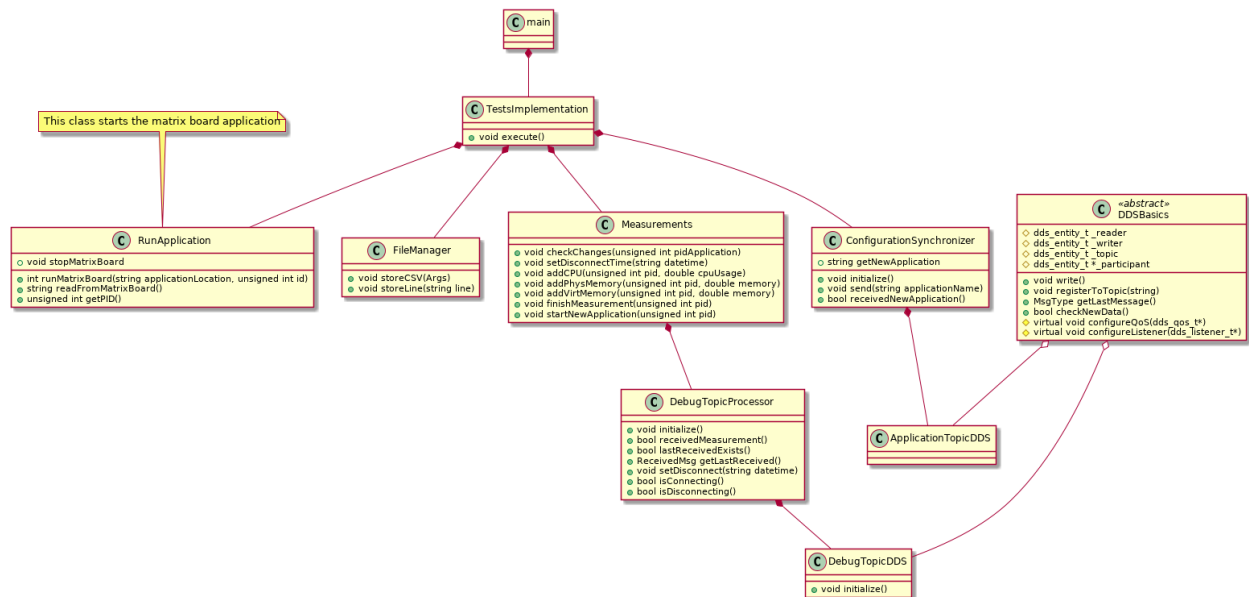
This section describes how the performance application works using the class diagram. This section is divided into the Master and Slave application.

Below, the deployment diagram is shown for the performance application. In this diagram, the communication between the master and slave application can be seen. This diagram shows that each application creates one matrix board application. The matrix board application sends data to the Debug topic. The Debug topic is being read by the master application. There is also an Application topic which is used to send the configuration to the slave applications. The slave application can then start the same application as the master.



## Master

The class diagram of the Master application is displayed below.



The main is the start of the application. Here, the input parameters are processed and the `TestImplementation` is executed.

The `TestsImplementation` class is where the tests/performance measurements are implemented. This class is the only class that needs to be changed for new performance measurements. This class sends the name of the application that is executed to the slave with the `ConfigurationSynchronizer` class (using DDS). This way, the slave and master can both start the same application (with the same configuration). By doing this, the started applications are compatible with each other.

After sending the name of the application to the slave, it executes that application. This application is executed multiple times until the `Execution` amount is reached of *Executing the performance application*. This execution amount stands for the number of times a certain configuration is executed. So with an execution amount of 50, each configuration is executed 50 times. This results in 50 measurements for one configuration (the configuration mentioned here are the configurations that can be found in the `custom_matrixboards` directory).

When the application is started, a loop starts. This loop measures the CPU, virtual memory, and physical memory usage of the application. The loop takes a random time from 2 to 7 seconds. This time was chosen because a static time could influence the results (have the same result, if it is a deterministic application). The debug topic is also being checked for any new messages. When the loop ends, the matrix board application is terminated with a kill signal and the start of the disconnect duration is stored.

The `Measurements` class manages the measurements. This class reads from the debug topic using the `DebugTopicProcessor` class and stores the results in a CSV (comma-separated value) file using the `FileManager` class. The `Measurements` class combines the measurements from the `DebugTopicProcessor` class to get the connect and disconnect duration from the timestamps it receives (by subtracting the start timestamp from the stop timestamp). The `DebugTopicProcessor` manages the individual messages from the debug topic and stores them based on their classification (see below for an explanation of how they are classified).

The matrix board application sends messages to the debug topic. These messages contain 3 different parameters, a string for a timestamp, a classifier, and an ID. The timestamp gives the ability to calculate the connect/disconnect duration. The classifier can be four different values. It can be connecting (start during a connect), connected (stop during a connect), disconnecting (start during a disconnect), and disconnected (stop during a disconnect)(see *Sequence diagram*). The ID of each message on the debug topic is the ID of the topic where the measurements are performed on. In the case of the sequence diagram above, it is 3 (see *Sequence diagram*, it's 3 because the measurements are executed on the `Matrix board 3` topic).

---

**Note:** Master

The master application should have at least one slave connecting to its topic for the measurements. Therefore, a slave with an ID less than the master's ID should be active.

---

---

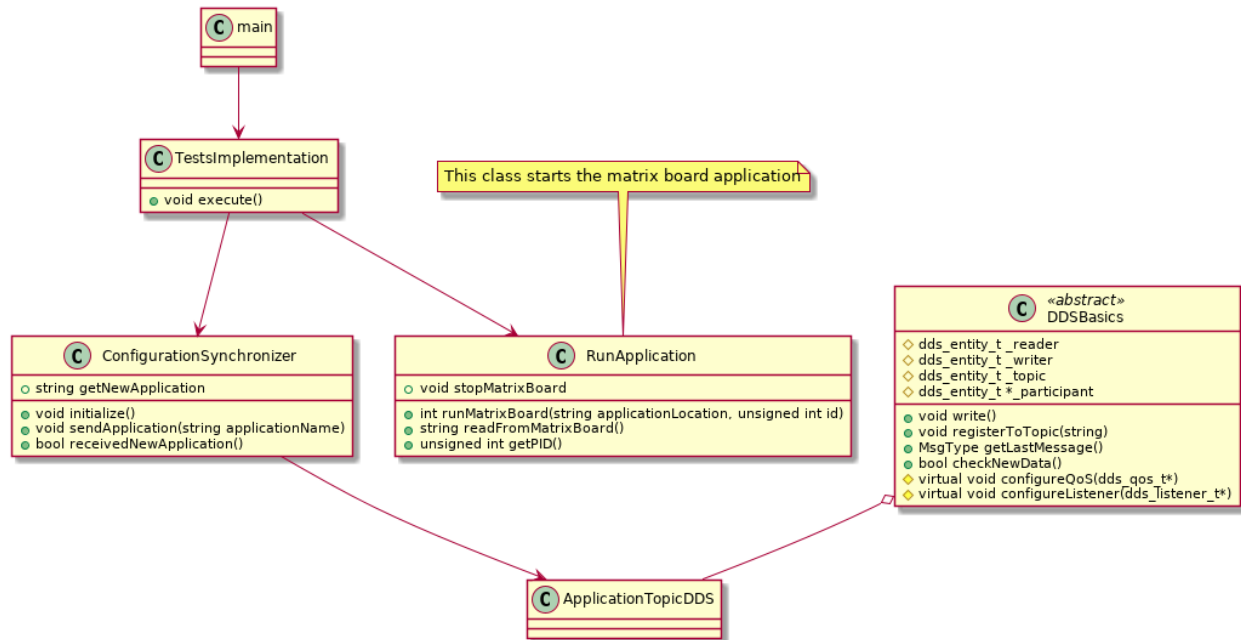
**Note:** Multiple masters

There can be multiple masters, but it must be verified that they execute the same matrix board configuration (and don't execute different configurations). Each master only reads the data on the topic of his matrix board. Therefore, this would give different results for each master.

---

## Slave

The class diagram of the slave application can be seen below.



The basics of this application work the same as the Master application. The main also parses the input parameters of the application. The TestsImplementation reads from the application topic with the ConfigurationSynchronizer class. If the Master sends a message containing an application, this application is executed. If a matrix board application was already running on the slave, this application would be forcefully stopped (these connect and disconnect durations are not measured).

## Participant

There are 2 different participants in the performance application. One for the application topic for sending/receiving the application that must be executed. The other one is for the debug topic. This one is created in the Measurements class and deleted when the object is deleted. This is necessary because writers and readers on a topic are only removed when the participant is deleted and not when the object that creates the writer/reader is deleted. When a Measurements object is deleted, the corresponding reader/writer also needs to be removed. This is achieved by having a separate participant for the Measurements class, which is removed when a Measurements object is deleted.

In the future, the DebugTopicManager class could become a static class, or singular readers/writers could be deleted from a topic (if this is possible within Cyclone DDS).

## Resource usage

This chapter describes how the resource usage (CPU, physical memory and virtual memory) is measured.

The CPU, physical memory, and virtual memory are all measured using the Linux system calls. Linux keeps track of each application what resources it uses. These values are read within the performance application using the PID (Process ID) of the matrix board application. Therefore, these measurements do not influence the resource usage of the matrix board application.

The CPU usage is only read every second because of a limitation for measuring the CPU usage. The CPU usage must be calculated with 2 values after a certain time. Therefore, the CPU usage could have a bigger deviation between the measurements (because of fewer measure points).

The results of the resource usage within the software are tested by comparing it with the `top` command of Linux. These values corresponded with the results within the software.

### Add/remove performance tests

Within the master application, new tests can be created easily. The slave does not have to be changed since it receives the matrix board application it needs to execute from the master.

The following directory contains the `TestsImplementation` class of the master:

```
src/demonstrators/MatrixBoard/C++/performance_measurements/
performance_application/tests_implementation_master/
```

In the function `void TestsImplementation::execute(const unsigned int matrixboardID)`, new matrix board applications can be added.

A new application can be added using the following C++ code:

```
sendAndExecuteApplication(matrixboardID, "durability_persistent");
```

This line of code starts an application with the name of the second parameter. In this case, the name is `durability_persistent`. This should be the name of the matrix board application that is stored in the application location. This location points by default to `src/demonstrators/MatrixBoard/C++/performance_measurements/custom_matrixboards/build`. This application is executed once or multiple times based on the `Execution amount` parameter (by default 4 times).

Custom matrix board applications can be added in the `src/demonstrators/MatrixBoard/C++/performance_measurements/custom_matrixboards/` directory. A directory containing a `specific_matrixboard.cpp` file should be created. The contents of this file are shown in the code-block below. The wanted QoS policies can be added at the place where the `// Add custom configurations` in this function comment is.

```
#include "matrixboard/dds/specific_matrixboard.hpp"

namespace matrixboard {

/**
 * @brief configures the qos policies
 *
 * @param qos the qos object
 */
void SpecificMatrixBoard::configureQoS(dds_qos_t *qos) {
    // Add custom configurations in this function
    dds_qosset_liveliness(qos, DDS_LIVELINESS_MANUAL_BY_PARTICIPANT, DDS_MSECS(2500));
}

} // namespace matrixboard
```

The liveliness should always be configured and should always be less than 4 seconds. This is because the `ActiveMatrixboards` topic should be slower compared to the `specific_matrixboard` topic with the liveliness. Otherwise, the connect or disconnect duration could be skipped to be sent towards the debug topic. The liveliness can be adjusted though in the code-block above.

For ease of compiling all the custom matrix boards, the new application should be added to the `build_all.sh` script. So all custom matrix boards can be compiled using one build script. The new application can be added by

appending the following to the `build_all.sh` file:

```
cmake -D CUSTOM_MBC=unreliable .. && make -j4
```

In this example, `unreliable` must be replaced by the name of the newly added application.

### Matrix board performance measurements

**authors** Joost Baars

**date** June 2020

#### Description

The matrix board performance measurements mainly measure the connect and disconnect duration of various QoS configurations. By comparing different QoS policies with the connect and disconnect duration, the overhead of the QoS policy can be visualized. The Quality of Service (QoS) policies are a set of configurable parameters that control the behavior of a DDS system (how and when data is distributed between applications). Some of the parameters can alter the resource consumption, fault tolerance, or the reliability of the communication.

Each entity (reader, writer, publisher, subscriber, topic, and participant) of DDS has associated QoS policies to it. Some policies are only for one entity, others can be used on multiple entities.

#### Measurement 1

The performance measurements application was executed with 50 measurements for each configuration. 15 different custom configurations were executed in these tests.

A description of each of the QoS policies and how they are used in the configurations can be found in: [Measured QoS policies](#).

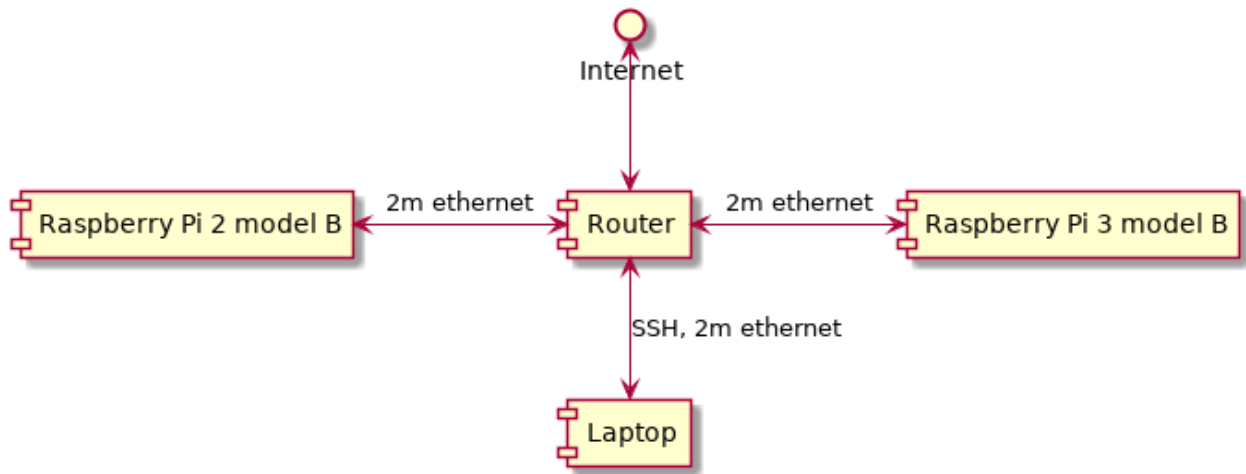
The result of the measurement can be found in: [Results measurement 1](#).

#### Setup

The setup consists of two Raspberry Pi's with each a clean install of Raspbian Buster Lite 2020-03-13. The Raspberry Pi's that are used are the Raspberry Pi 2 model B and a Raspberry Pi 3 model B.

These Raspberry Pi's only have the necessary dependencies installed. Both Raspberry Pi's ran PTP in the background for time synchronization.

The setup can be found in the image below. All devices are connected using ethernet cables (CAT6 cables were used).



Each device is connected using ethernet for having less deviation in the communication. Additionally, this results in better accuracy for PTP (time synchronization).

### Measured QoS policies

Every QoS policy in this list has the same configuration as the `default` configuration. With only one parameter changed. The QoS policy is changed for the writer and the reader. It is chosen to have the same QoS policies active on the writer and the reader to avoid any QoS incompatibilities.

More information about the durability types can be found here: [Durability](#).

By default, CycloneDDS has some QoS policies enabled/configured. These are described in *Cyclone DDS default QoS policies*.

### Cyclone DDS default QoS policies

The default QoS policies that are already enabled by Cyclone DDS are the following:

The default QoS policies enabled for the **writer**:

Durability service is configured with a history of 1. The resource limits are unlimited within this QoS policy.

The cleanup delay (which will be varied in the *Durability transient service*) is by default 0.

The reliability reliable is configured with a maximum blocking time of 100 milliseconds.

The ownership strength is configured with a value of 0 (not changed during the tests).

The transport priority is configured with a value of 0 (not changed during the tests).

The lifespan is configured to be infinite.

The writer data lifecycle is configured with auto dispose unregistered instances of 1 (not changed during the tests).

The default QoS policies enabled for the **reader**:

The reliability best effort is configured.

The time-based filter is set to a minimum separation of 0 (not changed during the tests).

The reader data lifecycle is configured to be infinite (not changed during the tests).

The lifespan is configured to be infinite.

The subscription keys care configured to be 0 (not changed during the tests).

These QoS policies are found from the source code of Cyclone DDS. The file can be found in [Default QoS policies Cyclone DDS](#). The functions `ddsi_xqos_init_default_reader` and `ddsi_xqos_init_default_writer` contain the default QoS policies for the reader and the writer.

### Default

Liveliness configured as manual by the participant (the participant must send a message manually within a certain timeframe). The liveliness time is configured as 2500 milliseconds. The reliability is reliable with a maximum blocking time of 10 seconds.

### Durability

The durability configures if messages must be stored. And if so, where they are stored.

The durability set in the QoS policy is the name of the measurement. With `durability_transient`, durability transient is enabled in the durability QoS policy.

More information about this QoS policy can be found here: [Durability](#).

When there is service in the name, there is an additional configuration.

### Durability transient service

More settings can be configured for the durability with the `durability service` QoS policy.

In the two tests with the `durability service`, only the cleanup delay was changed (how long information must be kept regarding an instance). The history policy was set to `KEEP ALL`. The settings for the resource limits within this service were all set to 25 messages/instances and samples per instance,

The `durability_transient_service_long` is configured with a time of 4 seconds. The `durability_transient_service_short` is configured with a time of 10 milliseconds.

More information about this QoS policy can be found here: [Durability service](#)

### Deadline

The deadline can be configured to set a requirement for the message frequency of the writer. If a writer does not send a new message within the deadline time, the reader is notified. When there is a backup writer, the reader automatically connects to the backup writer.

The `long_deadline` is configured with 4.5 seconds. The `short_deadline` is configured with 1.1 seconds. The `short_deadline` is above 1 second because that is the interval in which the matrix board sends its sensor data. If it would be below 1 second, the matrix board would have exceeded the deadline.



## Lifespan

The lifespan can be configured to set how long messages can still be sent to new readers. So a newly connected reader can read already sent messages with the lifespan configured.

The `long_lifespan` is configured with 4 seconds. The `short_lifespan` is configured with 50 milliseconds.

## Liveliness

The liveliness can be configured to be able to know if writers/readers are still alive on a topic. The liveliness can be automatically managed by DDS (by pinging the topic once in a while). It can also be configured manually if you already sent messages within the maximum liveliness.

The “short\_liveliness” is configured to automatically send messages (`liveliness_automatic`). The time is set to 50 milliseconds. The `long_liveliness` is configured to automatically send messages. The time is set to 4000 milliseconds. Four seconds has been chosen because this is under the liveliness of the Active Matrixboards topic of five seconds. The `manual_liveliness` is configured to base the liveliness on the topic. Messages must be sent manually (`liveliness_manual_by_topic`). The default configuration is configured to base the liveliness on the participant. Messages must also be sent manually in this configuration (`liveliness_manual_by_participant`).

## Reliability

The reliability can be configured to make sure that messages are received correctly by the reader. Reliability `reliable` makes sure that messages are received correctly. If something goes wrong in the communication, the message is resent.

The `reliable` application uses reliability `reliable` for the QoS policy. Therefore, messages are checked if they are correctly received. This setting is the same as the default QoS policy.

The `unreliable` application uses reliability `best effort` for the QoS policy. Therefore, messages are not checked if they are correctly received.

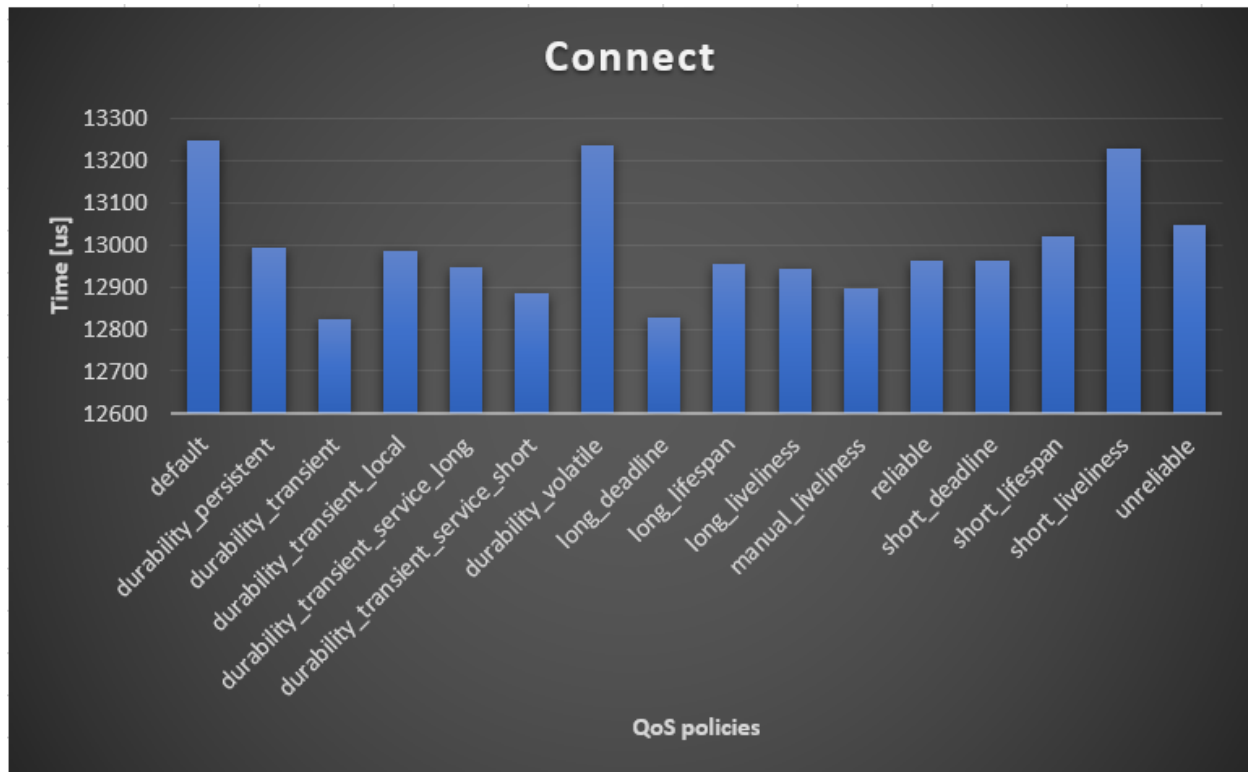
## Results measurement 1

For each custom configuration, the performance measurements were executed 50 times. The average of these 50 measurements can be seen in the graphs in this section.

For each result described below, the deviation is also shown. This deviation is the difference between the highest value and the lowest value of a custom configuration. Only one deviation value is shown because this did not have interesting results (more can be shown if explicitly mentioned). This deviation is the highest deviation (of the 50 measurements) on the default QoS policy.

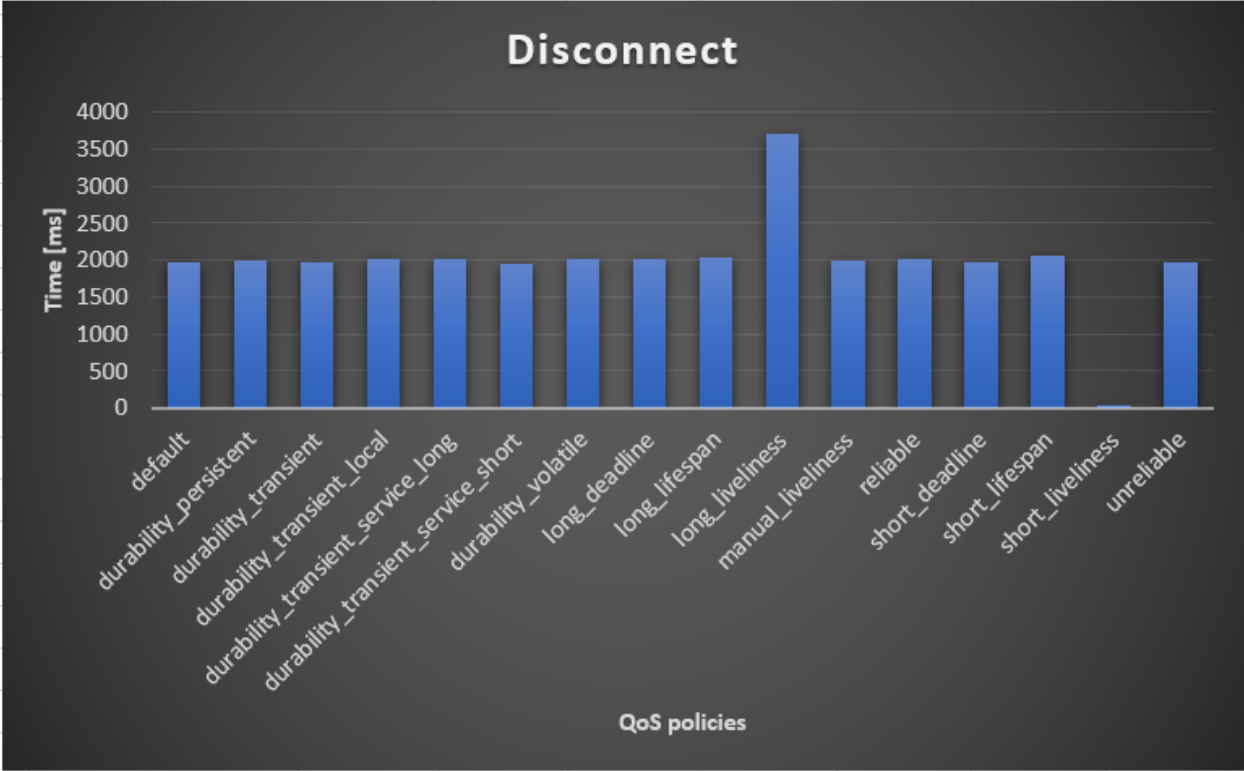
The connect durations in the graph below are in microseconds. The default configuration is exactly the same as the `reliable` configuration (see [Reliability](#)). Therefore, the longer default configuration is probably caused by being the first measurement. The average of all the QoS policies lays around 13 milliseconds connect duration. The `durability_volatile` and the `short_liveliness` QoS policy are a bit higher compared to the other QoS policies. Based on the `durability_volatile` usage, the QoS policy should not influence the connect duration. Therefore, the peak of the `durability_volatile` connect duration is probably due to connect duration fluctuations. As can be seen below, the maximum measured deviation is high (33% deviation compared to the average result). This shows that there is quite a lot of fluctuation in the connect duration.

**Maximum deviation:** 4321 microseconds



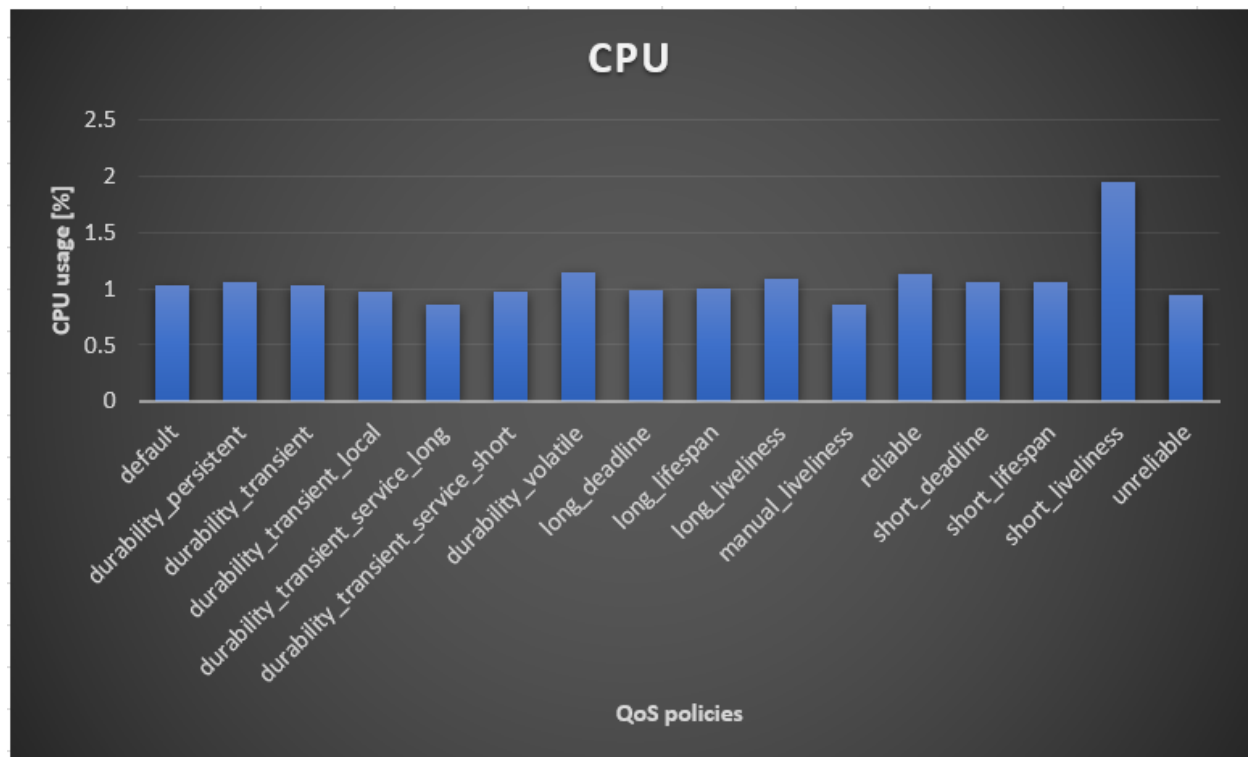
The disconnect duration seems mostly the same for the QoS policies. There are 2 QoS policies that are noticeably different in the graph below. The `long_liveliness` QoS policy takes a lot longer compared to the others. This is because a device is confirmed as “disconnected” after a longer time. The `short_liveliness` QoS policy takes a lot shorter for the same reason. The `short_liveliness` QoS policy detects a disconnect after it receives no messages from the device anymore within 50 milliseconds. Therefore, the result is below 50 milliseconds. The lowest result of the 50 measurements with the `short_liveliness` was only 9 milliseconds. The longest was 49 milliseconds. This shows that the QoS policy does its job very effectively (no measurements equal or above 50 milliseconds). The other QoS policies don’t seem to matter a lot for the disconnect duration.

**Maximum deviation:** 960



The CPU usage can be seen in the graph below. Most of the average CPU values lay around the 1%. There is one QoS policy that clearly requires more CPU usage. This is the short\_liveliness QoS policy. This is expected for the short\_liveliness QoS policy because it must ping to the other devices a lot more often. The application must ping once within the 50 milliseconds compared to the long\_liveliness with a ping within 4 seconds.

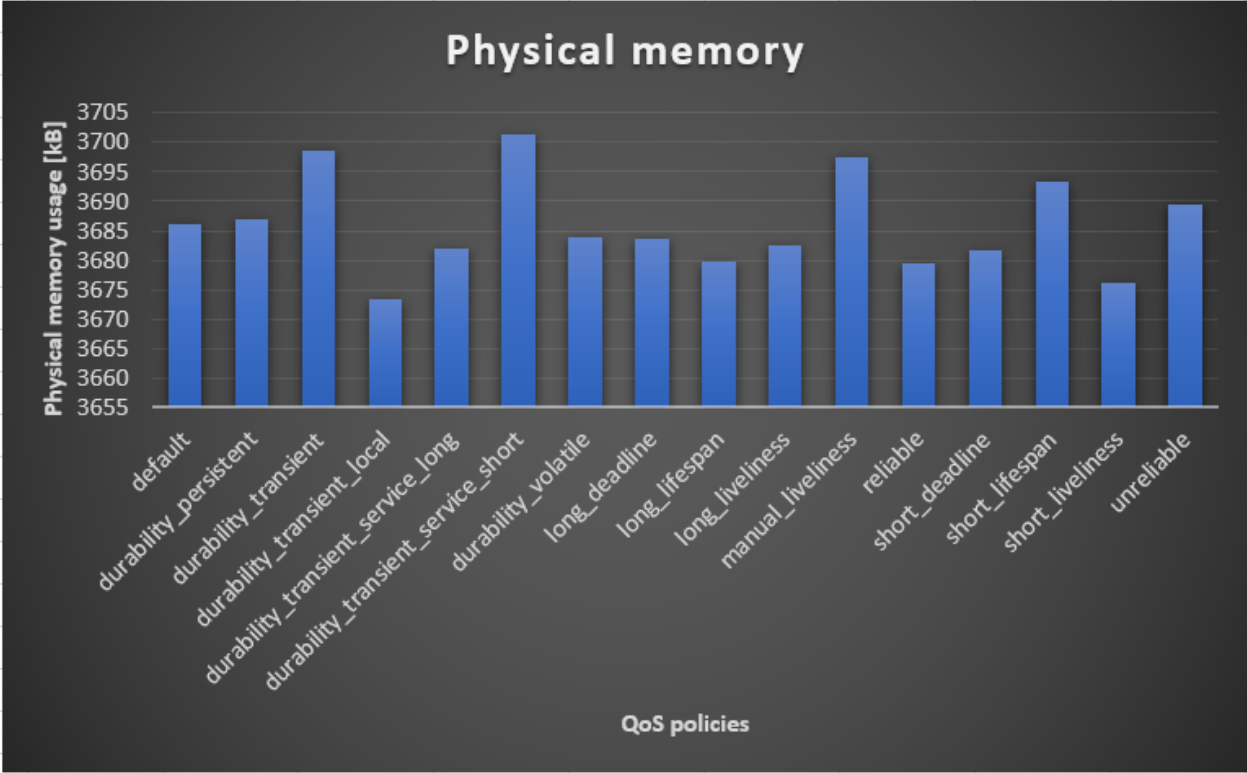
**Maximum deviation:** 2.48%



There are no big differences in the physical memory usage. These measurements were also rather stable with a maximum deviation of fewer than 150 kilobytes between the QoS policies (4% difference). The durability transient and durability transient service long QoS policy seems to use more memory. The manual liveliness and short lifespan also seem to use a little bit more memory.

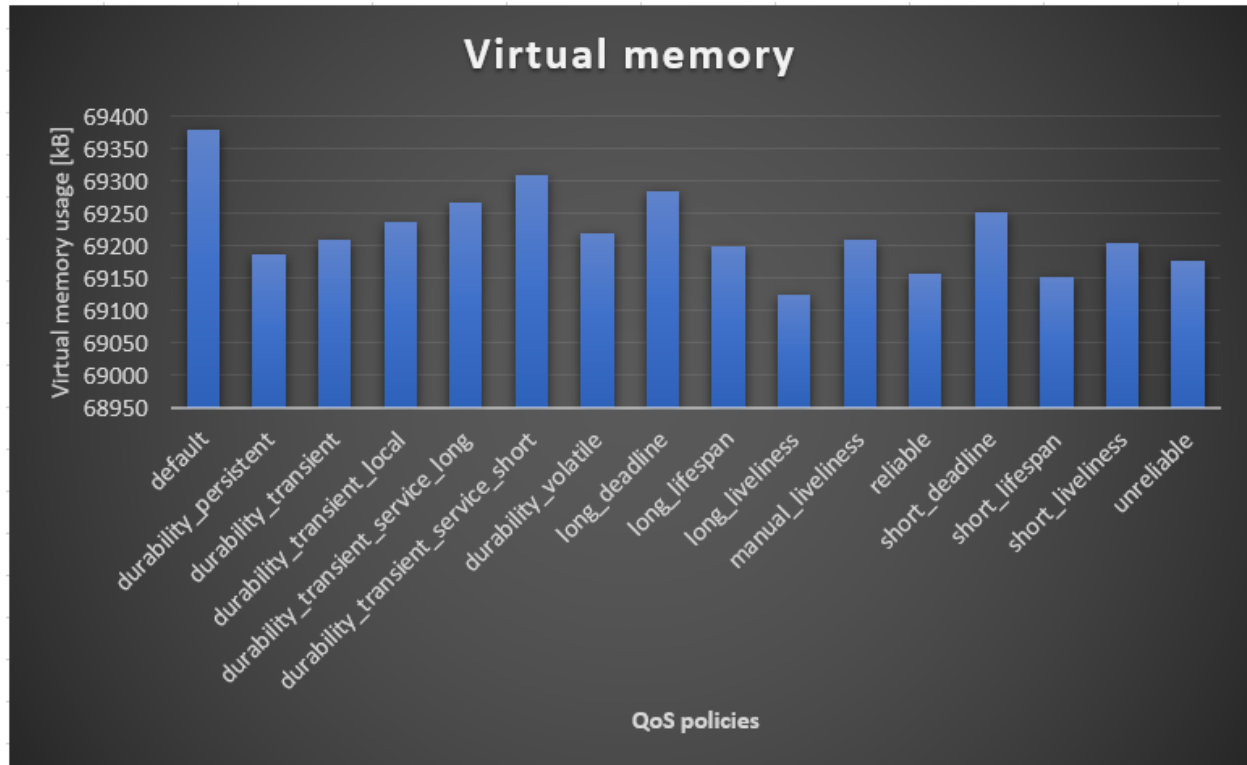
As expected, the durability transient QoS policy uses more memory (old messages are stored using the memory). The durability transient service short has around the same results compared to the durability transient QoS policy. The difference between the long and short variant for the durability transient service is that the short variant keeps the information regarding an instance for 4 seconds instead of 10 seconds. Possibly, this accounts for the difference in the graph.

**Maximum deviation:** 256.75 kB



The virtual memory usage was very stable and only differs 400kb maximum between the QoS policies (<1% difference). The differences could be accountable by the measure error. The default QoS policy is the same as the reliable QoS policy. Therefore, the deviation between the measurements seem to be because of measure inaccuracy instead of QoS policies.

**Maximum deviation:** 1442.1 kB



### Connect duration conclusions

The connect duration was mostly close to the 13 milliseconds. This didn't deviate much between the QoS policies. There was quite a lot of deviation between the connect duration measurements. This could be because short times (milliseconds) are measured and the Raspberry Pi's run a Linux OS that is not real-time. This OS could negatively influence the deviation between the measurements.

### Disconnect duration conclusions

The disconnect duration is mostly managed by the liveliness QoS policy. The matrix board application writes a message to its topic every second. Therefore, the liveliness is updated every second (also with the liveliness set to automatic).

The maximum deviation between the disconnect measurements is always less than the configured liveliness. This is also expected, because when the liveliness time is reached, the disconnect duration measurement stops.

When the liveliness was set to 2.5 seconds and the writer writes a message every second, the maximum measured deviation was a little bit less than a second. Also, the results were always between 1.5 seconds and 2.5 seconds. This makes sense because when the liveliness timer reaches 1.5 seconds, a new message is sent by the writer, and the liveliness is reset to 2.5 seconds.

When the automatic liveliness was set to 50 milliseconds, the maximum deviation was measured to be 40 milliseconds! The shortest measured disconnect duration was only 9 milliseconds. That means that the liveliness ping messages are in general executed every 40 milliseconds (with 50 milliseconds liveliness configured).

## Resource usage

The `liveliness automatic` QoS policy has a noticeable impact on the CPU usage. Especially when the `liveliness` is configured to be short. The `durability transient` and `durability transient service long` seem to have a small impact on the physical memory usage.

## Measurement 2

A second measurement was executed to see the scalability difference. The performance measurements `default` QoS was executed twice. Once with 2 devices on the network (master + slave) and the other time with 4 devices on the network (master + 3 slaves).

The measurements for the scalability were not reliable because it was not possible to have the same type of connection for more than 4 devices. The used router only had 4 ethernet ports. More devices could be connected using Wi-Fi, but Wi-Fi is a lot slower and could affect the measurements negatively. Additionally, there were only 4 Raspberry Pi's. Therefore, if more devices were added, laptops with different OSes had to be used (which could also negatively affect the test).

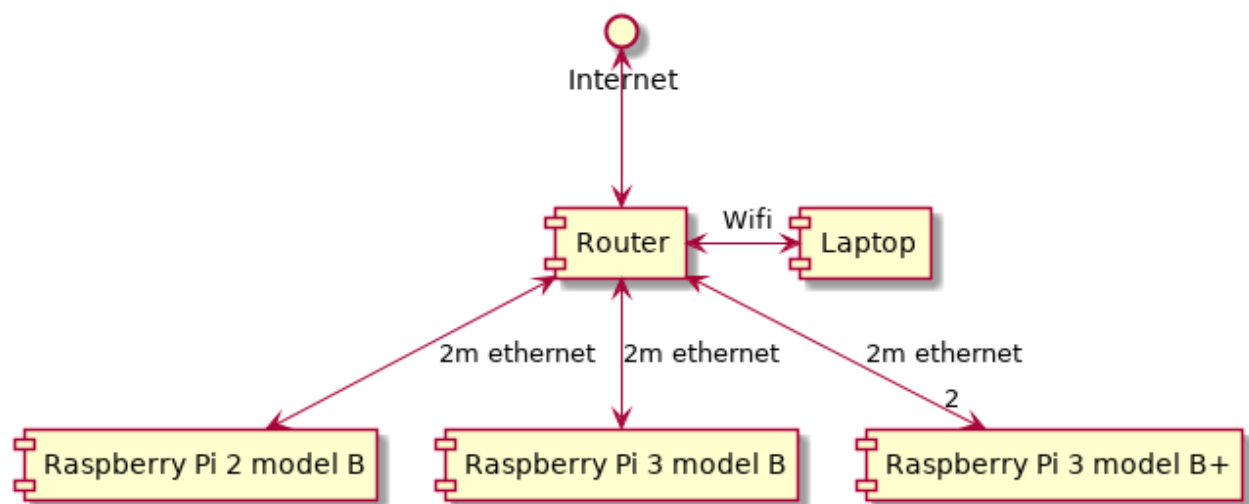
## Setup

The setup consists of 4 Raspberry Pi's with each a clean install of Raspbian Buster Lite 2020-03-13. The Raspberry Pi's that are used are:

- Raspberry Pi 2 model B
- Raspberry Pi 3 model B
- 2 x Raspberry Pi 3 model B+

These Raspberry Pi's only have the necessary dependencies installed. All 4 Raspberry Pi's ran PTP in the background for time synchronization.

The setup can be found in the image below. All devices are connected using ethernet cables.



The only devices connected to this router are the ones shown on the image above.

During the run with only 2 Raspberry Pi's, both Raspberry Pi 3 model B+'s were used.

The master had an ID of 15 in the tests.

2 devices test: The only slave had an ID of 11. 4 devices test: The slaves had the ID's: 11, 12, 16.

### Results

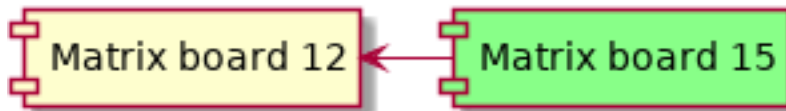
For each setup (2 devices and 4 devices) 50 measurements were executed. These measurements were executed on the default configuration.

The results of these measurements can be seen in the graphs below. This shows that the connect and disconnect times were slightly faster with 4 devices. The difference for the connect time is around 5.6%. More measurements should be executed to see if this difference is accountable by the deviation between the measurements or the number of devices.

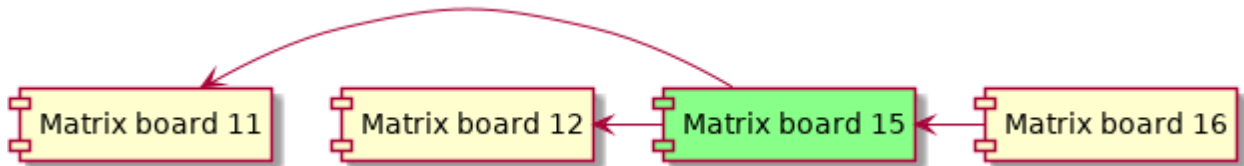
For the disconnect duration, there was only a difference of 2%. This is probably due to the deviation between the results. The result of the disconnect duration can be between the 1.5s and 2.5s. Therefore, the average result is expected to be around 2 seconds.

There is a huge difference between the CPU usages. The measurement with 4 devices uses over 2x more CPU on average. As shown in the UML diagrams below, the measurement with 4 devices has two more devices connected to the master. This is expected to use more CPU. This is also probably the reason that the physical and virtual memory usage is a bit higher on the 4 devices measurement.

**Communication matrix board 15 with 2 devices:**



**Communication matrix board 15 with 4 devices:**



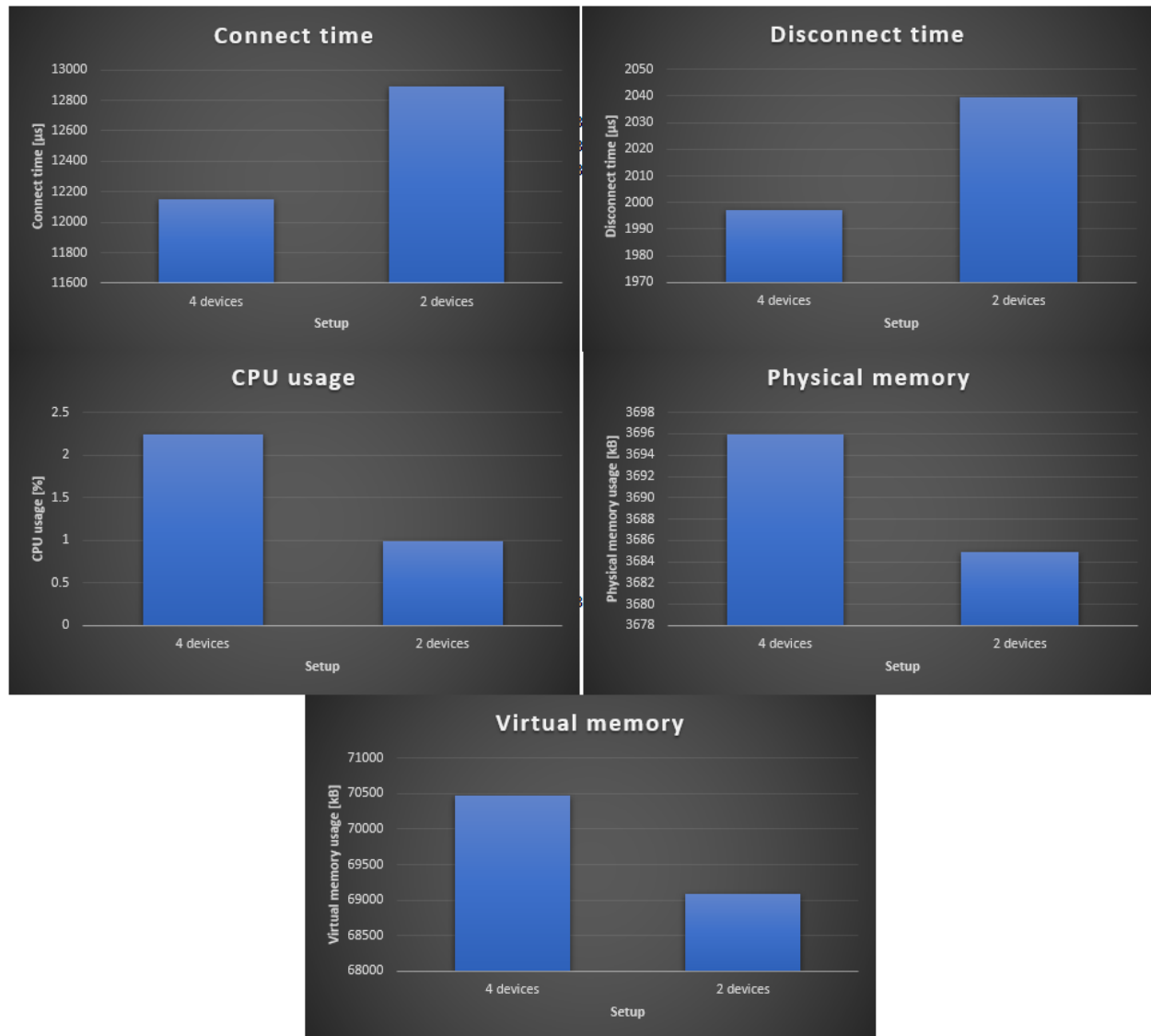
---

#### **Note:** Reliability

The reliability of the test is not high because the results are highly probably accountable by how the matrix board application works instead of the scalability of DDS. This test should be re-executed with a minimal of 5 devices on the network. The master with 2 slaves with an ID below the master should be connected with an ethernet cable. The other devices may be connected using Wi-Fi (the connect and disconnect durations are not affected then). The second test should be executed with >5 devices.

---





## Links

- [Durability](#)
- [Durability service](#)
- [Default QoS policies Cyclone DDS.](#)

## Performance measurements data analysis

**authors** Joost Baars

**date** June 2020

### Description

This page describes how the data of the matrix board performance measurement application can easily be analyzed (see application: *Matrixboard performance application*). This page contains a method that can easily calculate the average of each configuration, the only part that needs to be done manually is creating graphs (if wanted).

### Requirements

This page is tested on Microsoft Excel for Office 365. The code shown on this page should also work for other versions of Excel, but the steps will probably be different.

- The CSV results of the performance application
- Microsoft Excel

### Analyzing the data

Follow the steps below for easy calculating the average values of each configuration. Also, the maximum deviation is calculated using these steps. This method is only useful if each configuration is executed at least twice.

1. Open the CSV results with Excel.
2. Go to the `Developer` tab in Excel (in the top bar)
3. Click on `Insert`
4. Select `Command Button (ActiveX Control)` under `ActiveX Controls`
5. Click somewhere on the Excel spreadsheet to create the command button
6. Right click the command button and select `View Code`
7. Copy the code below
8. Paste the code between `Private Sub CommandButton1_Click()` and `End Sub`

```
Dim i As Integer
Dim Copyrange As String
Dim String1 As String
Dim Name As String
Dim Connect As String
Dim Disconnect As String
Dim CPU As String
Dim Phys As String
Dim Virt As String
Dim start As Integer
Dim sizeMeasurement As Integer
Dim storeY As Integer
Dim FirstFind As Integer
MsgBox ActiveSheet.UsedRange.Rows.Count

FirstFind = 0
start = 1
sizeMeasurement = 1
storeY = 2

Cells(1, 10) = "Configuration"
```

(continues on next page)

(continued from previous page)

```
Cells(1, 11) = "Connect duration average"
Cells(1, 12) = "Disconnect duration average"
Cells(1, 13) = "CPU usage average"
Cells(1, 14) = "Physical memory usage average"
Cells(1, 15) = "Virtual memory usage average"
Cells(1, 17) = "Connect duration maximum deviation"
Cells(1, 18) = "Disconnect duration maximum deviation"
Cells(1, 19) = "CPU usage maximum deviation"
Cells(1, 20) = "Physical memory usage maximum deviation"
Cells(1, 21) = "Virtual memory usage maximum deviation"

For i = 1 To ActiveSheet.UsedRange.Rows.Count
    Let Copyrange = "A" & i
    If IsNumeric(Cells(i, 1).Value) = False Or i = ActiveSheet.UsedRange.Rows.Count
        Then
            Let String1 = "J" & storeMeasurement
            If FirstFind = 1 Then
                Cells(storeY, 10) = Cells(start, 1)
                Cells(storeY, 11) = Application.Average(Range(Cells(start, 2), Cells(i - 1, 2)).Value)
                Cells(storeY, 12) = Application.Average(Range(Cells(start, 5), Cells(i - 1, 5)).Value)
                Cells(storeY, 13) = Application.Average(Range(Cells(start, 7), Cells(i - 1, 7)).Value)
                Cells(storeY, 14) = Application.Average(Range(Cells(start, 8), Cells(i - 1, 8)).Value)
                Cells(storeY, 15) = Application.Average(Range(Cells(start, 9), Cells(i - 1, 9)).Value)
                Cells(storeY, 17) = Application.Max(Range(Cells(start, 2), Cells(i - 1, 2)).Value) - Application.Min(Range(Cells(start, 2), Cells(i - 1, 2)).Value)
                Cells(storeY, 18) = Application.Max(Range(Cells(start, 5), Cells(i - 1, 5)).Value) - Application.Min(Range(Cells(start, 5), Cells(i - 1, 5)).Value)
                Cells(storeY, 19) = Application.Max(Range(Cells(start, 7), Cells(i - 1, 7)).Value) - Application.Min(Range(Cells(start, 7), Cells(i - 1, 7)).Value)
                Cells(storeY, 20) = Application.Max(Range(Cells(start, 8), Cells(i - 1, 8)).Value) - Application.Min(Range(Cells(start, 8), Cells(i - 1, 8)).Value)
                Cells(storeY, 21) = Application.Max(Range(Cells(start, 9), Cells(i - 1, 9)).Value) - Application.Min(Range(Cells(start, 9), Cells(i - 1, 9)).Value)

                storeY = storeY + 1
            End If
            start = i
            FirstFind = 1
        End If
    End If
Next i
```

8. Click ctrl + s on the keyboard to save the code (or use the save button in the bar on the top)
9. Close the code window
10. Within the Excel spreadsheet, go to the Developer tab again
11. Deselect the Design Mode button by pressing on it (only do this when the button is selected)
12. Click on the created button within the Excel spreadsheet

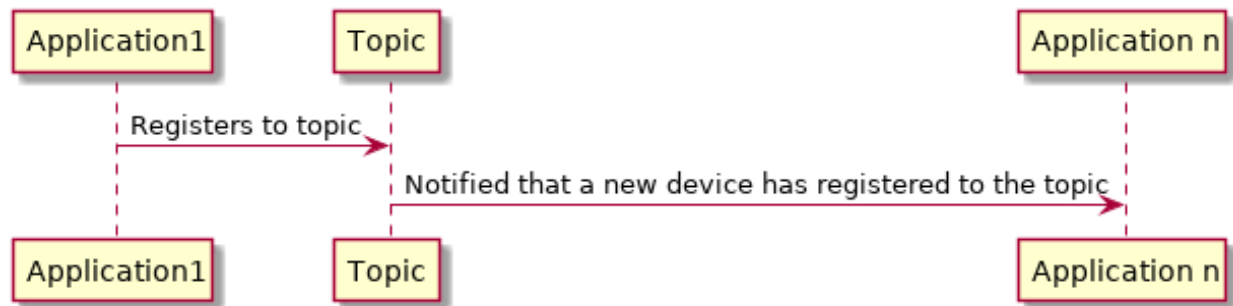
Data should be generated by clicking on the button.

The average data should now appear on the top of the Excel spreadsheet (on the right of the csv results). If the button is placed on the data, the button can be moved by selecting the *Design Mode* within the *Developer* tab.

The meaning of the data is also generated using this script. The maximum deviation is the deviation between the lowest and highest value within the measurements.

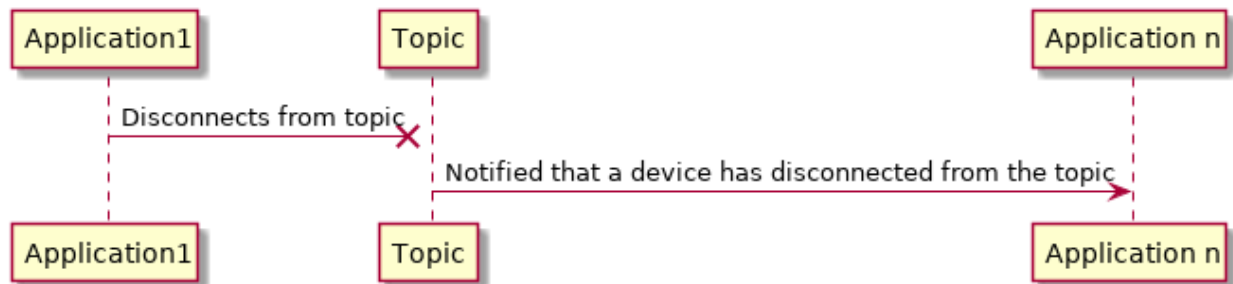
### Connect duration

The connect duration is measured between registering to a topic and being notified that a device is connected to the topic. This principle is displayed below for *Application1*. In this figure, the process that determines the duration between registering and other devices being notified is displayed. The duration between the start of the “registers to topic” arrow and the end of the “notified” arrow is the connect duration.



### Disconnect duration

The disconnect duration is measured between disconnecting from a topic and being notified that a device has disconnected from the topic. This principle is displayed below for *Application1*. In this figure, the process that determines the duration between disconnecting and other devices being notified is displayed. The duration between the start of the “Disconnects from topic” arrow and the end of the “notified” arrow is the disconnect duration.



### Helper pages

The following pages are written for explaining some topics. This information does not contain information needed for the performance measurements, but contains information that could be useful for debugging or learning more about the tools.

### systemctl/systemd basics

**authors** Joost Baars

date June 2020

## Description

This page describes some information about `systemctl` and `systemd`.

### systemctl commands

`systemctl` contains some useful commands for managing services (applications that can be run). An example of creating a service is shown in [Matrix board performance application system setup](#) for creating `ptpd` as a service.

Some useful commands are shown below. In these examples, the `<servicename>` tag should be replaced with a service name.

The command below requests the status of a service. In this status, the logging can be seen from the service. Also, it is shown if a service is running, stopped, crashed, or if it is never started in the first place.

```
systemctl status <servicename>
```

The commands below enable or disable a service. Only one of these commands should be used at a time. By enabling a service, the service is started upon boot. By disabling a service, the service is not started upon boot.

```
sudo systemctl enable <servicename>
sudo systemctl disable <servicename>
```

The commands below start, stop or restart a service. Only one of these commands should be used at a time. These commands control a service manually. With the start command, a service can manually be started. With the stop command, it can be stopped. And as you probably guessed, with the restart command, a service can be restarted.

```
sudo systemctl start <servicename>
sudo systemctl stop <servicename>
sudo systemctl restart <servicename>
```

### systemd information

With various `systemd` commands, the boot time and boot order can be shown. This can be useful if you want to speed up the boot time on a device or if you want to start a service after another service has been initialized (which is used in the section above).

#### Boot time

The boot time can be visualized using the following command:

```
systemd-analyze
```

This command shows the kernel setup time and the service initialization time. Each application and their influence of this time can be shown with the following command:

```
systemd-analyze blame
```

Using this command, the influence of every service is shown. This way, you can remove or reduce specific applications that take long to configure during boot.

### Boot order

The boot order can also be useful. The order can be adjusted. If you add a service manually, you can place it before other applications. This way, you can reduce the time it takes to execute that service upon boot. The following command stores the boot order in a svg file (named `bootup.svg`).

```
systemd-analyze plot > bootup.svg
```

If you run on a system without a GUI, this file should be transferred to your computer (for example using FTP).

### Links

- [More information about Systemctl](#)

### Links

- General information regarding QoS within DDS: <https://community.rti.com/glossary/qos>
- List of all QoS policies within DDS: [https://community.rti.com/rti-doc/500/ndds.5.0.0/doc/pdf/RTI\\_CoreLibrariesAndUtilities\\_QoS\\_Reference\\_Guide.pdf](https://community.rti.com/rti-doc/500/ndds.5.0.0/doc/pdf/RTI_CoreLibrariesAndUtilities_QoS_Reference_Guide.pdf)
- List of all QoS policies (less information but with hyperlinks): [https://community.rti.com/static/documentation/connex-dds/5.2.0/doc/manuals/connex-dds/html\\_files/RTI\\_ConnextDDS\\_CoreLibraries\\_UsersManual/Content/UsersManual/QoS\\_Policies.htm](https://community.rti.com/static/documentation/connex-dds/5.2.0/doc/manuals/connex-dds/html_files/RTI_ConnextDDS_CoreLibraries_UsersManual/Content/UsersManual/QoS_Policies.htm)
- Some examples given by RTI: [https://community.rti.com/static/documentation/connex-dds/5.2.0/doc/manuals/connex-dds/html\\_files/RTI\\_ConnextDDS\\_CoreLibraries\\_UsersManual/Content/UsersManual/ControllingBehavior\\_withQoS.htm](https://community.rti.com/static/documentation/connex-dds/5.2.0/doc/manuals/connex-dds/html_files/RTI_ConnextDDS_CoreLibraries_UsersManual/Content/UsersManual/ControllingBehavior_withQoS.htm)

### MQTT C++

**authors** Sam Laan

**date** March 2020

### Description

MQTT stands for Message Queuing Telemetry Transport. MQTT is a lightweight publish/subscribe messaging protocol [MQTT:online]. It was designed to be used with constrained devices and low-bandwidth, high-latency or unreliable networks. It was developed by IBM and standardized by OASIS. MQTT works using a client and a broker. The client publishes messages to a topic on a broker, clients that are subscribed to this topic get these messages from the broker. These messages can also be retained for future subscribers. Clients can subscribe to multiple topics and receive every message published to each topic. There are multiple MQTT clients and brokers available.

### MQTT install

For MQTT the Mosquitto broker is used, this broker was chosen because it has the best throughput, least latency and is the most popular broker.

```
sudo apt-add-repository ppa:mosquitto-dev/mosquitto-ppa
sudo apt-get update
sudo apt-get install mosquitto
```

For the first version paho MQTT was used, which can be installed by performing the following steps (check the readme of their GitHub page for updates): firstly you need to install the following packages:

```
sudo apt-get install build-essential gcc make cmake cmake-gui cmake-curses-gui
sudo apt-get install libssl-dev
sudo apt-get install libcppunit-dev
```

After this the c library of paho needs to be installed:

```
git clone https://github.com/eclipse/paho.mqtt.c.git
cd paho.mqtt.c
cmake -Bbuild -H. -DPAHO_WITH_SSL=ON -DPAHO_ENABLE_TESTING=OFF
sudo cmake --build build/ --target install
sudo ldconfig
```

This builds with SSL/TLS enabled. If that is not desired, omit the `-DPAHO_WITH_SSL=ON`. After this, the C++ library can be installed.

```
git clone https://github.com/eclipse/paho.mqtt.cpp
cd paho.mqtt.cpp
cmake -Bbuild -H. -DPAHO_BUILD_DOCUMENTATION=TRUE -DPAHO_BUILD_SAMPLES=TRUE
sudo cmake --build build/ --target install
sudo ldconfig
```

The library can now be used and should be linked as shown below using CMAKE:

```
find_package(PahoMqttCpp REQUIRED)
target_link_libraries({yourexecutable} paho-mqttpp3 paho-mqtt3as)
```

## MQTT solutions

This list contains the solutions regarding MQTT.

- *MQTT Matrixboard*

## Links

- List of brokers: <https://github.com/mqtt/mqtt.github.io/wiki/brokers>
- List of libraries: <https://github.com/mqtt/mqtt.github.io/wiki/libraries>
- Effects of QoS standards: [https://www.researchgate.net/profile/Shinho\\_Lee4/publication/261230513\\_Correlation\\_analysis\\_of\\_MQTT\\_loss\\_and\\_delay\\_according\\_to\\_QoS\\_level/links/5dc93fcb458515143500e981/Correlation-analysis-of-MQTT-loss-and-delay-according-to-QoS-level.pdf](https://www.researchgate.net/profile/Shinho_Lee4/publication/261230513_Correlation_analysis_of_MQTT_loss_and_delay_according_to_QoS_level/links/5dc93fcb458515143500e981/Correlation-analysis-of-MQTT-loss-and-delay-according-to-QoS-level.pdf)
- Comparison brokers: <http://www.shudo.net/publications/201709-CloudNet-2017-hetero-MQTT-brokers/banno-CloudNet-2017-hetero-MQTT-brokers.pdf>
- Github page paho.mqtt.cpp: <https://github.com/eclipse/paho.mqtt.cpp>

### ZMQ C++

**authors** Sam Laan

**date** March 2020

### Description

ZeroMQ (also known as ØMQ, 0MQ, or zmq) is a message-oriented middleware. It is used in various environments for example in embedded systems, aerospace and financial services. ZeroMQ provides a library, not a messaging server. It gives sockets that carry atomic messages across various transports like in-process, inter-process, TCP, and multicast. Each socket can handle communication with multiple peers. ZeroMQ allows these sockets to be connected through different patterns. The protocol uses ZMTP(ZeroMQ Message Transport Protocol) on the wire which makes use of message batching, asynchronous communication and supports zero-copy. ZeroMQ core is written in C/C++ and has bindings for most modern programming languages and operating systems.

### ZMQ install

To use ZeroMQ the C library has to be installed this can be done like so:

```
git clone https://github.com/zeromq/libzmq
cd libzmq
mkdir build
cd build
sudo cmake ..
sudo make install
```

After this the C++ lib can be installed. The cppzmq was chosen because it's the most lightweight, the most used and the best maintained C++ lib.

```
git clone https://github.com/zeromq/cppzmq
cd cppzmq
mkdir build
cd build
sudo cmake ..
sudo make install
```

### ZMQ solutions

This list contains the solutions regarding ZMQ and C++.

- *[ZMQ C++ Matrixboard](#)*

### Links

- List of bindings: [http://wiki.zeromq.org/bindings:\\_start](http://wiki.zeromq.org/bindings:_start)

### Excel

**authors** Sam Laan



**date** April 2020

## Description

Excel is used in some cases to visualise data collected within performance measurements. Used Excel features and techniques will be described. These descriptions can be found using the links underneath.

## Features and techniques

This list contains the findings regarding the QoS policies.

## Power Pivot

**authors** Sam Laan

**date** April 2020

## Description

This page shows information about using a CSV file with Power Pivot.

## Enabling Power Pivot

Power Pivot is not enabled by default in Excel and is also not available when using a CSV file. To enable Power Pivot, the following steps should be followed:

- Press file in the top bar.
- Select options.
- Select Add-ins.
- Select COM Add-ins in the bottom dropdown and click go.
- Check the box next to Power Pivot for Excel.

Congratulations! you have enabled Power Pivot.

## Importing a CSV File

Power Pivot is capable of importing data from various sources. These are the steps to import a CSV file.

- Choose Power Pivot in the top bar.
- Press manage
- Select Get External Data
- Select from other sources
- Scroll down and select Text File
- Browse and select the desired CSV
- Choose the applicable column separator

- Use the first row as headers if applicable
- Press finish

The CSV file will now be imported. The data from the file can be refreshed using the refresh button. To get a pivot table of your data on your Excel sheet press PivotTable. This pivot table can be used to filter data and as a source for graphs. You can also add measures.

### Measures

Measures can be used to create calculated columns within the data. DAX syntax is used to create measures click on your pivot table. To find out more about DAX syntax please look at the syntax reference (<https://docs.microsoft.com/nl-nl/dax/dax-syntax-reference>).

### Nucleo Setup

**authors** Furkan Ali Yurdakul

**date** April 2020

### Description

The Nucleo-F767ZI is a development board that will be used to demonstrate the DDS implementation on embedded bare-metal hardware.

The user will be guided through some steps, to make sure the hardware is correctly working and the user is capable of making use of the required features. Make sure the Nucleo is connected to the computer, when making use of a terminal or flashing the code.

From now on the Nucleo-F767ZI will be shortened to Nucleo.

More information about the Nucleo can be found in the [Links](#).

### Nucleo Initialization

This list contains guides to set up different parts of the Nucleo.

### How to setup the Nucleo

**authors** Furkan Ali Yurdakul

**date** March 2020

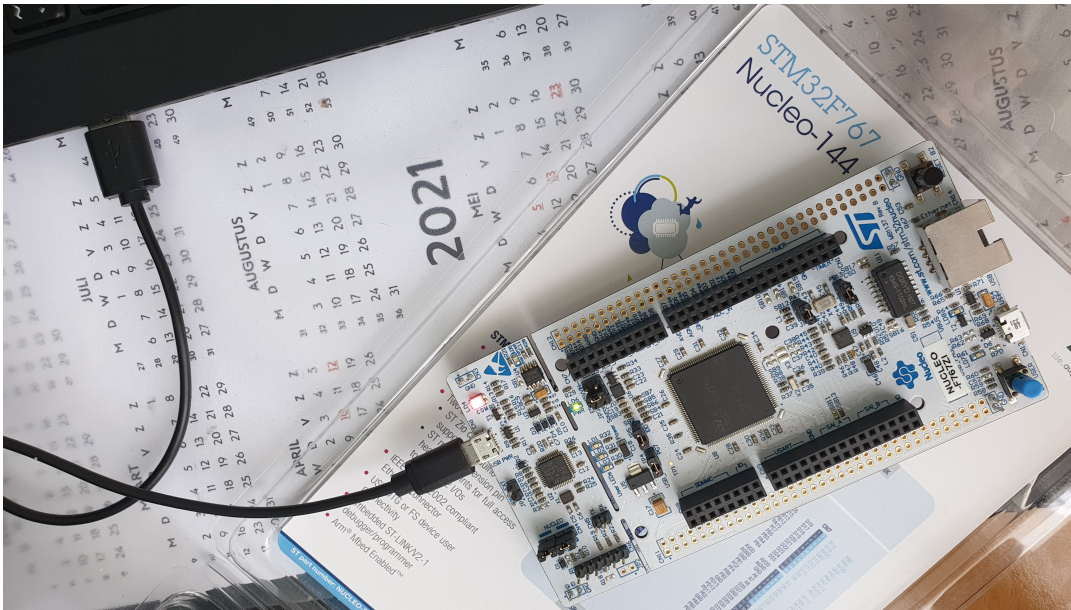
In this page, there will be a step by step tutorial to make sure your Nucleo is set up and ready to be programmed.

**For this tutorial the following products have been used:**

- Nucleo
- Lenovo Thinkpad T470 (Windows)
- Micro USB cable
- STM32CubeIDE

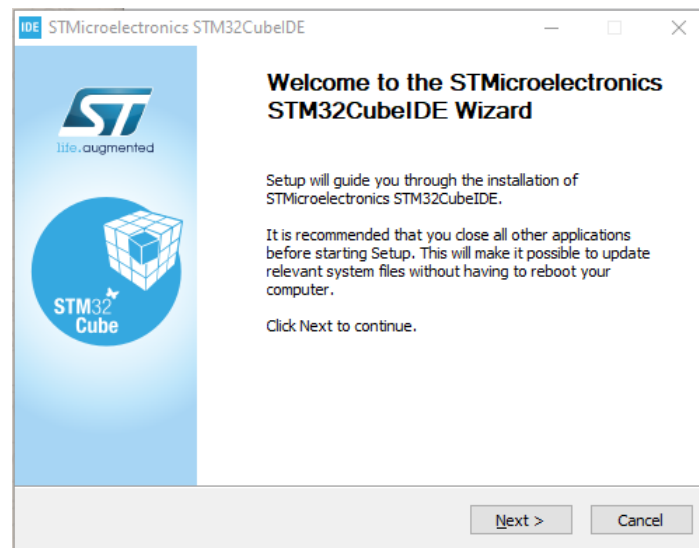
## Install programming environment

First, connect the Nucleo with a micro USB cable to your PC.

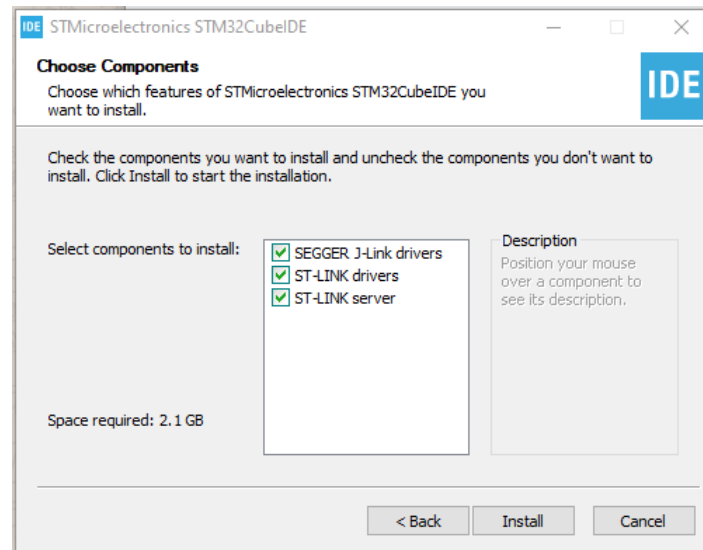


To be able to program the Nucleo the standard programming environment is used, called the STM32CubeIDE. This IDE can be downloaded via this link: <https://www.st.com/en/development-tools/stm32cubeide.html>.

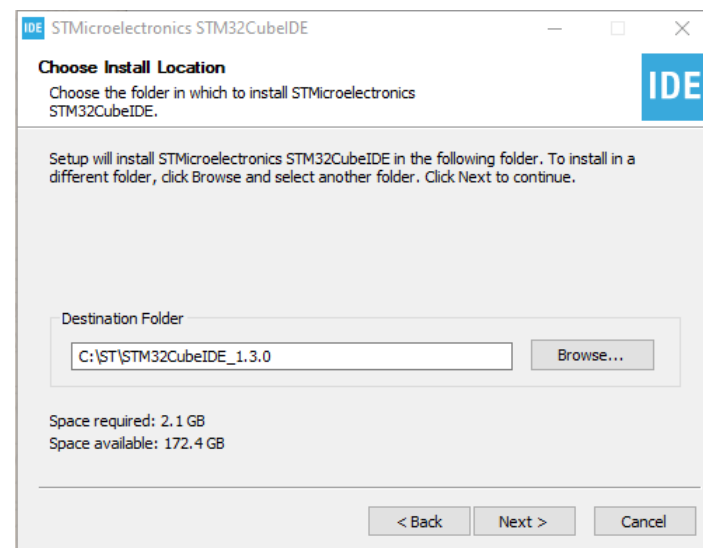
First, open the downloaded executable to start the installation.



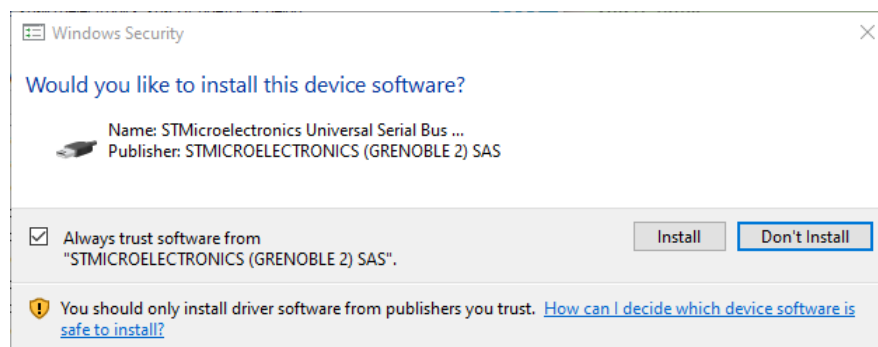
Within the installation, it is decided to install all the components as seen below.



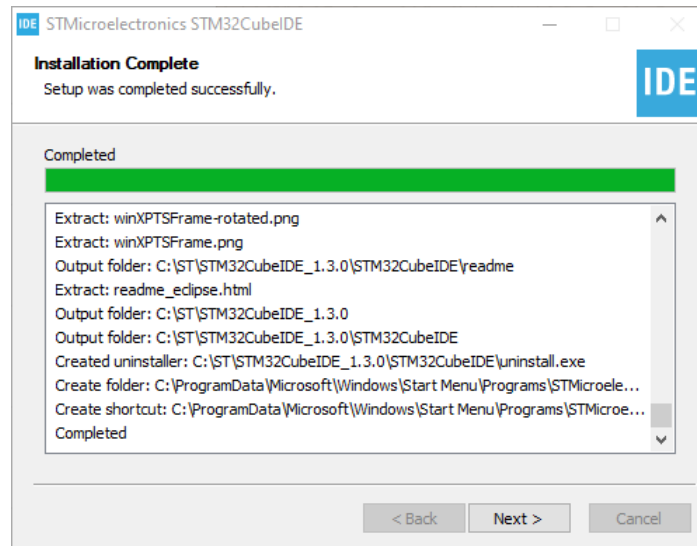
The destination folder should not matter, but for this tutorial it has been chosen to use the default destination.



If asked to install Universal Serial Bus software, click “Install”.



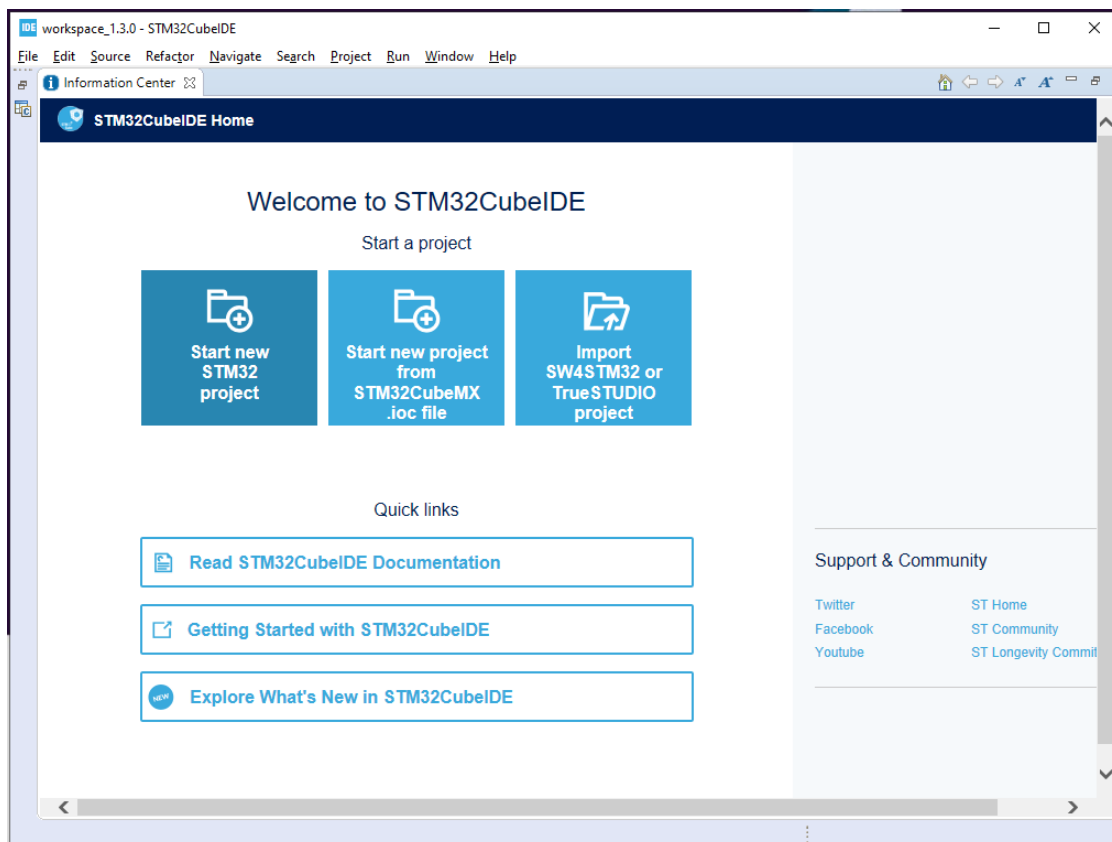
This is how it should look like when the installation of the driver is finished.



If everything went as expected, continue with the next step. If not, uninstall the STM32Cube IDE and redo these steps.

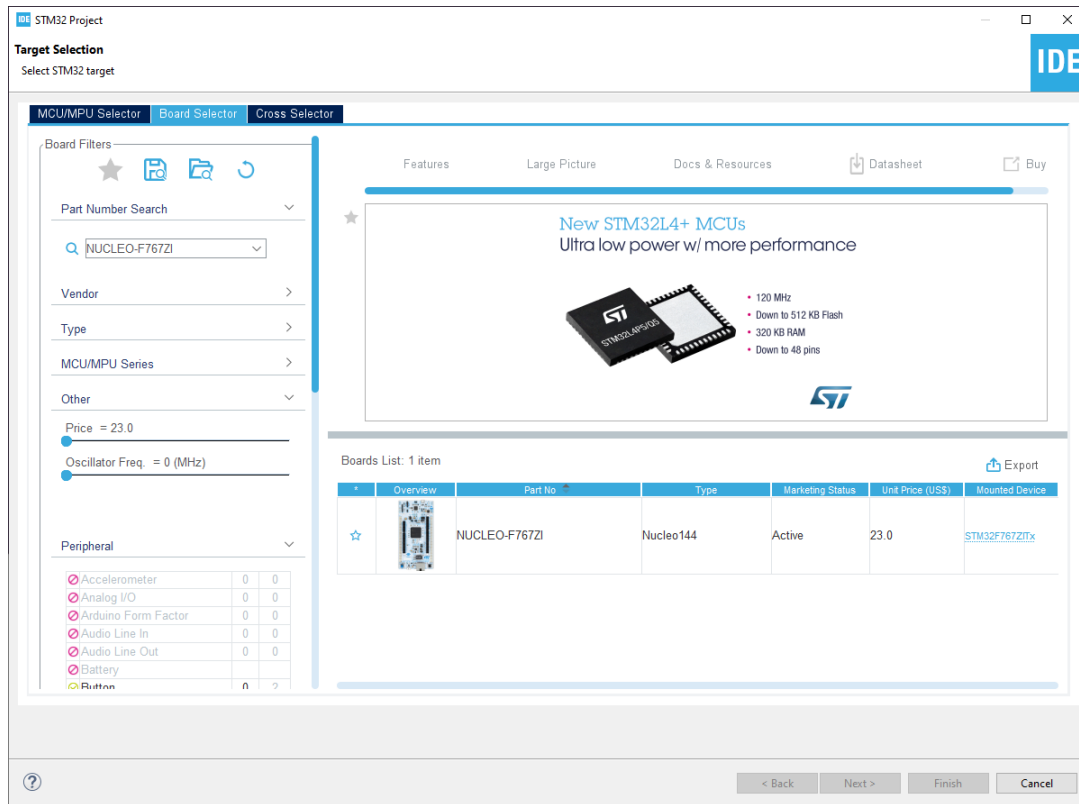
## Start a new project

After opening the IDE, the starting screen will look like the picture below. Click on the left top button to start a new project.



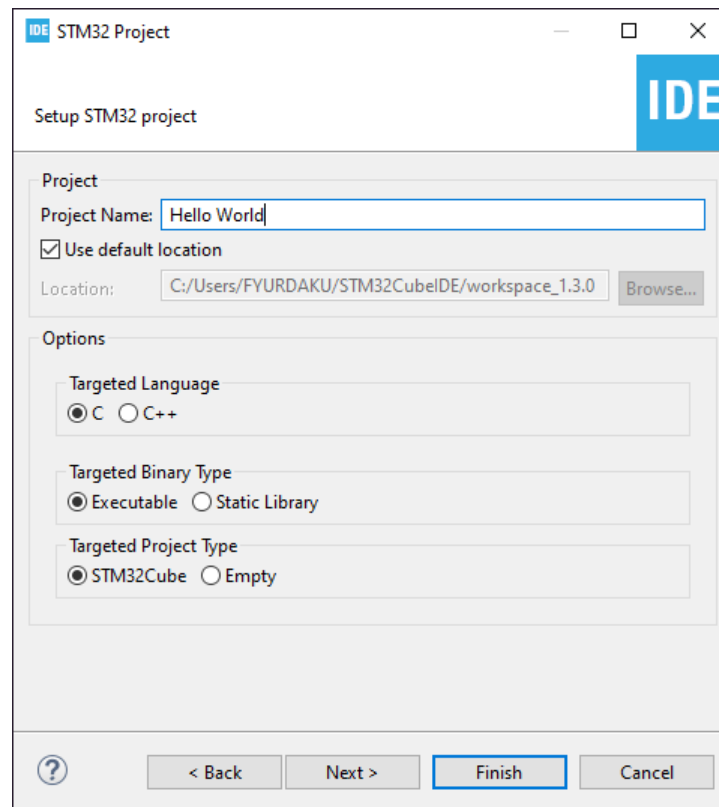
After starting a new project, the IDE will ask what the project is targeted for. In this case, the project is targeted

towards the Nucleo-F767ZI. To target the project for this specific board, click on board selector and search for “Nucleo-F767ZI”. Thereafter click on the star next to the board and click finish.

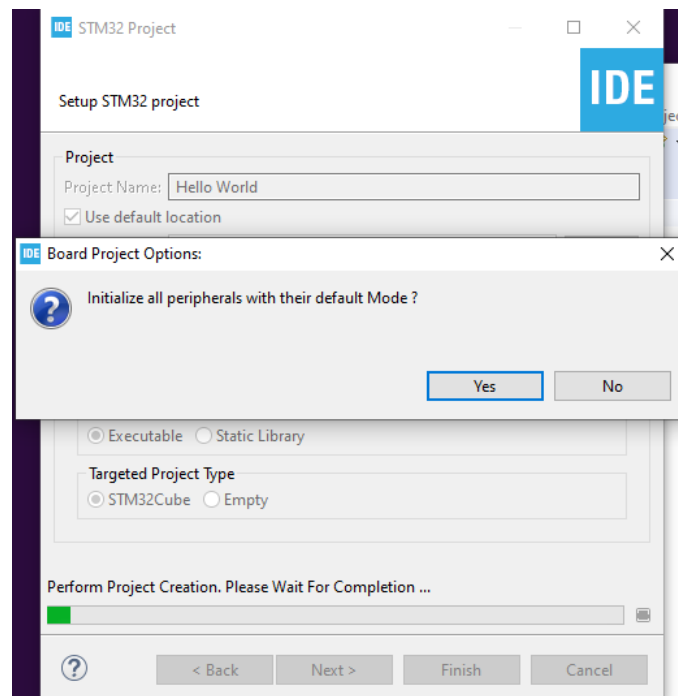


The project needs a name, any name will do. But for this project, it’s chosen to call it “HelloWorld” as this project will be used for more simple tests in the future.

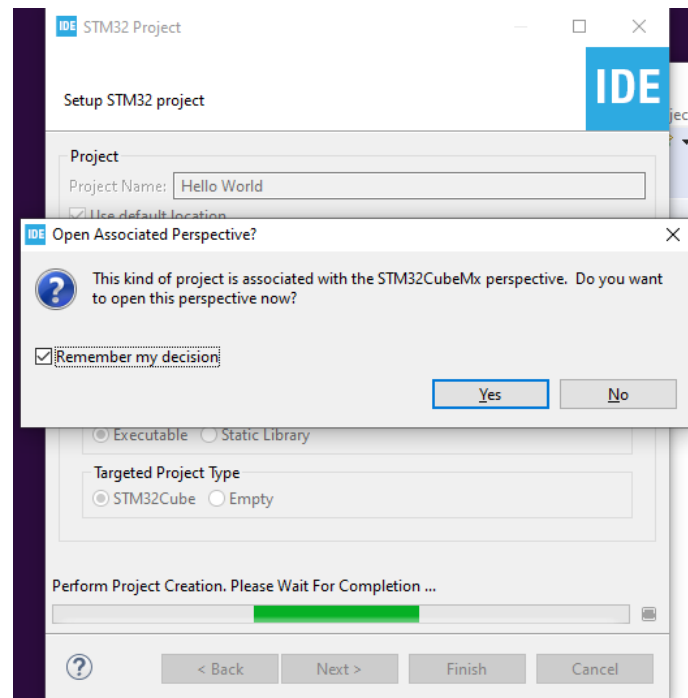
For this tutorial, the options selected in the picture below will be used.



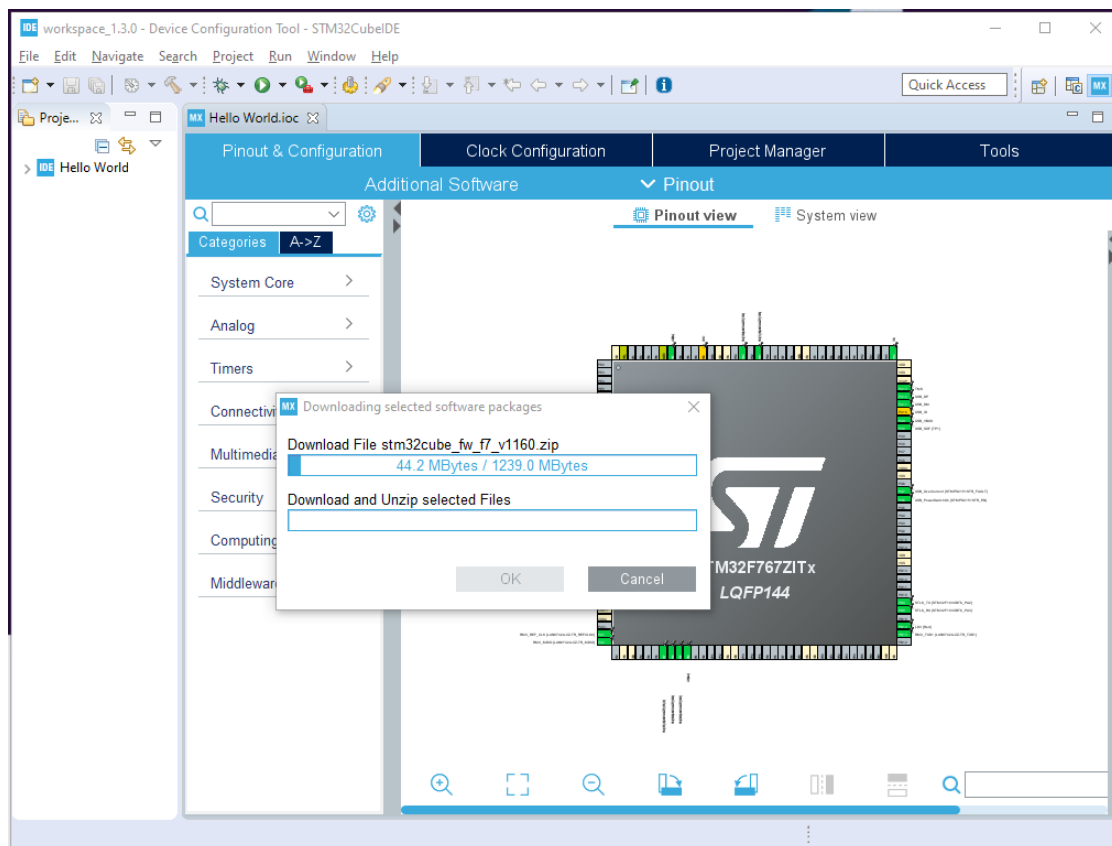
The setup will ask to initialize everything in its default mode. For this tutorial, it's chosen to make use of this. In case of additional research, it will make it easier for the user when using the default initialization.



Accept opening the project in the STM32CubeMx perspective.



After finishing the setup, the IDE will start downloading and loading all the needed files and it should look like the picture down below. When this is done the IDE is finished setting up.

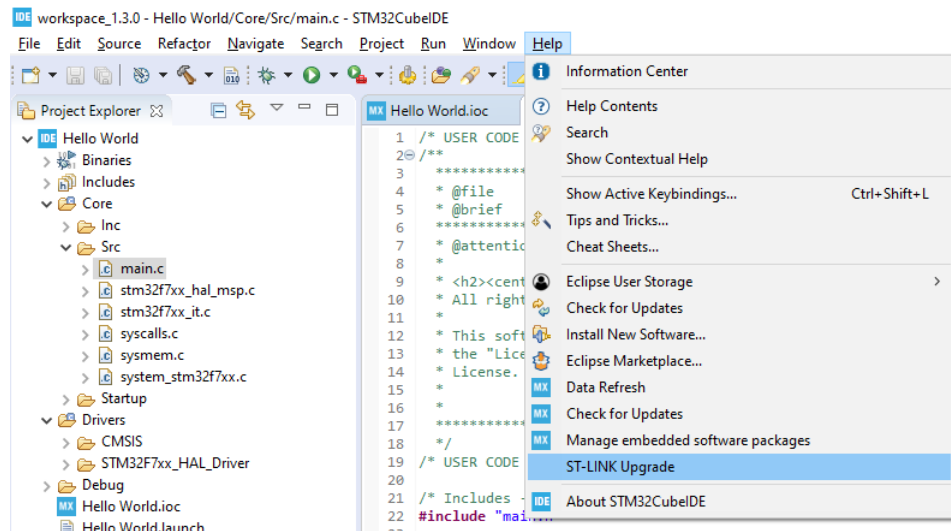


If everything went as expected, continue with the next step. If not, delete the project and redo this step.

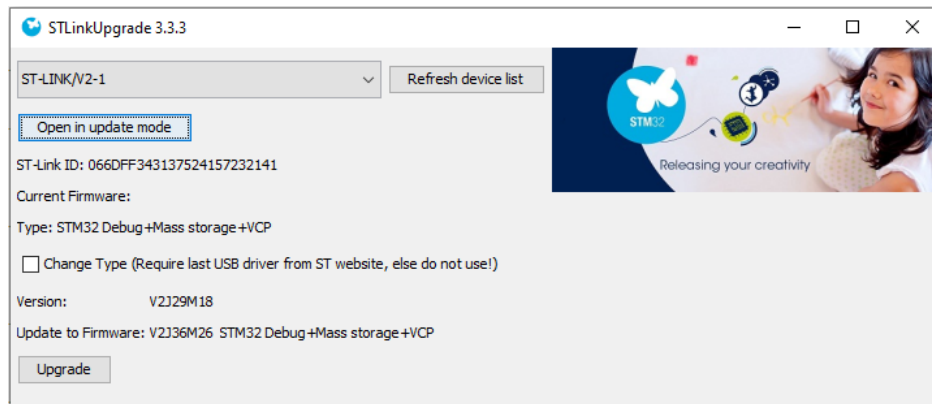


## Update drivers

When the project is finished building it is not yet able to program the Nucleo. This is because the drivers that are installed are not up to date. To update the drivers click on the help button and afterward click on ST-LINK Upgrade.



The tab that it opens requires it to be in update mode, to enable update mode click on the button “Open in update mode”. Afterward, click on the “Upgrade” button to start the installation and wait until it finishes.

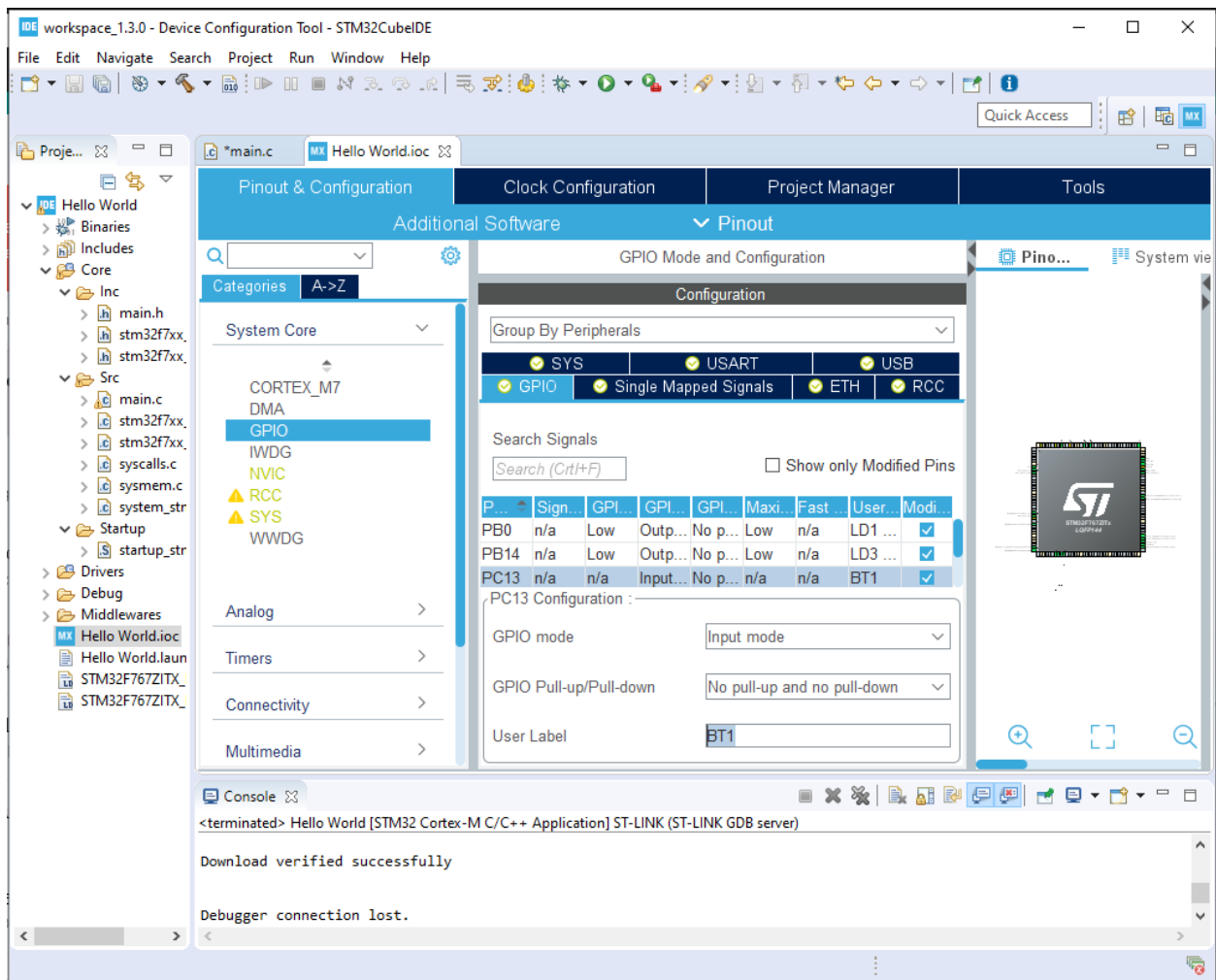


If everything went as expected, continue with the next step. If not, redo this step.

## Configure pin-out for onboard button

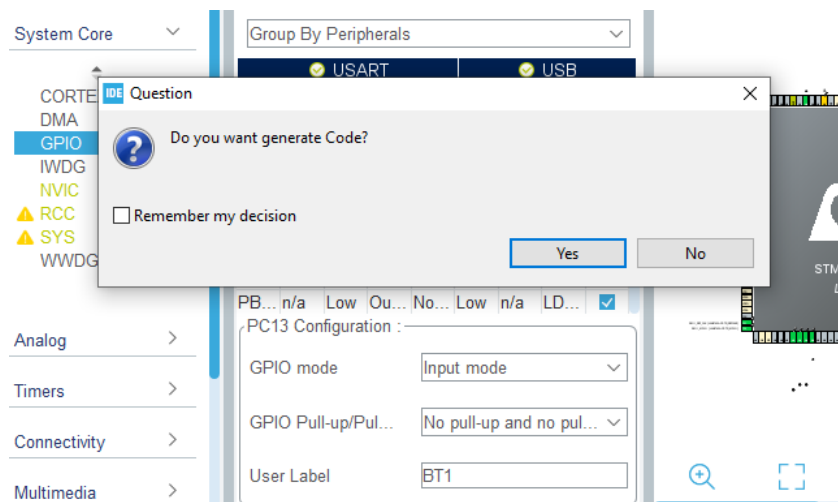
To test if the Nucleo isn't broken and can be programmed, the onboard button(BT1) will be used to toggle the red LED(LD3).

The configurations of these pin-outs can be done using the IDE controller interface. Open HelloWorld.ioc to open the interface. Afterward, click on the System Core category and select GPIO to find the configurations for the pin-outs.



Pin PC13 is connected to the onboard button and pin PB14 is connected to the red LED. Pin PB14 should already have a user label “LD3”. To make it easier to program pin PC13, PC13 will get the user label “BT1”.

When done, save the interface via “CTRL+S” buttons and click yes to generate the code for it.



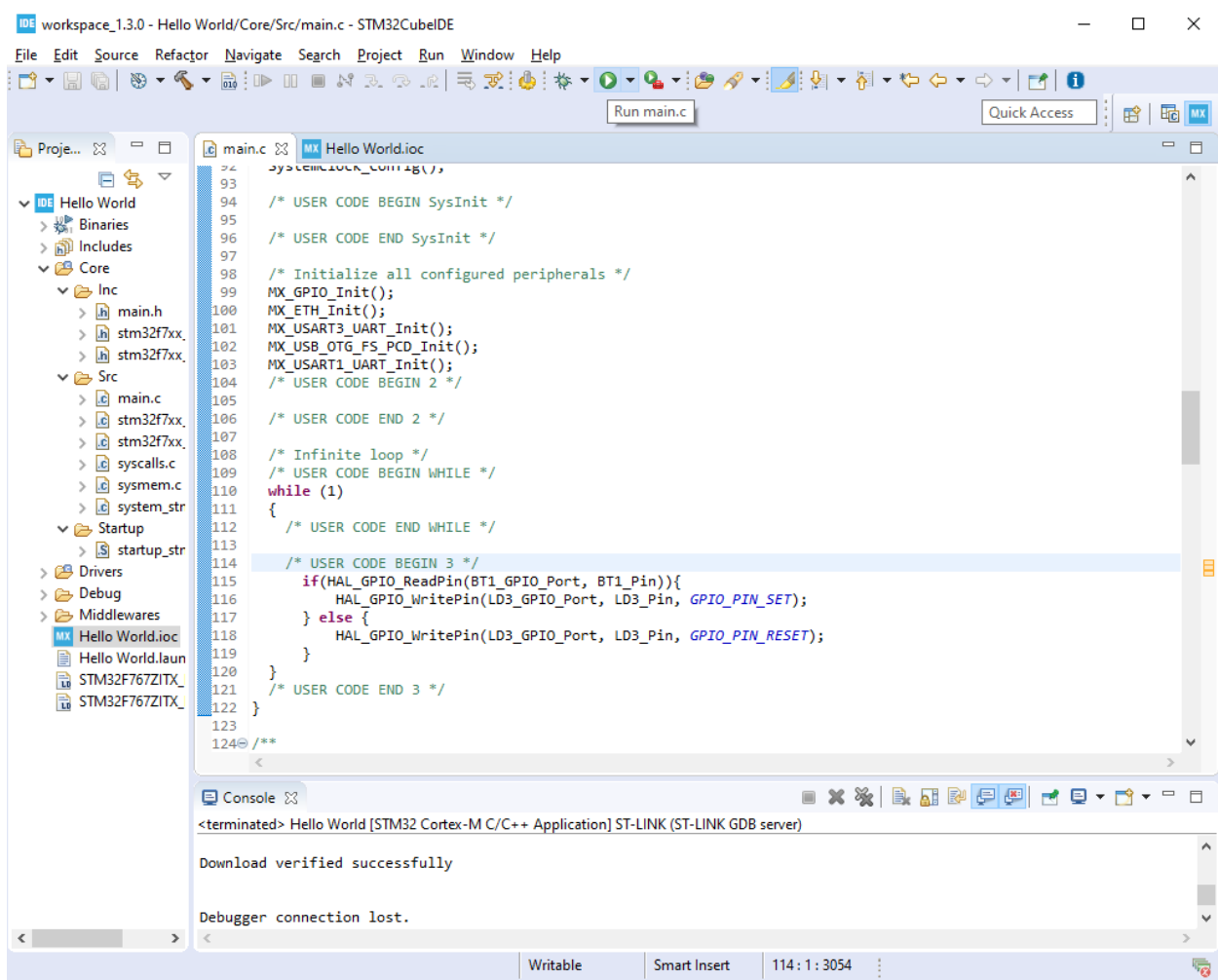
If everything went as expected, continue with the next step. If not, redo this step.

## Code flashing

Add the following code to your while loop:

```
if(HAL_GPIO_ReadPin(BT1_GPIO_Port, BT1_Pin)){
    HAL_GPIO_WritePin(LD3_GPIO_Port, LD3_Pin, GPIO_PIN_SET);
} else {
    HAL_GPIO_WritePin(LD3_GPIO_Port, LD3_Pin, GPIO_PIN_RESET);
}
```

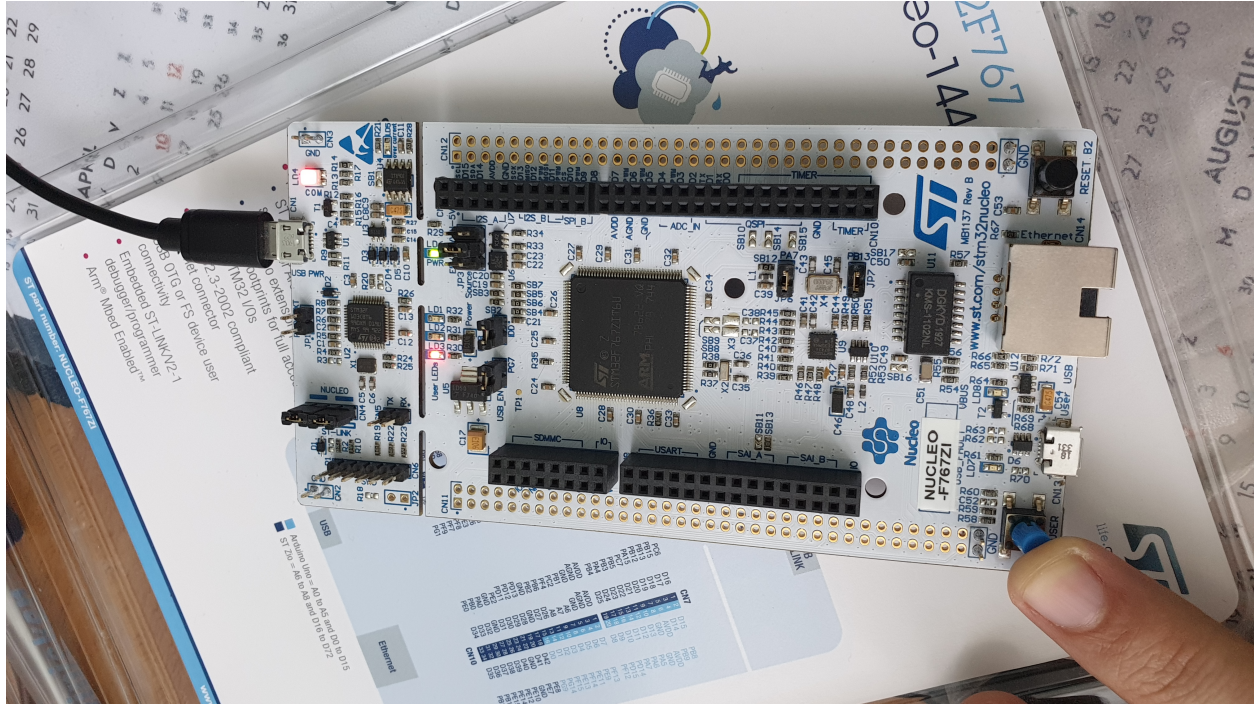
When done adding click on Run code and wait till it's finished. The Run code can be found on the top of the IDE see the picture below.



If everything went as expected, test the board and continue with the next step. If not, delete all written code and redo this step.

## End result

If everything went successful you should be able to click on the onboard button to toggle a led.



If this is not happening restart process from *Configure pin-out for onboard button*. If it did work, this means your board is successfully set and is ready for future projects.

Check “*Serial Communication with the Nucleo*” to learn how to make serial communication with the Nucleo.

### Serial Communication with the Nucleo

**authors** Furkan Ali Yurdakul

**date** April 2020

In this page, there will be a step by step tutorial to make sure your Nucleo can make use of USART to send data to your computer with the use of a micro USB cable. The expected result is to receive the same message back that will be sent from a computer.

**For this tutorial the following products have been used:**

- Nucleo
- Lenovo Thinkpad T470 (Windows)
- Micro USB cable
- STM32CubeIDE
- Hercules (Terminal)

### Hardware Pinout

The Nucleo can have multiple USART communications, these can be found in the [datasheet](#). The one that will be used for serial communication between a computer and a Nucleo is USART3.

The reason behind this is because the datasheet specifically tells that USART3 uses the output for ST-Link, also known as the port where the micro USB is connected.

USART3 can also be reconfigured to make use of the ST morpho output. This can be done if a specific solder bridge is changed, see the image below. But since this is not preferred, the default configuration for the solder bridges will be used. If your board doesn't have the default configuration, see the image below which settings are needed to use ST-Link as the output for USART3.

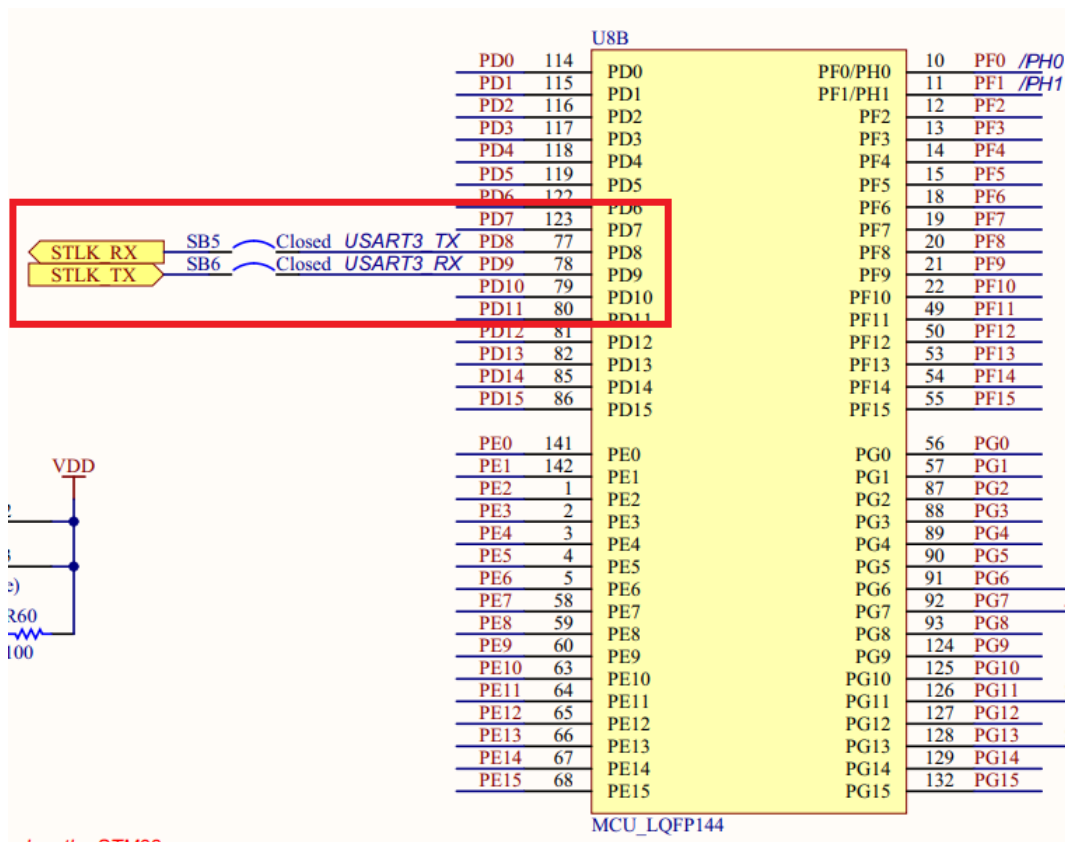
## 6.9 USART communication

The USART3 interface available on PD8 and PD9 of the STM32 can be connected either to ST-LINK or to ST morpho connector. The choice is changed by setting the related solder bridges. By default the USART3 communication between the target STM32 and the ST-LINK is enabled, to support the virtual COM port for the mbed (SB5 and SB6 ON).

Table 9. USART3 pins

Pin name	Function	Virtual COM port (default configuration)	ST morpho connection
PD8	USART3 TX	SB5 ON and SB7 OFF	SB5 OFF and SB7 ON
PD9	USART3 RX	SB6 ON and SB4 OFF	SB6 OFF and SB4 ON

See the following image for the hardware pinout how the pins are connected to ST-link.

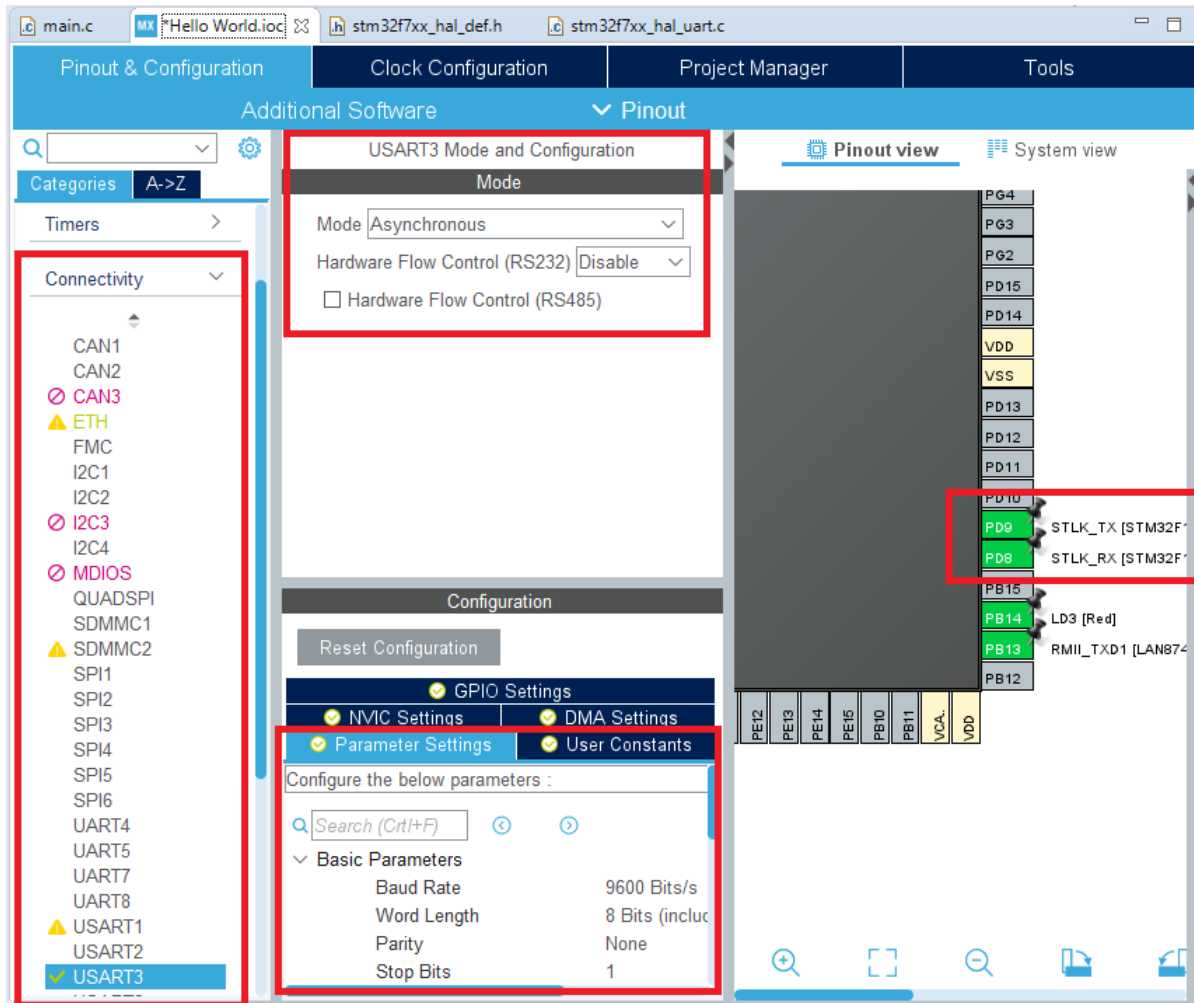


The next step is to initialize the USART3 in the Nucleo.

## Initialize USART3

To initialize USART3 for the Nucleo, you simply need to open the “.ioc” file in the STM32CubeIDE. The USART3 can be found in the Connectivity section, click on it. Set mode to Asynchronous and change the Baud Rate to 9600 Bits/s. Asynchronous is used, because there is no clock signal used for the communication. The Baud Rate is lowered to ensure stability over speed, because speed is not a necessity in for this tutorial.

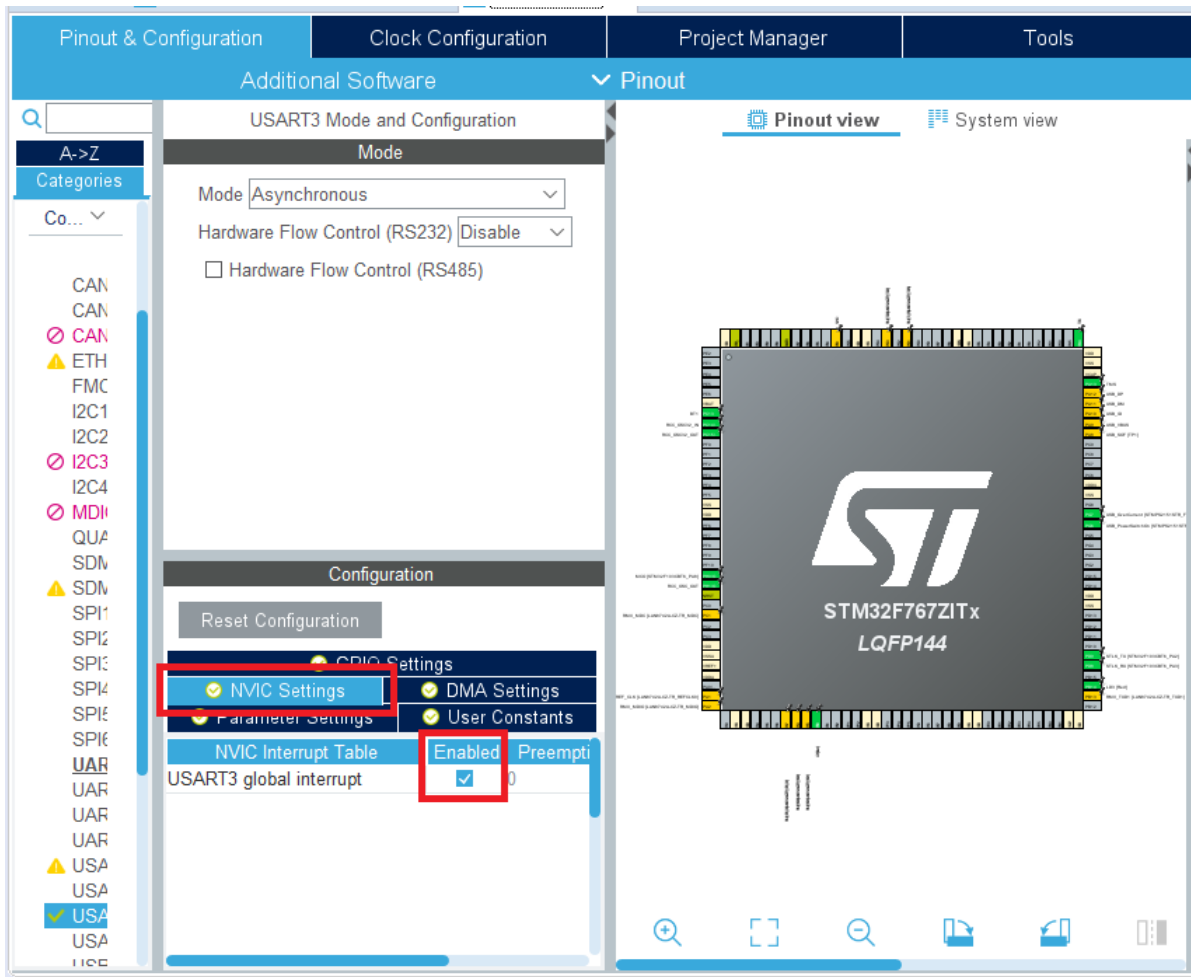
See the image below for how it should look like:



The pins PD8 and PD9 should be linked to ST-Link. To make sure the pins are linked, check the pinout view on the right side of the image.

It is also a must to enable global interrupts for USART3. This can be done by going into the NVIC settings of the USART3. See the image below:





When your IDE User Interface is looking the same as the images, press CTRL+S to save it and click yes to generate code for the initialization. The next step will be to write the code and flash it on the Nucleo.

## Code flashing

Because interrupts are used, we do not have to change the main code. The interrupts must ensure that the code for controlling the LED is not disturbed.

First, the received and transmitted characters must be stored so that they can be used to send it back. For this, 2 variables are created called rx\_buff for the received message and tx\_buff for the message to be sent.

These are added in the USER CODE 0 section:

```
/* USER CODE BEGIN 0 */
uint8_t tx_buff;
uint8_t rx_buff;
/* USER CODE END 0 */
```

The Nucleo will act as a slave since its only job is to echo the received messages. The receive interrupt must, therefore, be initialized first. This is done in the USER CODE 2 section:

```
/* USER CODE BEGIN 2 */
HAL_UART_Receive_IT(&huart3, &rx_buff, 1);
/* USER CODE END 2 */
```

A receive interrupt has been set for USART3. rx\_buff is the variable where the received byte is stored and 1 represents the interrupt calls for each byte received. These are added in the USER CODE 4 section:

```
/* USER CODE BEGIN 4 */
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    tx_buff = rx_buff;

    HAL_UART_Transmit_IT(&huart3, &tx_buff, 1);
}
void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart)
{
    HAL_UART_Receive_IT(&huart3, &rx_buff, 1);
}
/* USER CODE END 4 */
```

A transmit interrupt for USART 3 has been set within the receive callback. This way the moment it's done receiving the byte it will start transmitting it back.

When every piece of code is added, click on Run code and wait till it's finished. This should flash the Nucleo without errors and you can continue with the next step. If not, delete all written code and redo this step. The code will be tested in the next step, with the use of a terminal.

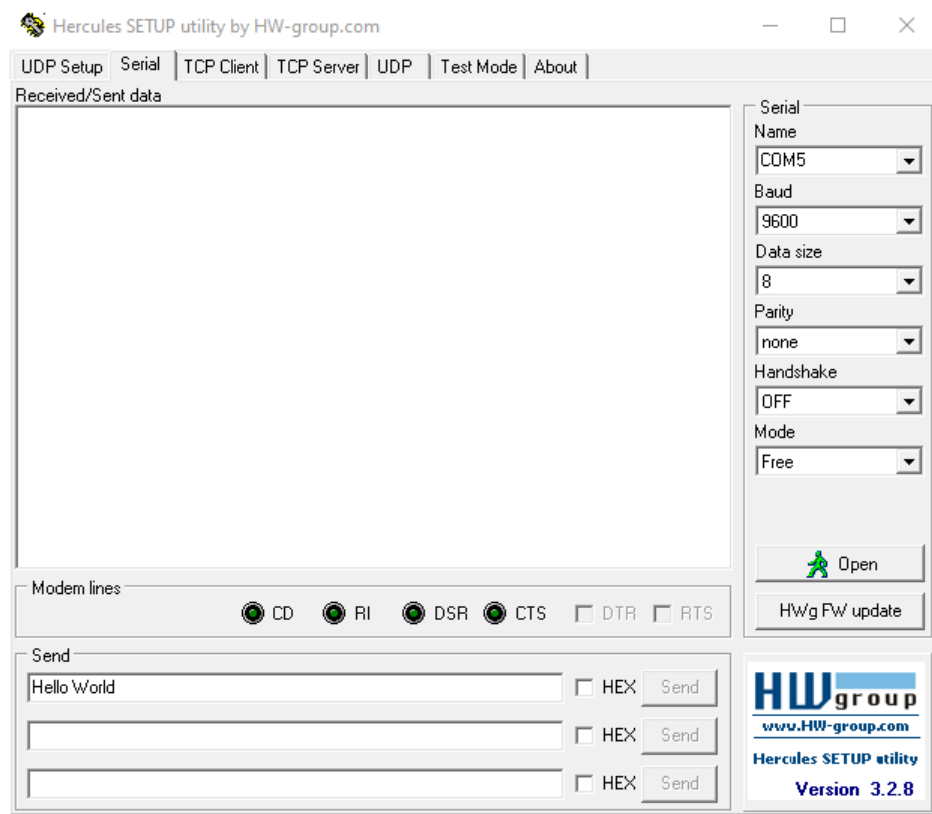
## Terminal

To communicate with the Nucleo, your computer needs software to receive and send serial data through your USB port(COM port). There are multiple good terminals someone could use as this is a matter of personal preference. There will be some recommended terminals down below:

- [Hercules](#)
- [PuTTY](#)
- [DockLight](#)

In this tutorial Hercules is used, but there is no problem with using anything else that is capable of making a serial connection via any COM port. For Hercules, it should look like the image below:

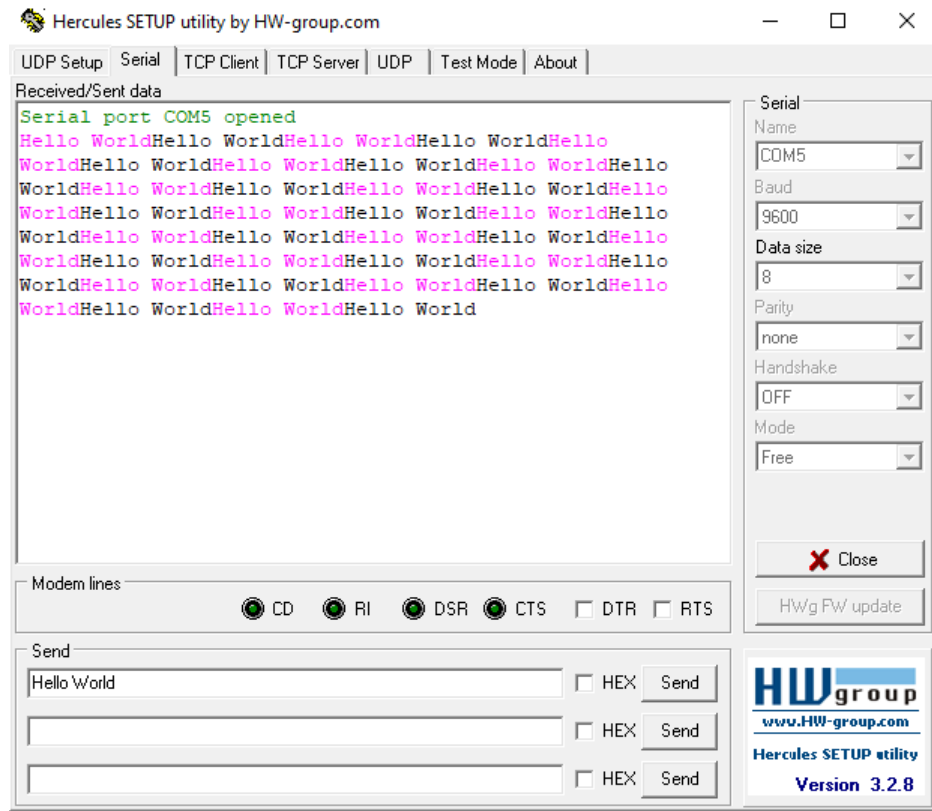




The settings for the terminal should be the same as the USART3 initialization of the Nucleo. Besides the settings, select the correct COM port in this case it was COM5.

## End Result

After the terminal is set and the code is flashed on the Nucleo, send a message via the terminal and the Nucleo should send the same message back.



If it did work, this means your board is ready to make use of serial communication. Otherwise restart the process from *Initialize USART3*.

### Links

- Datasheet of the Nucleo: <https://www.st.com/resource/en/datasheet/stm32f767zi.pdf>
- IDE used for the Nucleo: <https://www.st.com/en/development-tools/stm32cubeide.html>
- Source code for the guide: `src/demonstrators/BareMetalEmbedded/HelloWorld/`

### Nucleo DDS

**authors** Furkan Ali Yurdakul

**date** May 2020

### Description

DDS is a communication that is done in Real-Time. For this reason, the Nucleo is required to function in Real-Time as well.

Besides being Real-Time, the communication is required to use a network communication protocol, for example TCP/IP. For the Nucleo, this is only possible via the Ethernet port. The pin-out needs to be set up for the Ethernet port as well as the TCP/IP stack. For the TCP/IP the build-in LwIP stack will be used, which is made for the embedded application on STM32 microcontrollers. On top off the TCP/IP stack there will be the DDS stack. For this, the Eprosima's FastRTPS library is chosen.

These libraries will then be combined to create the solution for using DDS on embedded bare-metal hardware.

More information about the Nucleo can be found in the [Links](#).

## DDS Setup

This list contains guides to set up different parts of the Nucleo.

### FreeRTOS setup on the Nucleo

**authors** Furkan Ali Yurdakul

**date** May 2020

### Description

Embedded bare-metal hardware needs some adjustments to make proper use of DDS. These kinds of hardware are mainly not suited for software operating in Real-Time. DDS will work when the communication operates in a non-Real-Time software, but will lose its purpose. Because of this, the software would need an OS to make the Nucleo operate in Real-time. In this guide, FreeRTOS will be used to run two tasks simultaneously, which are for toggling LEDs at a different speed.

**For this tutorial the following products have been used:**

- Nucleo
- Lenovo Thinkpad T470 (Windows)
- Micro USB cable
- STM32CubeIDE

### Real-Time OS

There are multiple Operating Systems that are Real-Time OS (RTOS) with a focus on microcontrollers from 8- to 32-bits. Some examples of RTOSes are VxWorks, QNX, Win CE, pSOS, Nucleus® RTOS, NuttX, and FreeRTOS™. RTOSes enable to run multiple tasks and schedule them according to a preemptive scheduling algorithm at the same time with usages of interrupts. [More about RTOSes](#).

These RTOSes all have the same purpose, but the way how the RTOS is making use of the memory and/or interrupts will have an impact on the latency of the RTOS. This guide is not to show how we could improve the speed of the communication using DDS, but rather how to use the protocol itself. In this case it doesn't matter which RTOS is chosen, as long as it's compatible with the Nucleo.

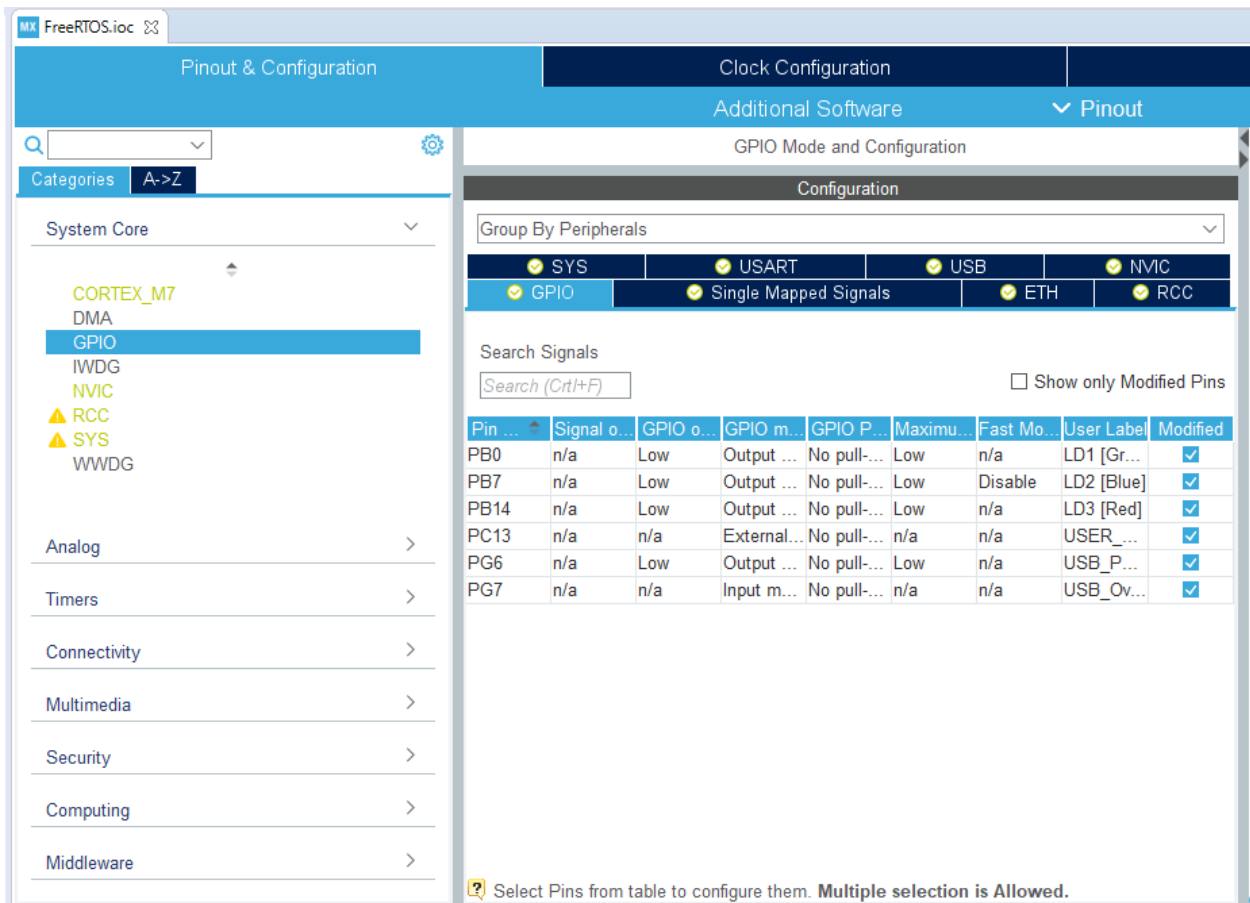
The decision for the best RTOS has been made by taking into consideration if there is enough support/documentation to be found on- or offline, and the accessibility of the RTOS. The datasheet of the Nucleo shows the support and usage of FreeRTOS.

The webpage of FreeRTOS mentions the software being open-source and makes use of an MIT-License. The reasons elaborated above make the usage of FreeRTOS optimal for this project and there is no need to look any further.

## Implementation of FreeRTOS

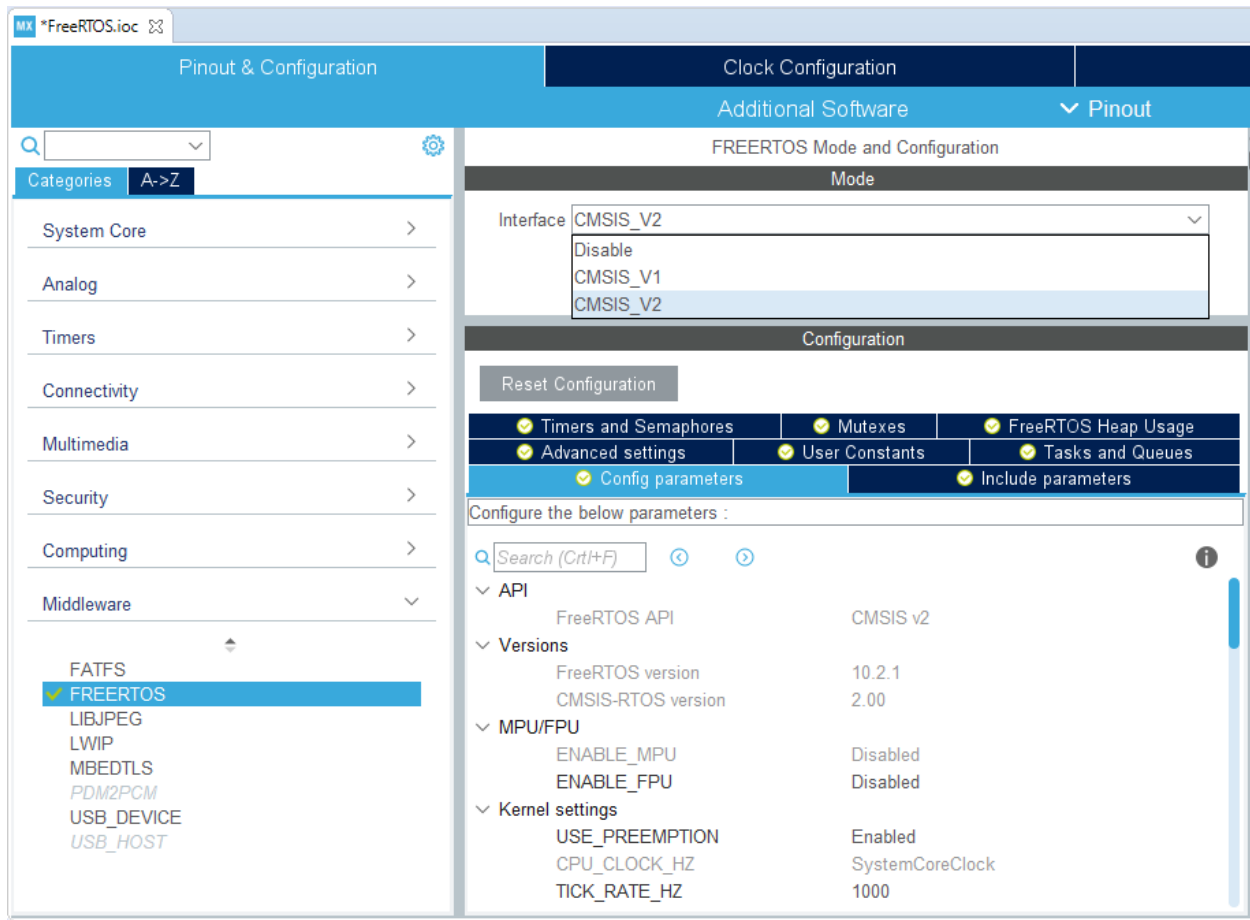
Let's start with making a clean project for this guide. If you have trouble or forgot how to do this, see [“How to setup the Nucleo”](#) how to make a clean project specifically for the Nucleo. You can name this project anything you want since it will not be used for the end product, but only to test FreeRTOS. For this guide the project is called “FreeRTOS”.

2 on-board LEDs will be used to test how to make 2 tasks run simultaneously. For this we need to enable 2 pins as output. Open the “.ioc” file and look for “GPIO” under the section “System Core”. If a clean build was made, this is how it should look like.

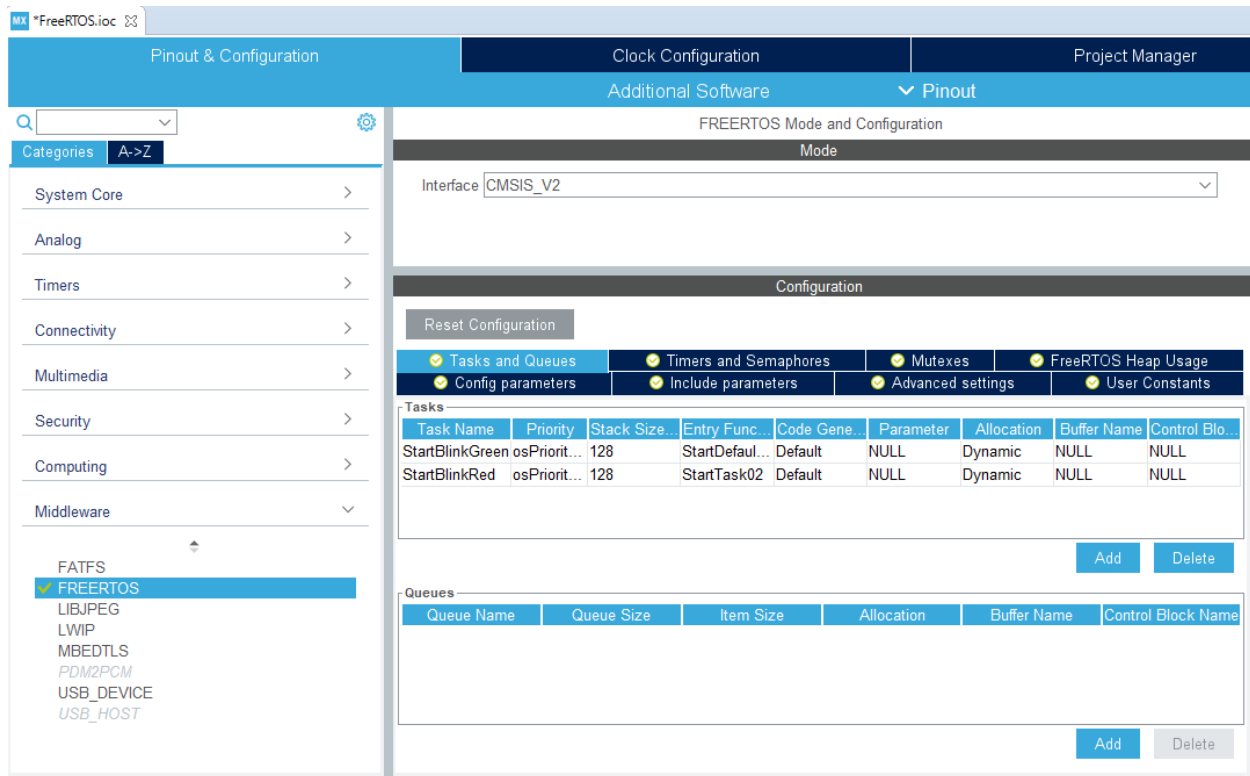


The on-board LEDs are preselected and the labels should be visible as LD1 [Green], LD2 [Blue] and LD3 [Red].

In the “Middleware” section you can find the FreeRTOS interface. Set the interface to CMSIS\_V2(CMSIS\_V1 is just an older API version) to enable the use of FreeRTOS.



This guide will show how to toggle 2 different LEDs at different speeds. For this, we will need 2 tasks which can be created in the tab called “Tasks and Queues”. The tasks made for this guide are called StartBlinkGreen and StartBlinkRed, and entry function names are BlinkGreen and BlinkRed. Judged by the name, BlinkGreen will toggle the green LED on and off and the BlinkRed will do the same for the red LED.



When everything is done as above and matches the pictures, continue with the next part. This part explains and implements the change for the timebase source.

## System Timer

The HAL, an STM32 embedded software layer that ensures maximized portability across the STM32 portfolio, makes use of a unique timebase source. The RTOS has full control over this source and most RTOSes force the system timer(SysTick) priority to be the lowest. For this reason, the timebase source will be changed from SysTick to a less commonly used timer. In the note below, you can find a quote taken from the datasheet of the STM32CubeIDE, which explains why exactly it is better to use a different timebase source instead of the SysTick ([datasheet](#)).

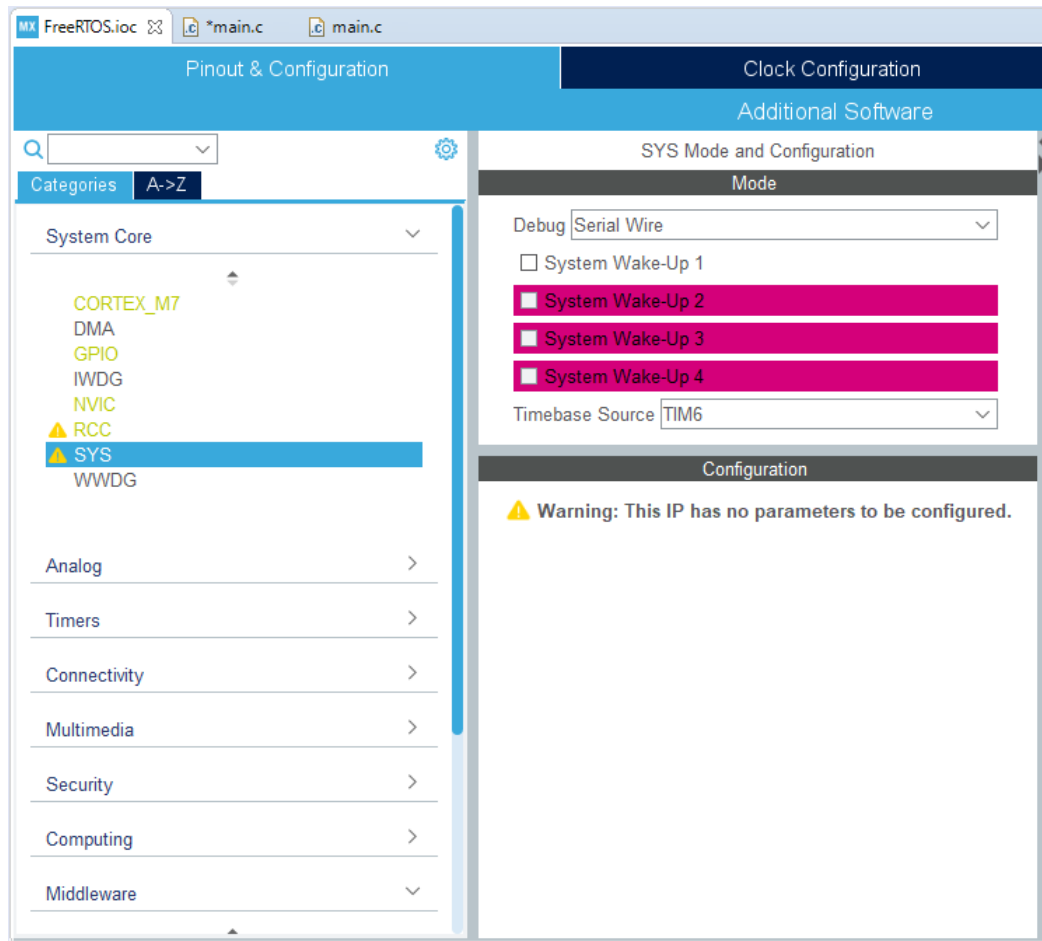
**Note:** “By default, the STM32Cube HAL is built around a unique timebase source, the Arm® Cortex® system timer (SysTick).

However, HAL-timebase related functions are defined as weak so that they can be overloaded to use another hardware timebase source. This is strongly recommended when the application uses an RTOS, since this middleware has full control on the SysTick configuration (tick and priority) and most RTOSs force the SysTick priority to be the lowest.

Using the SysTick remains acceptable if the application respects the HAL programming model, that is, does not perform any call to HAL timebase services within an Interrupt Service Request context (no dead lock issue).”

This project doesn’t make use of any timers yet but taken into consideration for the future purpose of this project a basic timer will be used. Based on the datasheet of the Nucleo, timer 6(TIM6) and timer 7(TIM7) are the basic 16-bit timers. To prevent conflicts in the future the timebase source will be TIM6.

To do this open the category “System Core” and open “SYS”. Here we can change the timebase source to TIM6.



After everything is selected, close the “.ioc” file and accept to generate the code. This would create all the software needed for FreeRTOS and the tasks for it.

## Testing FreeRTOS

From now on all the coding will be done within the created tasks and not in the main while loop. This is because the system is taken over by the FreeRTOS.

```

142  /* Create the thread(s) */
143  /* creation of StartBlinkGreen */
144  StartBlinkGreenHandle = osThreadNew(BlinkGreen, NULL, &StartBlinkGreen_attributes);
145
146  /* creation of StartBlinkRed */
147  StartBlinkRedHandle = osThreadNew(BlinkRed, NULL, &StartBlinkRed_attributes);
148
149  /* USER CODE BEGIN RTOS_THREADS */
150  /* add threads, ... */
151  /* USER CODE END RTOS_THREADS */
152
153  /* Start scheduler */
154  osKernelStart();
155
156  /* We should never get here as control is now taken by the scheduler */
157  /* Infinite loop */
158  /* USER CODE BEGIN WHILE */
159  while (1)
160  {
161      /* USER CODE END WHILE */
162
163      /* USER CODE BEGIN 3 */
164  }
165  /* USER CODE END 3 */
166 }
167

```

To toggle the LEDs, the software will make use of HAL\_GPIO\_TogglePin() command. To show that the system can run multiple tasks at the same time we will use a delay to toggle the LEDs. The command osDelay() will be used to create a delay. Without an RTOS the delays should conflict the frequency of the LEDs toggling on and off. For this test the green LED will toggle every second and the red LED will toggle every half second. The image below shows the implementation for toggling the pins and the used delays.

```

394  /* USER CODE BEGIN Header_BlinkGreen */
395  /**
396   * @brief Function implementing the StartBlinkGreen thread.
397   * @param argument: Not used
398   * @retval None
399   */
400  /* USER CODE END Header_BlinkGreen */
401  void BlinkGreen(void *argument)
402  {
403      /* USER CODE BEGIN 5 */
404      /* Infinite loop */
405      for(;;)
406      {
407          HAL_GPIO_TogglePin(LD1_GPIO_Port, LD1_Pin);
408          osDelay(1000);
409      }
410      /* USER CODE END 5 */
411  }
412
413  /* USER CODE BEGIN Header_BlinkRed */
414  /**
415   * @brief Function implementing the StartBlinkRed thread.
416   * @param argument: Not used
417   * @retval None
418   */
419  /* USER CODE END Header_BlinkRed */
420  void BlinkRed(void *argument)
421  {
422      /* USER CODE BEGIN BlinkRed */
423      /* Infinite loop */
424      for(;;)
425      {
426          HAL_GPIO_TogglePin(LD3_GPIO_Port, LD3_Pin);
427          osDelay(500);
428      }
429      /* USER CODE END BlinkRed */
430  }
431

```

After everything is done, save and compile the “main.c” file. the compilation shouldn’t give any conflicts. If it does delete the written code a rewrite matching the image above. If it doesn’t give any conflicts, flash the software on the Nucleo. The Nucleo should toggle the green LED every second and the red LED twice as fast. This shows that the 2 tasks, which both have delays build in, can run at the same time.

This allows us to have a Real-Time system and are ready to continue with the next step. The next step, “*Libraries*



for *DDS on bare-metal*", explains what kind of libraries are needed and how they are implemented to make use of the communication protocol DDS.

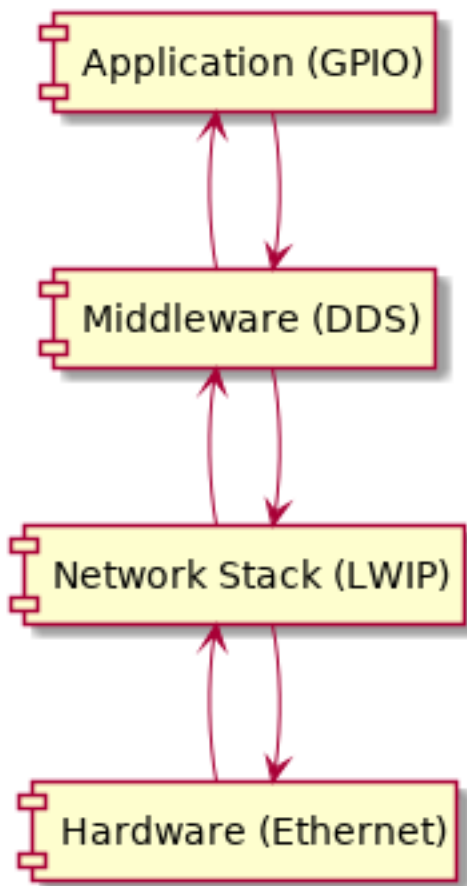
### Libraries for DDS on bare-metal

**authors** Furkan Ali Yurdakul

**date** June 2020

### Description

## Application Layer Diagram



The application layer diagram shows the layers of embedded bare-metal hardware when implementing DDS. On top, there is the Application layer where the main software runs. The main software does the calculations with the input and outputs. The Middleware is where the DDS library belongs. This library will set a proper message whatever it is you are sending or decypher whatever you are receiving. DDS makes use of the TCP/IP protocol to send messages on a low level through the Eternnet port. For this the Light Weight IP (LWIP) is used in the Network Stack layer.

**For this tutorial the following products have been used:**

- Nucleo
- Lenovo Thinkpad T470 (Windows)

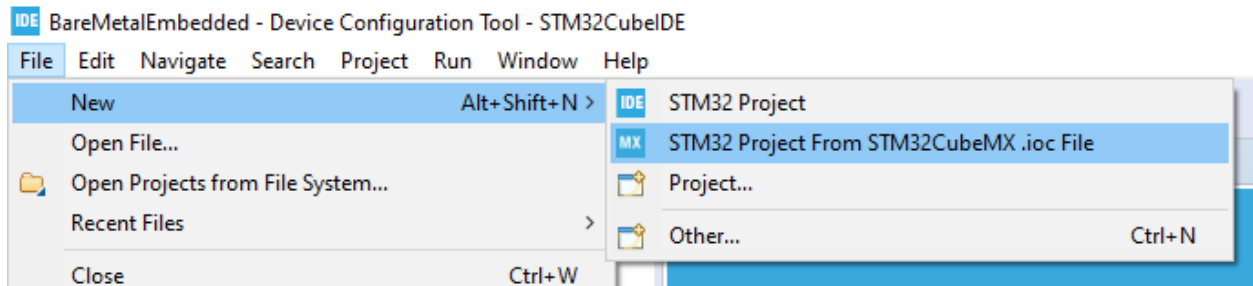
- Micro USB cable
- STM32CubeIDE

### Start project based on “.ioc” file

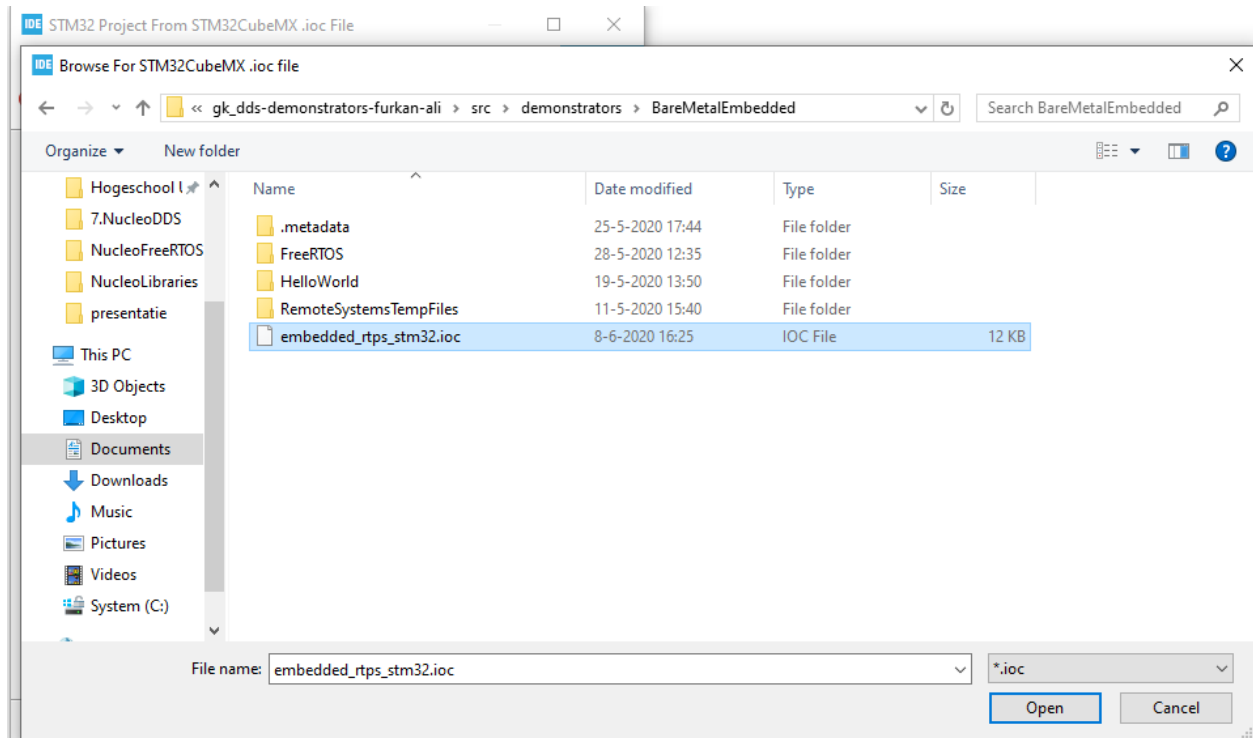
Until now 2 projects have been made on the STM32CubeIDE to test some parts of the Nucleo. Now we will make a final project, however this project is a little bit different from the others. This project will be based on C++, because the main library needed for DDS is based on C++. More about this library can be read later on this page.

Because most of the stuff is already done earlier in this guide, we will skip the parts where we add most of the libraries. This can be done by starting a project from a “.ioc” file. The .ioc file used for this guide can be found in `src/demonstrators/BareMetalEmbedded/embedded_rtps_stm32.ioc`. Download this file to your workspace and continue with the guide.

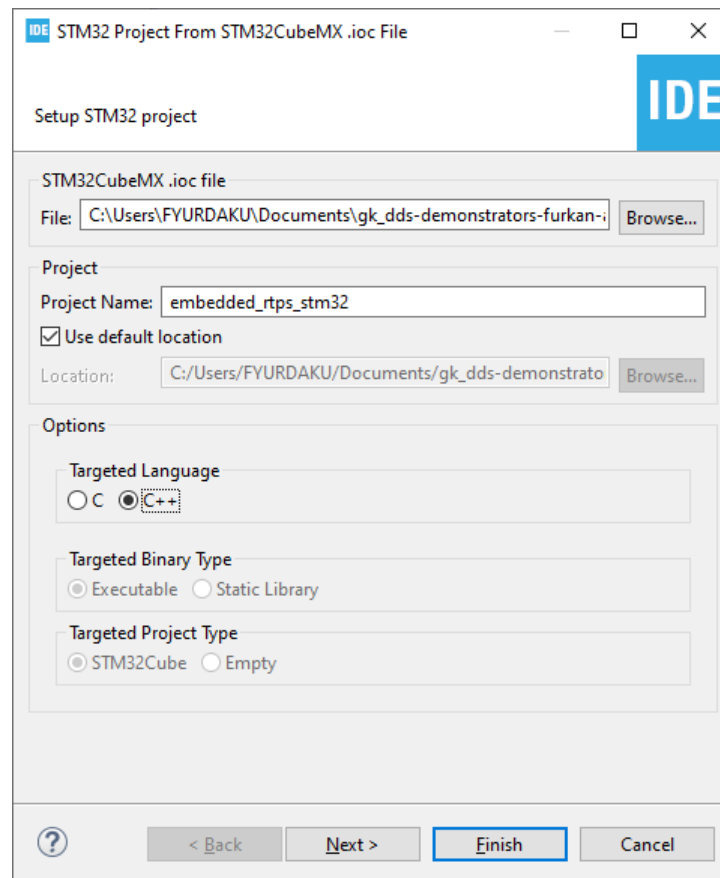
First, you will have to select that the project needs to be based on a “.ioc” file. Click on File followed by New. In this tab select STM32 Project from STM32CubeMX “.ioc” file.



On this pop-up select the downloaded “.ioc” file.



Select C++ as a targeted language and finish building the project.

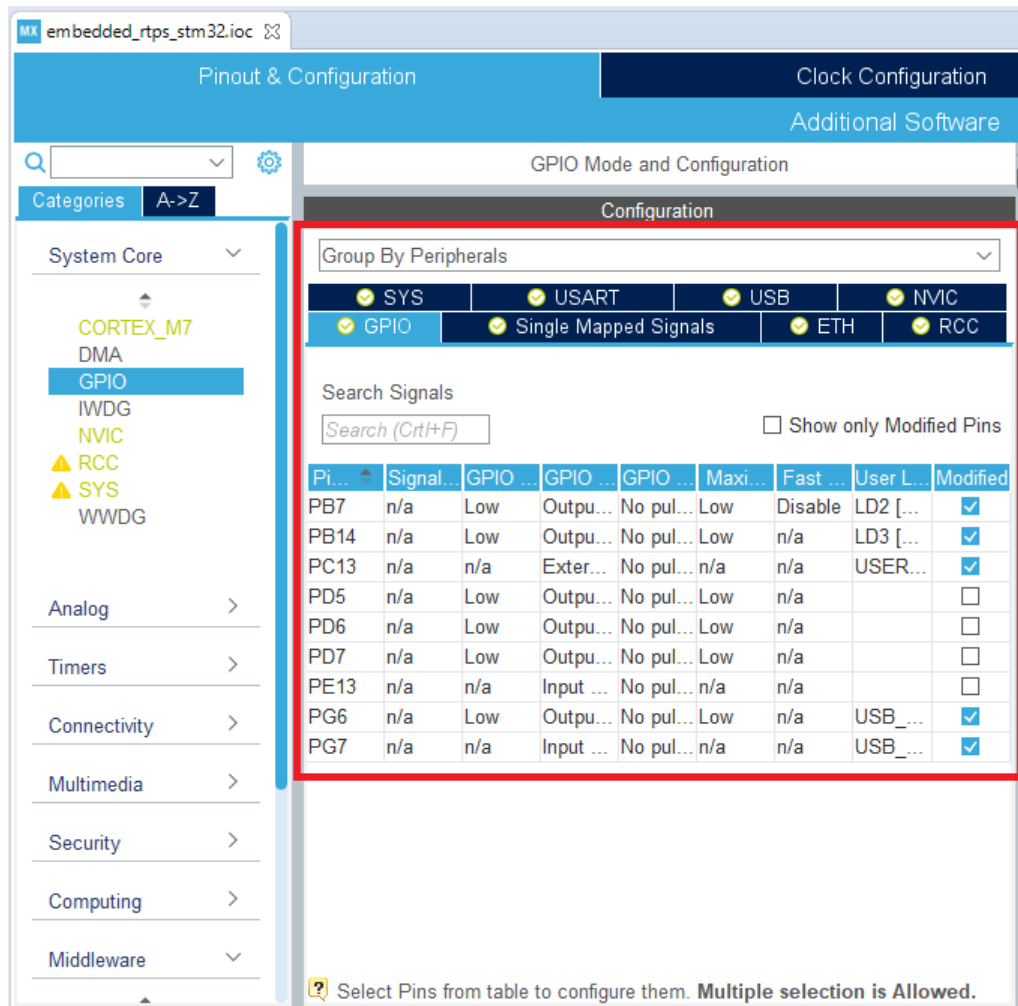


Because you have started a project from another source, the next step will be about doing a check if everything is set as it should be.

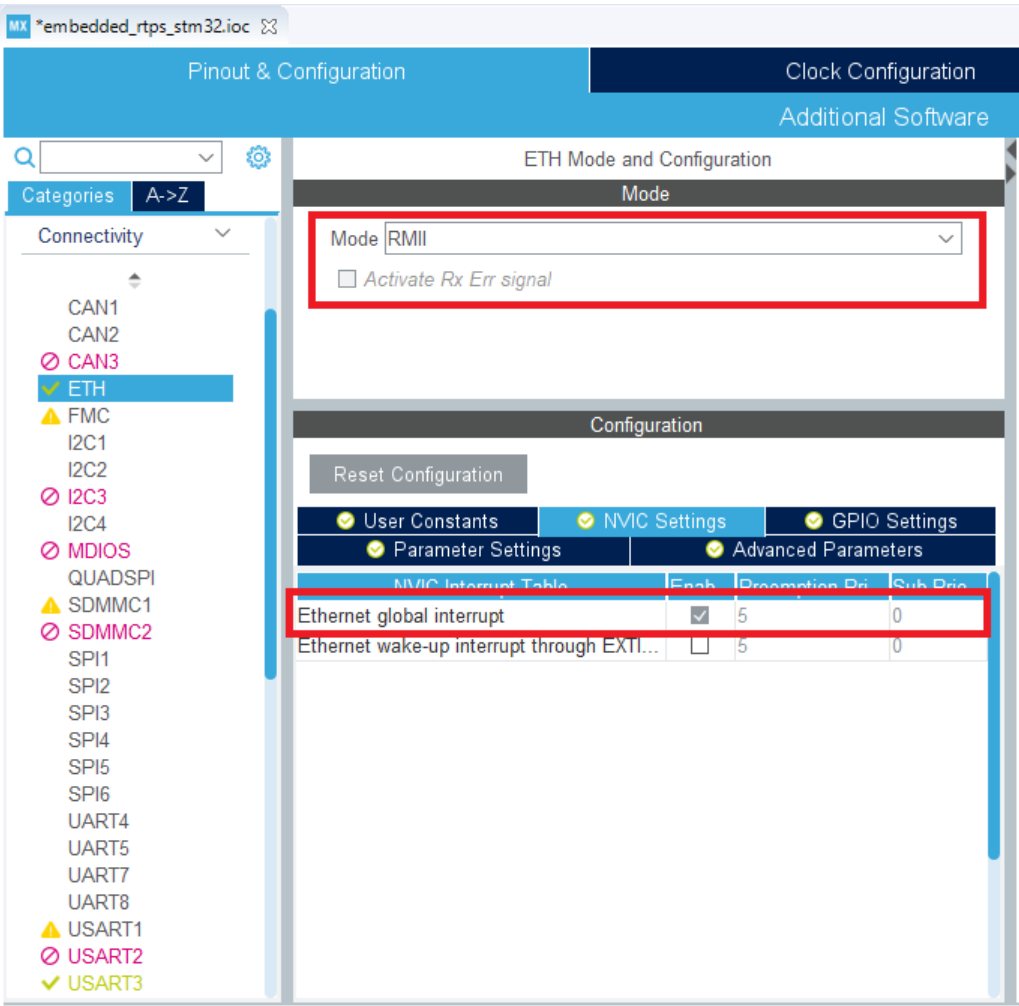
## Check Libraries

If you made a project from the “.ioc” file, you shouldn’t need to do this. But to prevent any mistakes you could check if every library is included

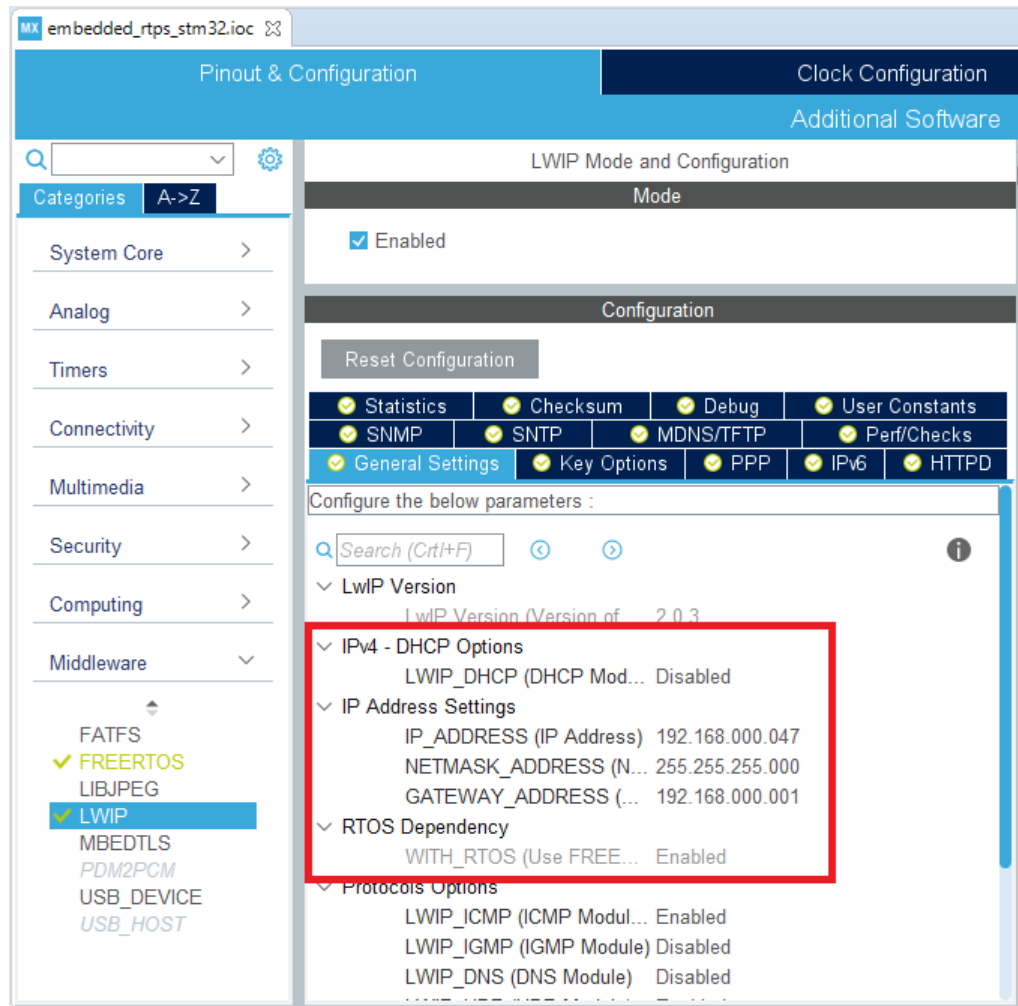
First, check if all the GPIO pins are set as expected. these pins include some LED’s, in- and output for USART and Ethernet pins.



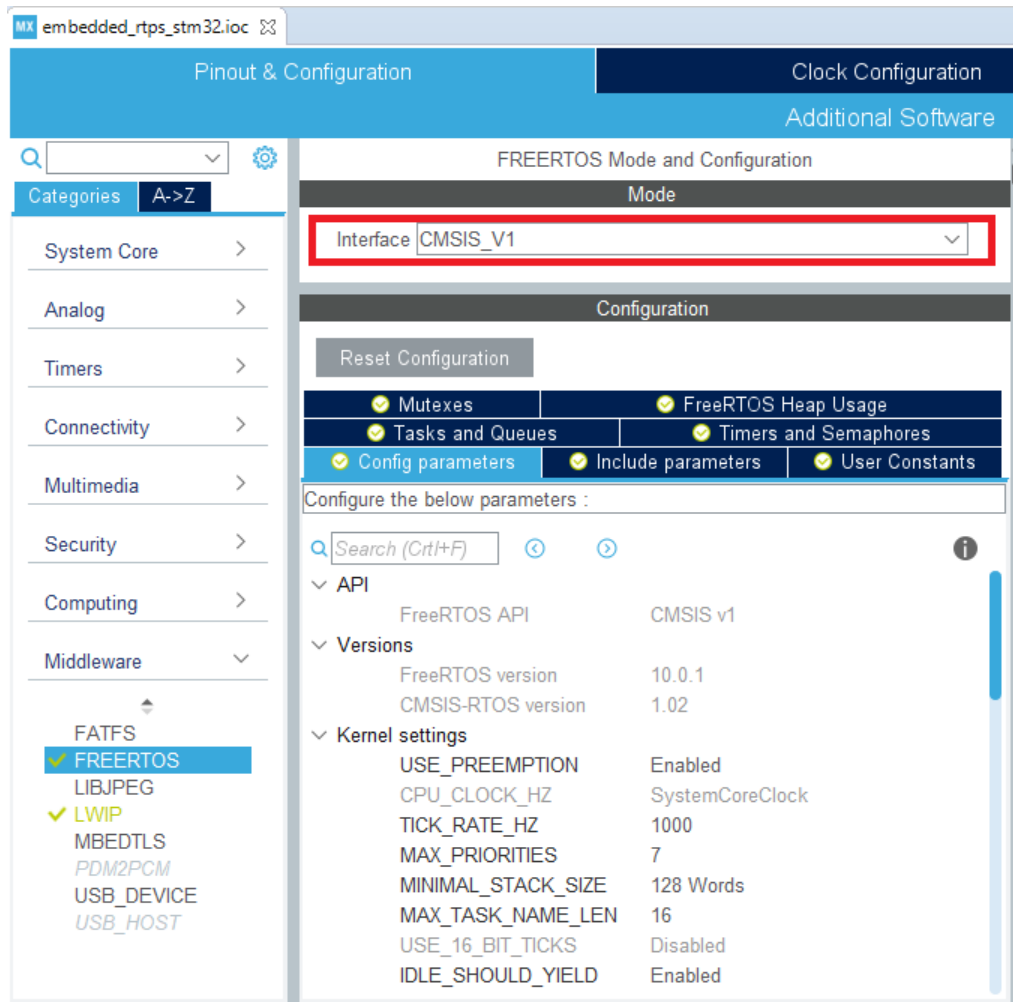
For the Ethernet port, it is also important to have the global interrupt enabled. This interrupt is used for the TCP/IP communication via the Ethernet port.



The LWIP is the library used for the TCP/IP communication. For this guide, the DHCP option is disabled and the IP settings are hardcoded. This will make the guide easier and focus more on DDS instead of the TCP/IP communication.



Last but not least you will need to check if FreeRTOS is enabled. The only importance is that FreeRTOS is enabled and there is no need for more tasks. The default task is enough to show the working of DDS.



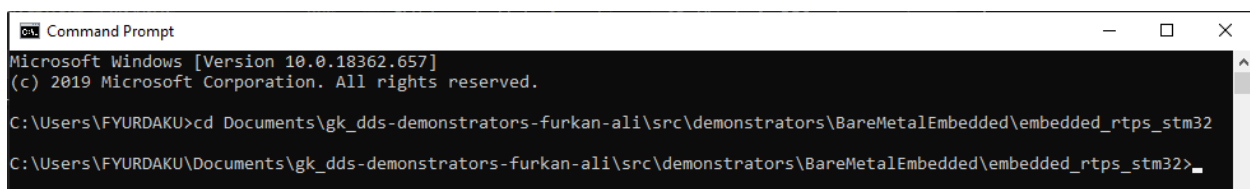
If your project is matching the settings in the images above, you can continue with the next step.

## Ported DDS

Several university students have ported the DDS library on to multiple devices, including the Nucleo. This ported library will be used for this project. To obtain the library, you will need to make use of the program Command Prompt.

The library can be found online on [GitHub](#). Because it is available on GitHub we can make use of the “git clone” command in Command Prompt to get a copy of the library on your computer. The library will then be placed in the folder of our project.

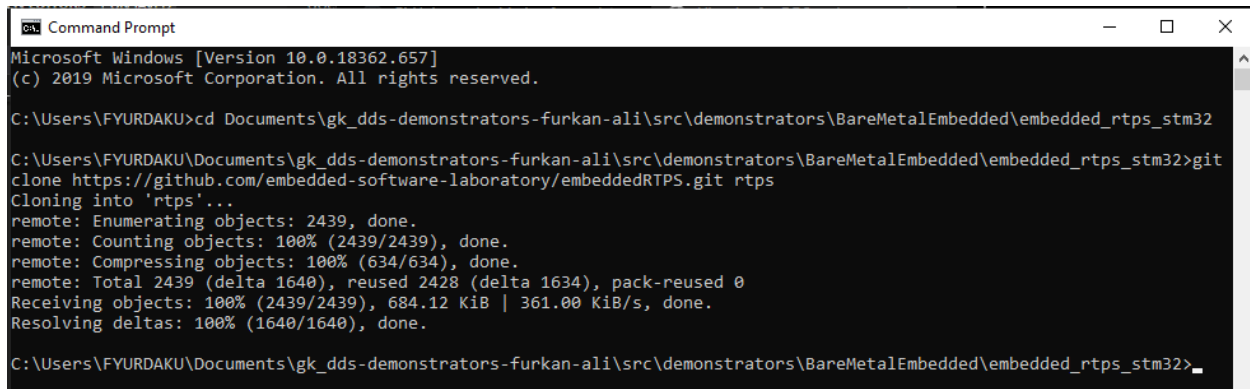
The first step is to make sure your directory is set to the folder of the project DDS. This is needed to provide easier access to the library within the software.



When you have successfully changed your directory, enter the following command to get the ported DDS library by making use of git.:

```
git clone https://github.com/embedded-software-laboratory/embeddedRTPS.git rtps
```

This is how Command Prompt should look like when finished:



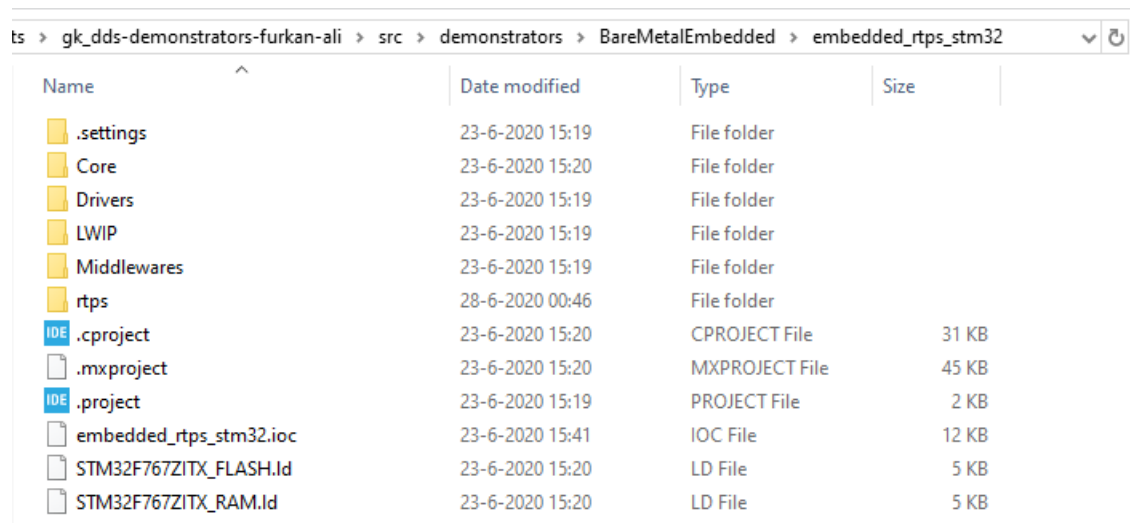
```
Command Prompt
Microsoft Windows [Version 10.0.18362.657]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\FYURDAKU>cd Documents\gk_dds-demonstrators-furkan-ali\src\demonstrators\BareMetalEmbedded\embedded_rtps_stm32

C:\Users\FYURDAKU\Documents\gk_dds-demonstrators-furkan-ali\src\demonstrators\BareMetalEmbedded\embedded_rtps_stm32>git
clone https://github.com/embedded-software-laboratory/embeddedRTPS.git rtps
Cloning into 'rtps'...
remote: Enumerating objects: 2439, done.
remote: Counting objects: 100% (2439/2439), done.
remote: Compressing objects: 100% (634/634), done.
remote: Total 2439 (delta 1640), reused 2428 (delta 1634), pack-reused 0
Receiving objects: 100% (2439/2439), 684.12 KiB | 361.00 KiB/s, done.
Resolving deltas: 100% (1640/1640), done.

C:\Users\FYURDAKU\Documents\gk_dds-demonstrators-furkan-ali\src\demonstrators\BareMetalEmbedded\embedded_rtps_stm32>
```

This is how your folder in your project should look like when finished:

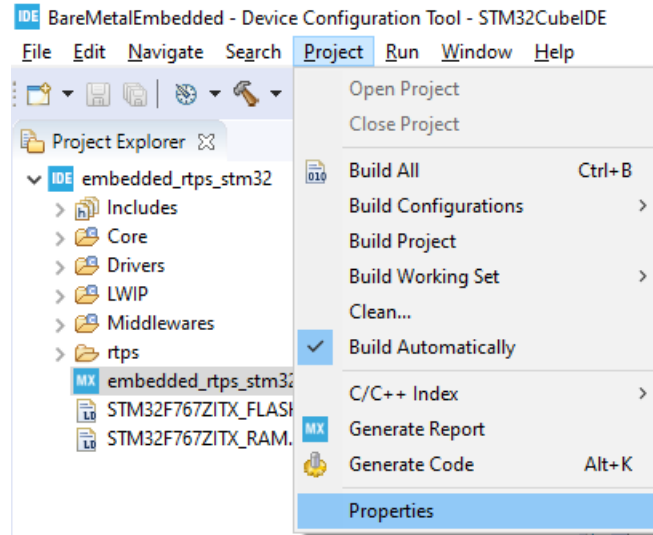


ts > gk_dds-demonstrators-furkan-ali > src > demonstrators > BareMetalEmbedded > embedded_rtps_stm32				
Name	Date modified	Type	Size	
.settings	23-6-2020 15:19	File folder		
Core	23-6-2020 15:20	File folder		
Drivers	23-6-2020 15:19	File folder		
LWIP	23-6-2020 15:19	File folder		
Middlewares	23-6-2020 15:19	File folder		
rtps	28-6-2020 00:46	File folder		
.cproject	23-6-2020 15:20	CPROJECT File	31 KB	
.mxproject	23-6-2020 15:20	MXPROJECT File	45 KB	
.project	23-6-2020 15:19	PROJECT File	2 KB	
embedded_rtps_stm32.ioc	23-6-2020 15:41	IOC File	12 KB	
STM32F767ZITX_FLASH.ld	23-6-2020 15:20	LD File	5 KB	
STM32F767ZITX_RAM.ld	23-6-2020 15:20	LD File	5 KB	

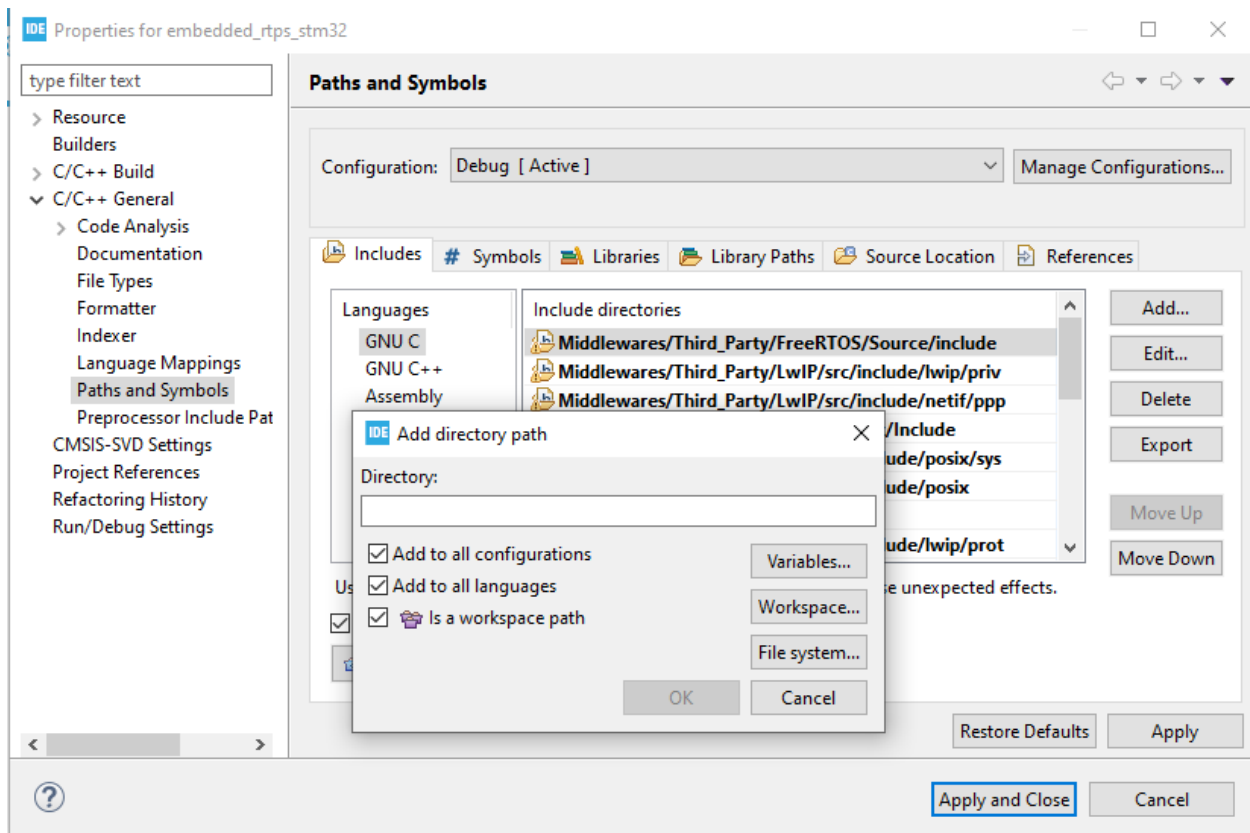
If you don't want/have git on your computer, click on the link to the Github page and make sure to download the DDS library to the right directory in your workspace.

This folder will now be in your workspace, but the IDE won't recognize it as a library. For the IDE to recognize this library, you would need to open the properties of the project.

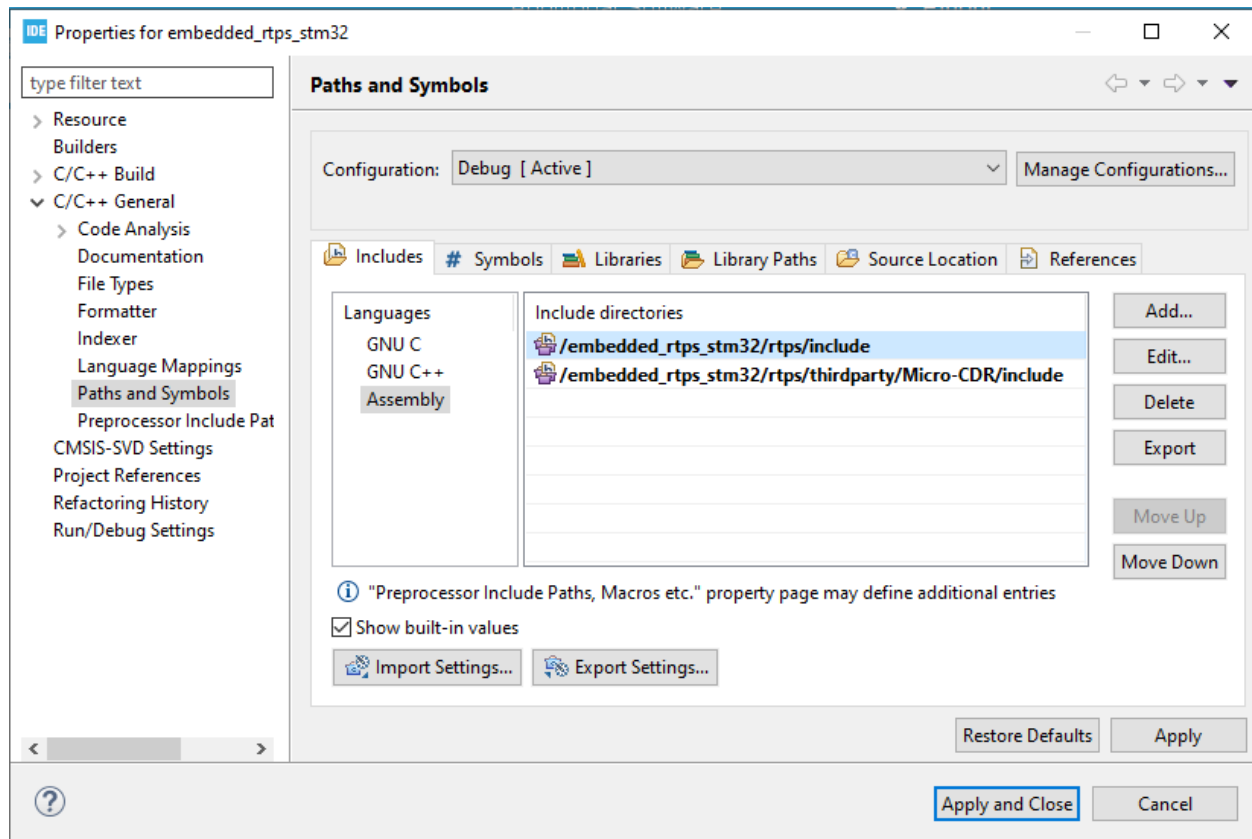




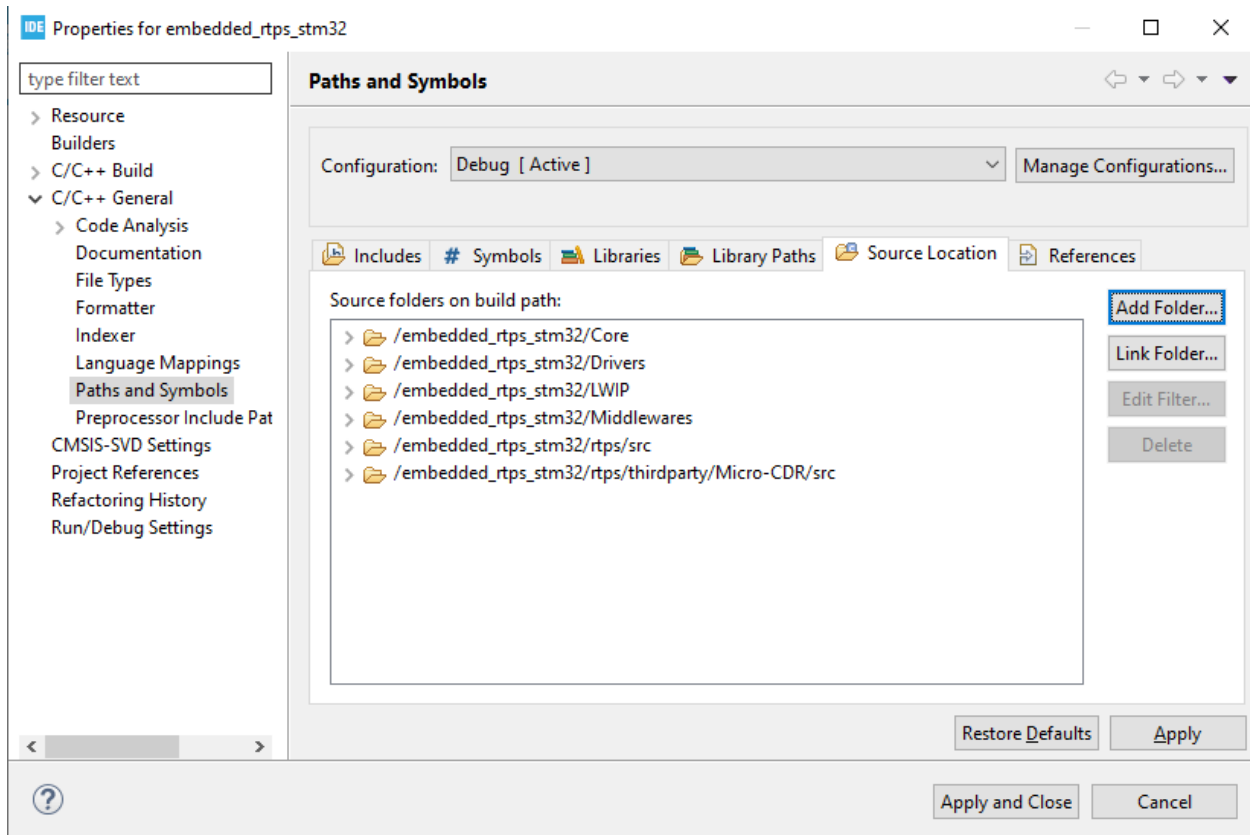
In the “Paths and Symbols” category you will need to add the include folders and source folders of the library. To add this include to all the compilers, select all the options. This will add them to all compilers instantly.



After including all the “include” folders of the library, it should look like the following image.



In the Source location sub-category, make sure to add the “src” folders from the library. This should show the following settings:



The last thing that needs to be changed is a filename of a specific configuration file. As you could read from the GitHub page linked earlier. This library is ported to several devices, thus including multiple config files. To make sure you make use of the correct one, you will need to change the “config\_stm.h” to “config.h”. This “config\_stm.h” can be found in `embedded_rtps_stm32\rtps\include\rtps`.

monstrators > BareMetalEmbedded > embedded_rtps_stm32 > rtps > include > rtps				
Name	Date modified	Type	Size	
common	28-6-2020 00:46	File folder		
communication	28-6-2020 00:46	File folder		
discovery	28-6-2020 00:46	File folder		
entities	28-6-2020 00:46	File folder		
messages	28-6-2020 00:46	File folder		
storages	28-6-2020 00:46	File folder		
utils	28-6-2020 00:46	File folder		
config.h	28-6-2020 00:46	H File	4 KB	
config_aurix.h	28-6-2020 00:46	H File	4 KB	
config_desktop.h	28-6-2020 00:46	H File	4 KB	
rtps.h	28-6-2020 00:46	H File	2 KB	
ThreadPool.h	28-6-2020 00:46	H File	3 KB	

At this point, you have added all the needed libraries and set all the correct settings to make use of the ported DDS library. In “*DDS Coding*”, the focus will be on the code of using the library and see the results of the code.

### DDS Coding

**authors** Furkan Ali Yurdakul

**date** June 2020

### Description

There are a few changes before you can execute this test code. The first change is the laptop to a personal computer. The laptop of Sogeti can cause issues with their firewall and other securities for external communication. Secondly, you will be using an [Ubuntu Windows Subsystem for Linux\(WSL\)](#), because the project to test the DDS communication will be executed on the WSL.

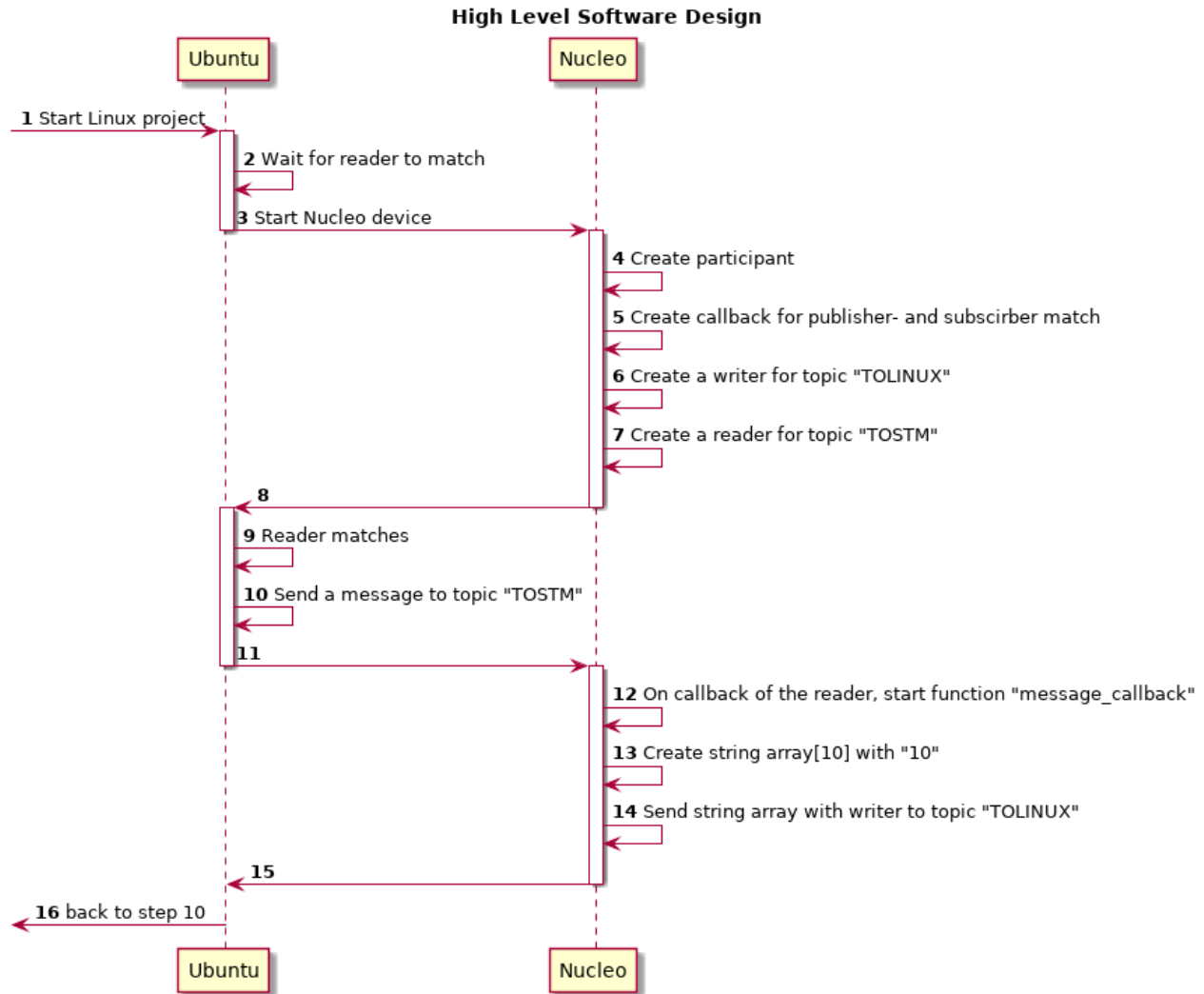
This test code is only to show the working of DDS making use of the libraries shown in “*Libraries for DDS on bare-metal*”. For the extras such as USART communication or making use of the on-board button/LED will not be added.

**For this tutorial the following products have been used:**

- Nucleo
- A PC running on default Windows 10 (without extra security measures)
- Ubuntu(WSL)
- Micro USB cable
- Ethernet cable CAT.5e (2x)
- STM32CubeIDE

### Software design

Before you start using the ported library, study the High-Level Design below to see what is expected. The reason for this design is because of the WSL project that sends and reads from a specific topic. The software for the Nucleo will be built for this project.



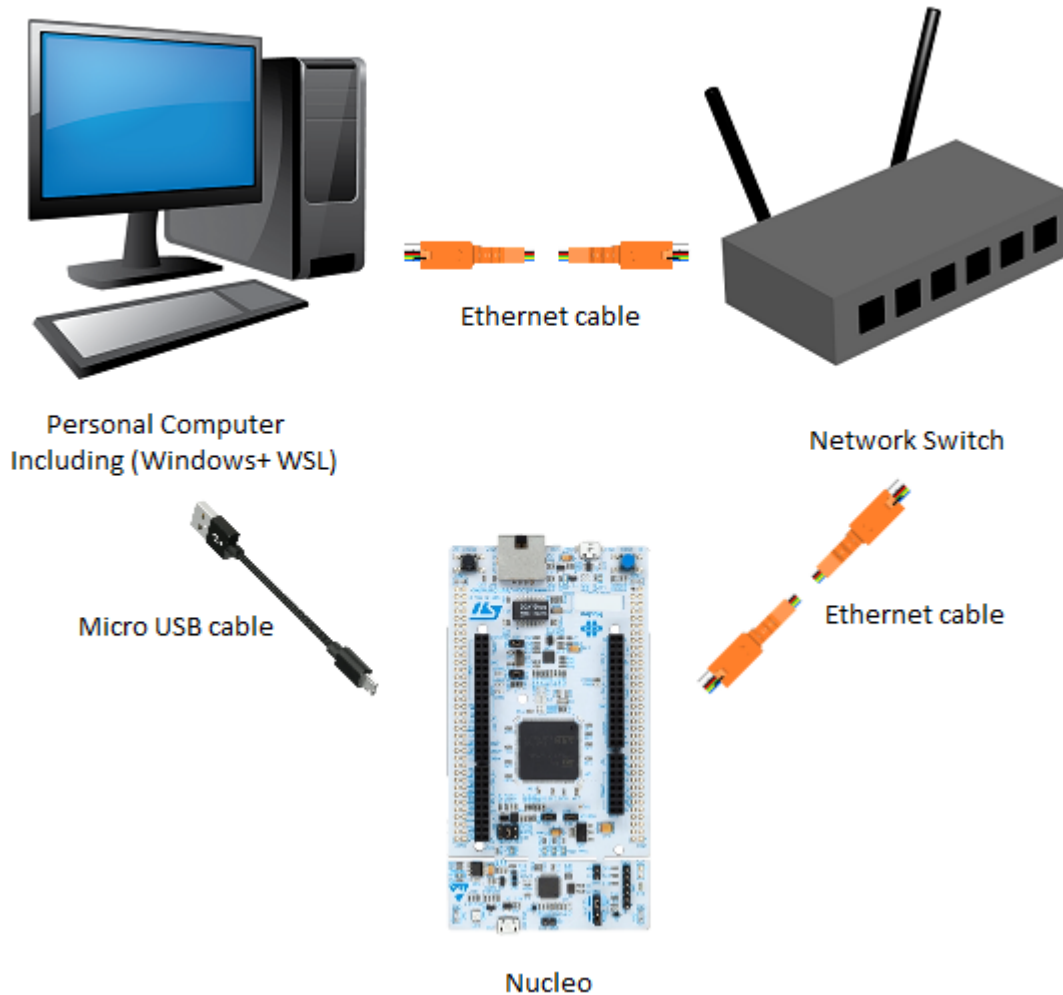
How to get and install the software on your WSL is explained in [WSL project](#). First, start the project on WSL until it waits for a reader match. As you know the basics of DDS, a participant is created within the domain. For this participant, 2 callbacks are created to be able to notice a match on the writer and the reader.

You will need a writer to send messages to the topic “TOLINUX” and a reader to receive a message from the topic “TOSTM”. Upon making a reader for this specific topic, it should match with WSL which responds with a message. This message is recognized because of a callback that is created after creating the reader.

Function “message\_callback” is called and makes the Nucleo send 10 messages containing “10” to the topic “TOLINUX”. After this, there is nothing written, which makes the test repeat itself until it crashes(see [Known issues](#)). The software for Nucleo will be explained a bit more thoroughly in the [Main Code](#).

## Hardware design

For this test the hardware is set as the following:



You can connect the Nucleo directly to the computer. To make it future proof the setup has a network switch in between the Nucleo and the PC. This way, you could add more Nucleo boards to see how it would perform in different circumstances. The micro USB cable is only needed to flash software on the Nucleo. For this guide, it is also used for debugging the software.

## Main Code

The “main.cpp” file can be found in “/src/demonstrators/BareMetalEmbedded/embedded\_rtps\_stm32/Core/Src/main.cpp”. The main part that needs focus is the *Software design*, stated before. “StartDefaultTask” is the default task of FreeRTOS, where you call the function “startRTPStest();”.

```
void StartDefaultTask(void const * argument)
{
    /* init code for LWIP */
    MX_LWIP_Init();

    /* USER CODE BEGIN 5 */
    startRTPStest();
}
```

(continues on next page)

(continued from previous page)

```

volatile auto size = uxTaskGetStackHighWaterMark(nullptr);

while(1);
/* USER CODE END 5 */
}

```

As seen in the *Software design*, this is where the participant, callbacks, writer, and reader are created. The initialization of DDS happens in the function “startRTPSTest();”.

```

//Function to start the RTPS Test
void startRTPSTest() {

    //Initialize variables and complete RTPS initialization
    bool subMatched = false;
    bool pubMatched = false;
    bool received_message = false;

    static rtps::Domain domain;

    //Create RTPS participant
    rtps::Participant* part = domain.createParticipant();
    if(part == nullptr){
        return;
    }

    //Register callback to ensure that a publisher is matched to the writer before_
    ↪ sending messages
    part->registerOnNewPublisherMatchedCallback(setTrue, &pubMatched);
    part->registerOnNewSubscriberMatchedCallback(setTrue, &subMatched);

    //Create new writer to send messages
    rtps::Writer* writer = domain.createWriter(*part, "TOLINUX", "TEST", false);
    rtps::Reader* reader = domain.createReader(*part, "TOSTM", "TEST", false);
    reader->registerCallback(&message_callback, writer);

    domain.completeInit();

    //Check that writer creation was successful
    if(writer == nullptr || reader == nullptr){
        return;
    }

    //Wait for the subscriber on the Linux side to match
    while(!subMatched || !pubMatched){

    }

    while(true){}
}

```

The function “message\_callback” starts upon receiving something from WSL. This function creates a string array length of 10. This string array contains 10 times the string “10”, which is then sent as a message through DDS to WSL.

```

void message_callback(void* callee, const rtps::ReaderCacheChange& cacheChange){
    rtps::Writer* writer = (rtps::Writer*) callee;

```

(continues on next page)

(continued from previous page)

```
static std::array<uint8_t, 10> data{};
data.fill(10);
auto* change = writer->newChange(rtps::ChangeKind_t::ALIVE, data.data(), data.
↪size());
}
```

These are the main parts of this project. The whole project can be found in `src/demonstrators/BareMetalEmbedded/embedded_rtps_stm32/`. In the next part, the setup of the project for the WSL will be explained.

## WSL project

If you have a Ubuntu(WSL) set and ready you can go ahead and install the following dependencies:

---

**Note:** Visit <https://ubuntu.com/wsl> if you don't know how to install Ubuntu on Windows Subsystem for Linux.

---

```
sudo apt-get -y update && apt-get -y upgrade
sudo apt-get install -y \
    software-properties-common \
    libboost-all-dev \
    libssl-dev \
    build-essential \
    cmake \
```

After installing the dependencies, the project can be cloned using the following command:

```
git clone https://github.com/embedded-software-laboratory/embeddedRTPS-STM32
```

To compile the project navigate to the “linux” folder, then use the following commands:

```
cd embeddedRTPS-STM32 \
cd linux \
mkdir build \
cd build \
cmake -DTHIRDPARTY=ON .. \
make
```

Having everything into place, you should be able to make a connection between the Nucleo and your computer. The next part will show you the results and how to get them.

## Results

As seen in the *Software design*, the DDS project will first be started on WSL. This way it will wait for a reader to match with the Nucleo.

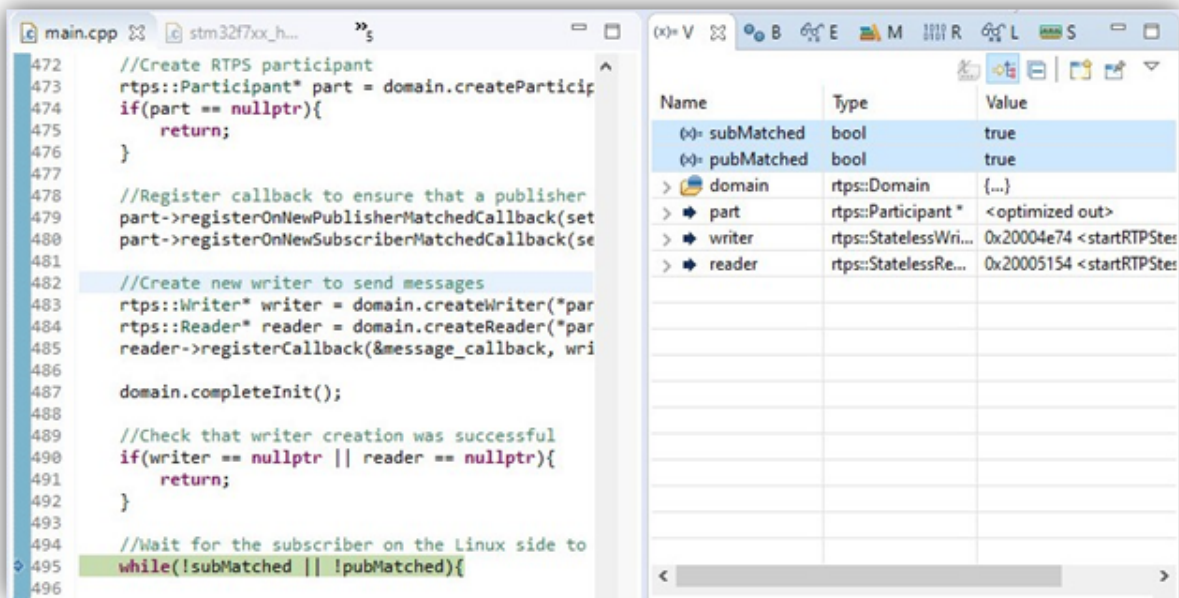


```

fay@DESKTOP-JG2CNT1:~/embeddedRTPS-STM32/linux/build$ ./embedded_rtps_test
Creating RTPS Participant...
Creating Participant...
Created Participant succesfully.
Creating RTPS reader on topic:TOLINUX
Creating FastRTPS Writer on topic TOSTM
Successfully created writer.
Waiting for reader match with STM32...

```

After starting DDS on the WSL, the Nucleo can be reset in order to establish a connection and sending messages back and forth. In this guide, the Nucleo is started in debug mode to show the variables changing to true when they connect successfully.



On your WSL it should look like the image below where you can see the messages “10” being received.

```
Fay@DESKTOP-JG2CNT1:~/embeddedRTPS-STM32/linux/build$ ./embedded_rtps_test
Creating RTPS Participant...
Creating Participant...
Created Participant succesfully.
Creating RTPS reader on topic:TOLINUX
Creating FastRTPS Writer on topic TOSTM
Successfully created writer.
Waiting for reader match with STM32...
Reader matched to remote Writer.
Remote Endpoint GUID:1.2.3.4.5.6.7.8.9.a.c.0|0.0.1.3
Local Endpoint GUID:1.f.da.be.76.0.0.0.1.0.0.0|0.0.1.4
Got Reader match - starting tests...
Conducting new Test...
Send message to the STM32.
Received message from STM32 with len:10
0 : 10
1 : 10
Conducting new Test...
2 : 10
3 : 10
Send message to the STM32.
4 : 10
5 : 10
6 : 10
7 : 10
8 : 10
9 : 10

Received message from STM32 with len:10
0 : 10
Conducting new Test...1
: 10
2 : 10
```

Congratulations! You have successfully implemented the communication protocol DDS on an embedded bare-metal hardware device.

The next part is needed if you haven't encountered the same result as the guide has.

### Known issues

There are a few known issues discovered when testing the software. In the picture below we see the first 2 of them:

```

Received message from STM32 with len:10
0 : 10
1 : 10
2 : 10
3 : 10
4 : 10
5 : 10
6 : 10
7 : 10
8 : 10
9 : 10

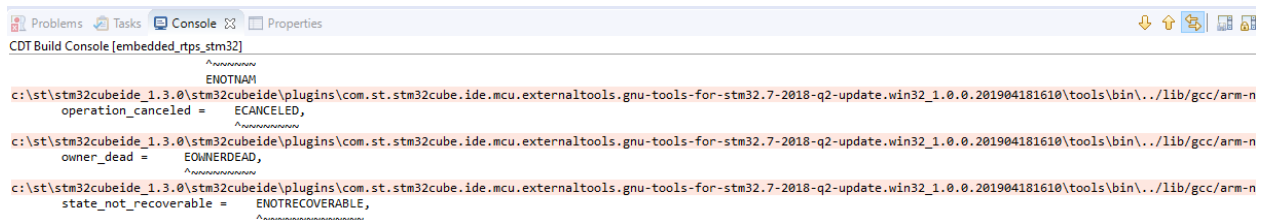
Segmentation fault (core dumped)
fay@DESKTOP-JG2CNT1:~/embeddedRTPS-STM32/linux/build$ ./embedded_rtps_test
Creating RTPS Participant...
Creating Participant...
Created Participant succesfully.
Creating RTPS reader on topic:TOLINUX
Creating FastRTPS Writer on topic TOSTM
Successfully created writer.
Waiting for reader match with STM32...
Reader matched to remote Writer.
Remote Endpoint GUID:1.2.3.4.5.6.7.8.9.a.c.0|0.0.1.3
Local Endpoint GUID:1.f.da.be.7c.0.0.0.1.0.0.0|0.0.1.4
Got Reader match - starting tests...
Conducting new Test...
Send message to the STM32.
Reader matched to remote Writer.
Remote Endpoint GUID:1.2.3.4.5.6.7.8.9.a.c.0|0.0.1.3
Local Endpoint GUID:1.f.da.be.7c.0.0.0.1.0.0.0|0.0.1.4

```

A segmentation fault, this is possible because of the software running the code infinitely and crashes at some point.

Connection issue, this could be because of the firewall your computer. It could also be possible because of the way the hardware is connected.

These 2 issues are not a problem for showing the working of DDS. But the next issue can be a big deal because it gives an error when compiling the code. The error says it's a scoping issue, but it looks more like a compiler issue.



```

CDT Build Console [embedded_rtps_stm32]
~::~::~:
ENOTNAM
c:\st\stm32cubeide_1.3.0\stm32cubeide\plugins\com.st.stm32cube.ide.mcu.externaltools.gnu-tools-for-stm32.7-2018-q2-update.win32_1.0.0.201904181610\tools\bin\..\lib/gcc/arm-n
operation_canceled = ECANCELED,
c:\st\stm32cubeide_1.3.0\stm32cubeide\plugins\com.st.stm32cube.ide.mcu.externaltools.gnu-tools-for-stm32.7-2018-q2-update.win32_1.0.0.201904181610\tools\bin\..\lib/gcc/arm-n
owner_dead = EOWNERDEAD,
c:\st\stm32cubeide_1.3.0\stm32cubeide\plugins\com.st.stm32cube.ide.mcu.externaltools.gnu-tools-for-stm32.7-2018-q2-update.win32_1.0.0.201904181610\tools\bin\..\lib/gcc/arm-n
state_not_recoverable = ENOTRECOVERABLE,
~::~::~:

```

There has not been proper research done on these issues. These issues could be addressed in future projects. Visit the [GitHub page](#), which still gets updates, for the project which this guide is based on.

#### Todo:

- Research the cause of the connection issue

- Research the cause of the incompatibility with the ported library.
  - A project to make multiple bare-metal devices communicate with each other for a round-trip
- 

### Links

- Datasheet of the Nucleo: <https://www.st.com/resource/en/datasheet/stm32f767zi.pdf>
- IDE used for the Nucleo: <https://www.st.com/en/development-tools/stm32cubeide.html>
- Manual of the IDE: [https://www.st.com/resource/en/user\\_manual/dm00104712-stm32cubemx-for-stm32-configuration-and-initialization.pdf](https://www.st.com/resource/en/user_manual/dm00104712-stm32cubemx-for-stm32-configuration-and-initialization.pdf)
- Comparison standard and Real-Time operating system: <https://www.microcontrollertips.com/real-time-standard-how-to-choose-rtos/>
- Webpage of FreeRTOS: <https://www.freertos.org/>
- Ported DDS: <https://github.com/embedded-software-laboratory/embeddedRTPS/tree/dc5e5106b136e91fad54dc969f3a4d68ea101566>
- Source code for FreeRTOS: `src/demonstrators/BareMetalEmbedded/FreeRTOS/`
- Source code for DDS: `src/demonstrators/BareMetalEmbedded/embedded_rtps_stm32/`

### Clean Install with internet

**authors** Joost Baars

**date** Mar 2020

This page describes how the general DDS necessities can be installed (Cyclone DDS library and the requirements for the documentation). This tutorial was tested from a clean Raspbian Buster Lite installation (version 13-03-2020).

This tutorial is specifically for Linux systems based on Debian. Therefore, this tutorial also works for Ubuntu systems.

This tutorial also specifies what commands need root access. These commands use `sudo` in front of the actual command that is executed.

Start with the following commands to make sure the repositories are up to date:

```
sudo apt-get update && sudo apt-get upgrade
```

### Cyclone DDS install

For this library, you have to install the packages using this command:

```
sudo apt-get install cmake git openjdk-11-jdk-headless maven openssl
```

If you don't have a C++ compiler, it can be installed with the following command:

```
sudo apt-get install gcc
```

Install Cyclone DDS:

```
git clone https://github.com/eclipse-cyclonedds/cyclonedds.git
cd cyclonedds
mkdir build
cd build
cmake ..
make
sudo make install
```

If nothing generated errors, you successfully installed Cyclone DDS.

---

**Note:** Cyclone DDS version used in this tutorial

In general you can just use the newest version of Cyclone DDS. But if the newest version does not work for any reason, this tutorial used the last commit: 269f18e.

This commit can be used by using the command:

```
git checkout 269f18e
```

---

## Documentation

The documentation needs the following packages to be installed:

```
sudo apt-get install python3 plantuml python3-pip python3-sphinx
pip3 install sphinxcontrib.napoleon sphinxcontrib-plantuml sphinxcontrib.
→needs
```

Now the repository of the DDS-Demonstrators can be downloaded and the documentation can be build. For the dependencies of individual software programs on this repository, and how to run it. You should check the documentation page containing that application.

```
git clone https://bitbucket.org/HighTech-nl/dds-demonstrators.git
cd dds-demonstrators/docs
make
```

The documentation is stored in the `__result` directory. Now you should open `__result/html/index.html` with your browser for the documentation.

If you use a windows linux subsystem (WSL), then you can use the following command for viewing the documentation in your browser. Otherwise, you have to replace `x-www-browser` with the browser you have installed.

```
x-www-browser __result/html/index.html
```

## Software Quality Control

**authors** Joost Baars

**date** Mar 2020

The C++ software within this project must be checked for issues and should be posted online when there are no issues or warnings in the software.

The software can be checked using static code analyzers. For the C++ software, I've used the static code analyzer called "cppcheck".

The compiler can check a lot of issues too. Therefore, I turn on most of the compiler warnings.

### Cppcheck

A static code analyzer for C++.

### Install

Cppcheck can be installed using the following command:

```
sudo apt-get install cppcheck
```

### Usage

The command below should be executed in the main directory of the C++ project. Ideally, the main directory contains all the source code. If not, `cppcheck` can only partially scan the code. This command could also be executed in a higher directory level containing all source code, then you should exclude all other files that are not the source code.

```
cppcheck --enable=all --inconclusive --std=posix -i build/ .
```

This scans all the files recursively from your main directory. The `-i build/` command tells `cppcheck` not to check the build folder (because these are not C/C++ files). The `-i` can be used for more directories. For each directory, an additional `-i <file/directory name>` tag needs to be used. So for multiple directories that need to be ignored, there are multiple `-i` tags.

The other commands: `--enable=all --inconclusive --std=posix` are enabling all possible warnings/errors for the static code analyzer. These results can be ignored if you have a good reason for it. The results are in general useful and enhance the quality of the code.

All commands of `cppcheck` can be found using only the command `cppcheck` in the terminal.

### Compiler warnings

In C++ I enable the following compiler warnings:

```
-pedantic -Wall -Wcast-align -Wcast-qual -Wdisabled-optimization  
-Winit-self -Wmissing-declarations -Wmissing-include-dirs -Wredundant-decls  
-Wshadow -Wsign-conversion -Wundef -Werror  
-Wempty-body -Wignored-qualifiers -Wmissing-field-initializers  
-Wsign-compare -Wtype-limits -Wuninitialized
```

This list shows most of the warnings. In most cases, warnings mean bad code. Therefore, one of the warnings that is enabled is `-Werror`. This warning makes errors from warnings. Therefore, you must solve the warning before the code becomes compilable again.

### CMake

In CMake, these warnings can be enabled by placing the following in the `CMakeLists.txt` file:

```
# Add warnings
target_compile_options(<Project Name> PRIVATE -pedantic -Wall -Wcast-align -
↳Wcast-qual -Wdisabled-optimization
                                -Winit-self -Wmissing-
↳declarations -Wmissing-include-dirs -Wredundant-decls
                                -Wshadow -Wsign-conversion -
↳Wundef -Werror
                                -Wempty-body -Wignored-
↳qualifiers -Wmissing-field-initializers
                                -Wsign-compare -Wtype-limits -
↳Wuninitialized )
```

<Project Name> should be the name of the project or the library.

## Links

- Cppcheck official website: <https://sourceforge.net/projects/cppcheck/>

## Requirements of performance measurements for DDS

This page contains the requirements for the performance measurements in DDS. It is separated into 2 chapters. One of them contains the requirements of the client. The other chapter contains the specifications. The specifications describe the requirements in a **SMART** way. These specifications are therefore testable to see if the requirement is met. The specifications are, in this project, created by the developer and discussed with the client.

The table below shows all the requirements and specifications. Additionally, it shows the specifications that belong to a requirement. This way, the specifications linked to a requirement can easily be found.





ID	Type	Title	Incoming	Outgoing
<i>Doc_1</i>	req	readable docu- mentation	<i>DOC_DEV</i> ; <i>DOC_INSTRUCTIONS</i> ; <i>DOC_SPH</i> ; <i>DOC_QoS</i>	
<i>Doc_2</i>	req	formal documen- tation	<i>DOC_UML</i> ; <i>DOC_DDS</i>	
<i>Doc_3</i>	req	team pages docu- mentation		
<i>Perf_1</i>	req	performance of DDS	<i>PERF_LIB</i> ; <i>PERF_CPU</i> ; <i>PERF_MEM</i> ; <i>PERF_CONNECT</i> ; <i>PERF_DISCONNECT</i> ; <i>PERF_QoS</i> ; <i>PERF_SCALABLE</i>	
<i>Learn_1</i>	req	learning	<i>LEARN_WRAPPER</i>	
<i>Com- mer_1</i>	req	commercial availability	<i>COMMERCIAL_LICENSE</i>	
<i>API_1</i>	req	API's	<i>API_TEST</i> ; <i>PERF_API</i>	
<i>DOC_DEV</i>	spec	further develop- ment documen- tation		<i>Doc_1</i>
<i>DOC_INSTRUCTIONS</i>	spec	documentation instructions		<i>Doc_1</i>
<i>DOC_SPH</i>	spec	documentation style		<i>Doc_1</i>
<i>DOC_QoS</i>	spec	documentation of Quality of Service policies	<i>PERF_QoS</i>	<i>Doc_1</i> ; <i>PERF_CPU</i> ; <i>PERF_MEM</i> ; <i>PERF_CONNECT</i> ; <i>PERF_DISCONNECT</i>
<i>DOC_UML</i>	spec	PlantUML usage		<i>Doc_2</i>
<i>DOC_DDS</i>	spec	software doc- umentation of DDS		<i>Doc_2</i>
<i>PERF_LIB</i>	spec	software DDS li- brary		<i>Perf_1</i>
<i>PERF_CPU</i>	spec	performance CPU	<i>DOC_QoS</i> ; <i>PERF_QoS</i> ; <i>PERF_SCALABLE</i>	<i>Perf_1</i>
<i>PERF_MEM</i>	spec	performance memory	<i>DOC_QoS</i> ; <i>PERF_QoS</i> ; <i>PERF_SCALABLE</i>	<i>Perf_1</i>
<i>PERF_CONNECT</i>	spec	performance of connecting	<i>DOC_QoS</i> ; <i>PERF_QoS</i> ; <i>PERF_SCALABLE</i>	<i>Perf_1</i>
<i>PERF_DISCONNECT</i>	spec	performance of disconnecting	<i>DOC_QoS</i> ; <i>PERF_QoS</i> ; <i>PERF_SCALABLE</i>	<i>Perf_1</i>
<i>PERF_QoS</i>	spec	performance difference Qual- ity of Service policies		<i>Perf_1</i> ; <i>PERF_CPU</i> ; <i>PERF_MEM</i> ; <i>PERF_CONNECT</i> ; <i>PERF_DISCONNECT</i> ; <i>DOC_QoS</i>
<i>PERF_SCALABLE</i>	spec	documentation of Quality of Service policies		<i>Perf_1</i> ; <i>PERF_CPU</i> ; <i>PERF_MEM</i> ; <i>PERF_CONNECT</i> ; <i>PERF_DISCONNECT</i>
<i>LEARN_WRAPPER</i>	spec	software wrap- per		<i>Learn_1</i>
<i>COM- MER- CIAL_LICENSE</i>	spec	commercial license		<i>Commer_1</i>
<i>API_TEST</i>	spec	different API's	<i>PERF_API</i>	<i>API_1</i>
<i>PERF_API</i>	spec	performance dif- ferent API's		<i>API_1</i> ; <i>API_TEST</i>

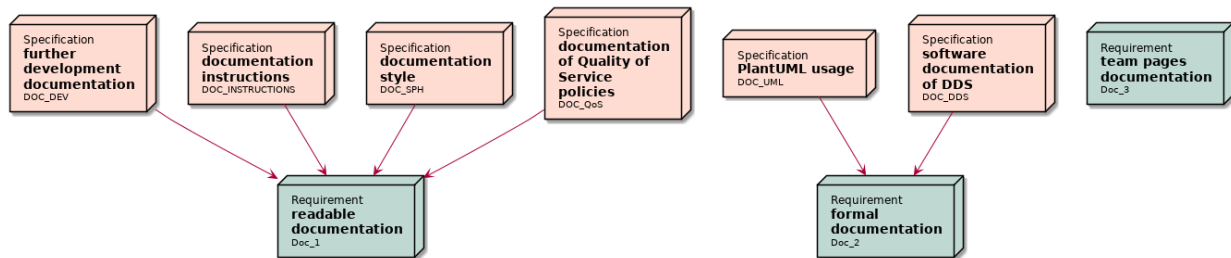
## Requirements

The requirements are given by the client and are not SMART defined.

## Documentation

The graph below shows the linkage between the requirements and the corresponding specifications. These requirements are all related to the documentation.

The requirement of the team pages does not contain any SMART defined specifications. The team pages can be used for documenting things quickly and less precise. There are no additional specifications bound to this requirement.



Requirement: **readable documentation** *Doc\_1*

tags: documentation

links incoming: *DOC\_DEV*, *DOC\_INSTRUCTIONS*, *DOC\_SPH*, *DOC\_QoS*

The documentation must be readable for developers who want to learn more about DDS.

Requirement: **formal documentation** *Doc\_2*

tags: documentation

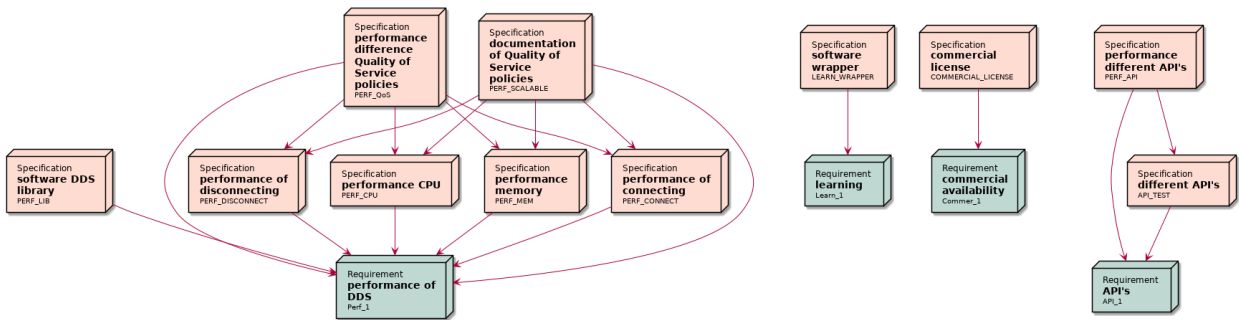
links incoming: *DOC\_UML*, *DOC\_DDS*

The formal documentation is the documentation for the delivered artefacts and should be clear and concise.

Requirement: <b>team pages documentation</b> <i>Doc_3</i>
tags: documentation
The team pages may have less concise language, there are no additional requirements for the team page documentation other than the General requirements.

Software

The graph below shows the link between the requirements and the corresponding specifications. These requirements are all related to the software that shall be developed during the project.



Requirement: <b>performance of DDS</b> <i>Perf_1</i>
tags: software
links incoming: <i>PERF_LIB</i> , <i>PERF_CPU</i> , <i>PERF_MEM</i> , <i>PERF_CONNECT</i> , <i>PERF_DISCONNECT</i> , <i>PERF_QoS</i> , <i>PERF_SCALABLE</i>
The performance of DDS shall be measured with various configurations.

Requirement: <b>learning</b> <i>Learn_1</i>
tags: software
links incoming: <i>LEARN_WRAPPER</i>
The delivered artefacts are made for learning/explaining DDS.

Requirement: <b>commercial availability</b> <i>Commer_1</i>
tags: software
links incoming: <i>COMMERCIAL_LICENSE</i>
The delivered software can be used in commercial products.

Requirement: <b>API's</b> <i>API_1</i>
tags: software
links incoming: <i>API_TEST</i> , <i>PERF_API</i>
The effect of different API's for DDS must be compared with each other, if possible.

### Specifications

The specifications are defined with the client. These are derived from the main requirements.

### Documentation

The specifications for the documentation are listed below.

Specification: <b>further development documentation</b> <i>DOC_DEV</i>
tags: documentation requirement: MUST  links outgoing: <i>Doc_I</i>
<p>Every artefact shall be documented in such extent that a developer knows how DDS is used within the artefact.</p> <hr/> <p><b>Note:</b> This specification can be checked by giving 2 developers the project including the documentation. They should be able to explain how DDS is used within the project.</p> <hr/>

Specification: <b>documentation instructions</b> <i>DOC_INSTRUCTIONS</i>
tags: documentation requirement: MUST  links outgoing: <i>Doc_I</i>
<p>Every artefact contains execution instructions that can be followed by developers.</p> <hr/> <p><b>Note:</b> This requirement can be checked by giving 2 developers the project including the documentation. They should be able to successfully execute the instructions.</p> <hr/>

Specification: <b>documentation style</b> <i>DOC_SPH</i>
tags: documentation requirement: MUST  links outgoing: <i>Doc_I</i>
<p>The documentation shall be written in ReStructured Text for Sphinx.</p>

Specification: <b>documentation of Quality of Service policies</b> <i>DOC_QoS</i>
tags: documentation requirement: MUST  links outgoing: <i>Doc_1</i> , <i>PERF_CPU</i> , <i>PERF_MEM</i> , <i>PERF_CONNECT</i> , <i>PERF_DISCONNECT</i> links incoming: <i>PERF_QoS</i>
<p>The performance differences between Quality of Service policies are documented.</p> <hr/> <p><b>Note:</b> There must be results of performance measurements that show what Quality of Service policies are faster or slower based on the performance measurements (performance CPU (<i>PERF_CPU</i>), performance memory (<i>PERF_MEM</i>), performance of connecting (<i>PERF_CONNECT</i>), performance of disconnecting (<i>PERF_DISCONNECT</i>)). At least 2 different Quality of Service policies must be compared with each other.</p> <hr/>

Specification: <b>PlantUML usage</b> <i>DOC_UML</i>
tags: documentation requirement: MUST  links outgoing: <i>Doc_2</i>
<p>All diagrams within the Sphinx documentation shall be written using PlantUML, if they are supported by the PlantUML language.</p> <hr/> <p><b>Note:</b> This also includes the extended functionalities of PlantUML like Ditaa and DOT.</p> <hr/>

Specification: <b>software documentation of DDS</b> <i>DOC_DDS</i>
tags: documentation requirement: MUST  links outgoing: <i>Doc_2</i>
The use of DDS within the software shall be documented.
<b>Note:</b> The documentation of the software shall contain a technical explanation of how DDS is used within the application.

## Software

The specifications for the software are listed below.

Specification: <b>software DDS library</b> <i>PERF_LIB</i>
tags: software requirement: MUST  links outgoing: <i>Perf_1</i>
The Cyclone DDS library shall be used as the DDS implementation.
<b>Note:</b> Only if something is not implemented in the CycloneDDS library, a different DDS implementation may be used for the tests.

Specification: <b>performance CPU</b> <i>PERF_CPU</i>
tags: software requirement: SHOULD  links outgoing: <i>Perf_1</i> links incoming: <i>DOC_QoS, PERF_QoS, PERF_SCALABLE</i>
The CPU usage of different Quality of Service policies shall be measured.

Specification: <b>performance memory</b> <i>PERF_MEM</i>
tags: software requirement: SHOULD  links outgoing: <i>Perf_1</i> links incoming: <i>DOC_QoS, PERF_QoS, PERF_SCALABLE</i>
The memory usage of different Quality of Service policies shall be measured.

Specification: <b>performance of connecting</b> <i>PERF_CONNECT</i>
tags: software requirement: MUST  links outgoing: <i>Perf_1</i> links incoming: <i>DOC_QoS, PERF_QoS, PERF_SCALABLE</i>
The time between the DDS registration of a device and the notification of other devices shall be measured.



Specification: **performance of disconnecting** *PERF\_DISCONNECT*

tags: software

requirement: MUST

links outgoing: *Perf\_1*

links incoming: *DOC\_QoS*, *PERF\_QoS*, *PERF\_SCALABLE*

The time between the DDS disconnection of a device and the notification of other devices shall be measured.

Specification: **performance difference Quality of Service policies** *PERF\_QoS*

tags: software

requirement: MUST

links outgoing: *Perf\_1*, *PERF\_CPU*, *PERF\_MEM*, *PERF\_CONNECT*, *PERF\_DISCONNECT*, *DOC\_QoS*

The performance of Quality of Service policies within DDS shall be compared to each other.

Specification: **documentation of Quality of Service policies** *PERF\_SCALABLE*

tags: software

requirement: SHOULD

links outgoing: *Perf\_1*, *PERF\_CPU*, *PERF\_MEM*, *PERF\_CONNECT*, *PERF\_DISCONNECT*

The difference in performance, CPU/memory footprint, registration time, disconnection time, shall be recorded with 2 devices in the network compared to at least 4 devices in the network.

**Note:** This requirement tests the scalability of DDS.

Specification: **software wrapper** *LEARN\_WRAPPER*

tags: software

requirement: SHOULD

links outgoing: *Learn\_1*

The functions of Cyclone DDS shall not be used within a wrapper.

Specification: **commercial license** *COMMERCIAL\_LICENSE*

tags: software

requirement: MUST

links outgoing: *Commer\_1*

The delivered software shall be in accordance with the MIT, BSD or Apache licenses, with an exception for the DDS library.

Specification: **different API's** *API\_TEST*

tags: software

requirement: SHOULD

links outgoing: *API\_1*

links incoming: *PERF\_API*

The communication between 2 different APIs shall be tested (The C/C++ API and the Python API).

Specification: <b>performance different API's</b> <i>PERF_API</i>
tags: software requirement: SHOULD  links outgoing: <i>API_1</i> , <i>API_TEST</i>
The difference in performance, speed of the implementation, shall be measured between different APIs (if this is possible, see requirement <a href="#">different API's (API_TEST)</a> ).

## 1.2.6 Things to do:

### DDS-Demonstrators

Many Demonstrators can/do exist in several variants. And/or measurements between the variants are useful. Typically there is a backlog of Demonstrators and Requirements (on e.g. variants)

- See the *Demonstrators*, for some ideas
- And the 'incoming' links there for variants or requirements

### ToDoList

There are also a lot of things on the todo list; collected from the docs

---

**ToDo:** SetUp the M&B comparison between DDS and other protocols in this environment

---

**Tip:** Is this part of "DDS-Demonstrators", or another HighTech-NL TechPush project?

---

(The [original entry](#) is located in `/home/docs/checkouts/readthedocs.org/user_builds/dds-demonstrators/checkouts/latest/docs/doc/Backlog/1.RoundTrip+Flood/hints.rst`, line 10.)

---

**ToDo:** Fix *MQTT, DDS, ZMQ Matrix board application* in C++.

The matrix board application sends out messages each 24 milliseconds, and is also receiving messages from two other boards at this pace. Each time a message is sent or received, the message and a timestamp are written to a file. This is done by creating a thread so that the main applications performance will be affected minimally by writing these messages. The application may crash after a while due to the high number of threads created to write measurements. An error will occur because no further threads can be started at that point. If you think you can go ahead i wish you the best of luck!

---

(The [original entry](#) is located in `/home/docs/checkouts/readthedocs.org/user_builds/dds-demonstrators/checkouts/latest/docs/doc/Demonstrators/ComparedToOtherProtocols/MatrixBoard/rt_matrix_board_C++.rst`, line 63.)

---

---

**Todo:** Install CycloneDDS on a Windows machine.

---

(The [original entry](#) is located in `/home/docs/checkouts/readthedocs.org/user_builds/dds-demonstrators/checkouts/latest/docs/doc/Teams/1.Hurricane/setupCycloneDDS.rst`, line 81.)

---

**Todo:**

- Research the cause of the connection issue
  - Research the cause of the incompatibility with the ported library.
  - A project to make multiple bare-metal devices communicate with each other for a round-trip
- 

(The [original entry](#) is located in `/home/docs/checkouts/readthedocs.org/user_builds/dds-demonstrators/checkouts/latest/docs/doc/Teams/2.Demonstrators/7.NucleoDDS/DDSCode.rst`, line 311.)

**Ask ...**

For all assignments, ask the PO (albert)