A REAL-TIME COMPUTER ARCHITECTURE BASED ON A CLIENT-SERVER APPROACH FOR A MULTI-ARM ROBOT MANIPULATION (MARM) PLATFORM

D. Antonucci[1], A. Margan[1], A. Laurenzi[1], A. Rodriguez[2], P. Romeo[2], J. Barrientos[2], J. Estremera[2], A. Rusconi[3], G. Sangiovanni[3], N.G. Tsagarakis[1], and S. Cordasco[1]

[1]*Istituto Italiano di Tecnologia (IIT), Via S. Quirico 19d, 16163 Genova GE*
[2]*GMV, Isaac Newton 11 Tres Cantos 28760, Madrid, Spain*
[3]*Leonardo S.p.A., Viale Europa, 20014, Nerviano, Italy*

ABSTRACT

This paper introduces the real-time computer architecture developed for the Multi-arm Robot Manipulation (MARM) platform realized within the MIRROR project: Multi-Arm Installation Robot for Reading ORUS and Reflectors, Figure 1.

We provide an extensive description about implementation challenges and design decisions, and finally validate our architecture on the real robot testing it with different use-cases. Furthermore, we also considered and managed some safety aspects related in situations of control deterioration due to communication quality degradation, considering also recovery actions.

Key words: real-time; safety, MARM; MIRROR.

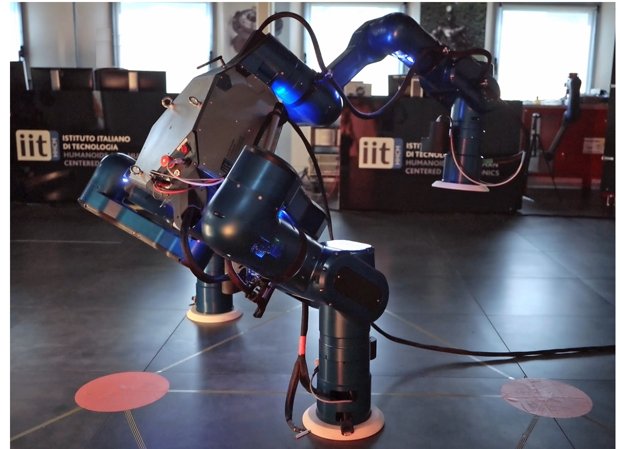

Figure 1. The Multi-Arm Relocatable Manipulator (MARM) robotic platform developed for SPACE application.

1. INTRODUCTION

The recent increasing demand of robotic platforms for space applications, highlights the need of effective and robust embedded hardware and software architectures that ensure the real-time performance of the customized robotic platforms, providing flexibility in implementing high level control modules the execution of which is decoupled from the execution of the real-time critical software components. To achieve this, a physical separation in term of embedded systems is needed, where one embedded computation unit is dedicated for real-time (RT) performance needed for controlling the actuators, sensors reading or other RT devices operations and the other one (or more) devoted for high level control tasks that do not strictly require hard real-time performance (i.e perception and visual servoing, gravity compensation or cartesian space tasks etc..). This avoids most issues related to software dependencies such as different operating systems, libraries or other OS environmental conditions, or hardware performances and resources sharing and needs between hard and soft real-time processes.

The Figure 2 shows the client-server approach and hardware components in the MIRROR project where the client runs together with the high level application implementation inside the Robot Control Unit.

The server, instead, is an internal process of an EtherCAT Master module, which runs in its dedicated Embedded PC, and exploits the EtherCAT protocol to read or write data objects to the EtherCAT slaves (RT devices installed on the robotic platform). In order to interact and operate them ,the high level application needs to instance a client calling the APIs provided enabling the interaction between the high level application and the low level RT components through a set of commands that include:

- client/server protocol initialization (connect, disconnect, quit server, etc..);
- setting of control modes;
- start/stop actuation;
- release/engage brakes;

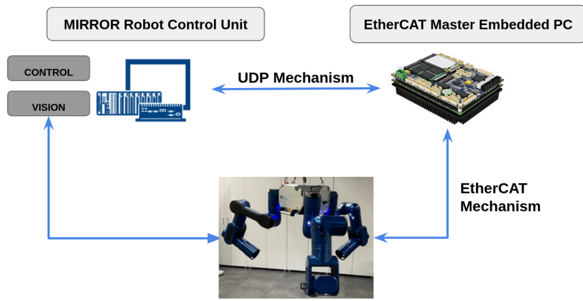

Figure 2. High level overview of the hardware and software/communication design of the MARM robotic platform

- read actuation states, force-torque sensor, IMU, power board date and the states of other devices providing full telemetry of the robot state;
- write actuation set point references;
- read/write service data object;

2. ARCHITECTURE DESIGN

The first study, trial and consolidation of this architecture employs the UDP protocol to exchange data, in particular, the boost::asio [1] is used for the network protocol while the msgpack [2] performs the packing and unpacking of data. The UDP choice is a project request since these APIs are used by our partner (GMV company) within the project MIRROR. Instead the use of packaging helps to manage the data transfer quickly and transparently.

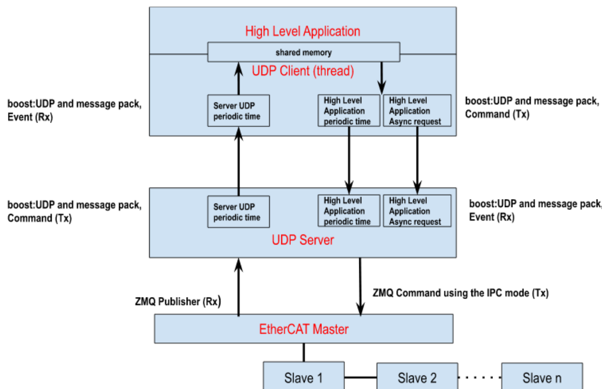

Figure 3. Schematic overview of the proposed software and control architecture

Both Client and Server have to open an UDP socket where the communication should be established, then create command and event handlers. The commands are used by the client to send requests to the server like start/stop motors, release and engage the brakes or send references, in the same way, the server sends the motors status or other information to the client. As the communication protocol is asynchronous the event handlers are important for receiving between the endpoints, i.e ACK/NACK of a client's command from the server that has already handled the request via its event handler.

2.1. Event handler

The event handlers are simple functions that have to be registered, i,e:

```
// Register Message Handler
registerHandler(ServerMsg::MSG_MOTOR_STATUS,
&Client::motor_status_handler);
```

When the event is raised the function registered will be called and then it's possible to unpack the data and manipulate them.

```
void Client::motor_status_handler(char *buf,
                                  size_t size)
{
   _mutex_motor_status->lock();

    static MSS motors_status;
   // unpacking
   auto ret =
   proto.getEscStatus(buf,
                      size,
                      ServerMsg::MSG_MOTOR_STATUS,
                      motors_status);
   // manipulation
   .......
   _mutex_motor_status->unlock();
}
```

Note: The mutex mechanism is useful to synchronize the client code with the high level application process.

2.2. Command handler

As already mentioned the commands are used to send request to the server or client, i.e:

```
bool Client::start_motors(const MST &motors_start)
{
    CBuffT<4096u> sendBuffer{};
    bool ret_cmd_status=false;
    // packing
    auto sizet =
    proto.packReplRequestMotorsStart(sendBuffer,
                                     motors_start);
```

[1] The boost::asio library is described at https://www.boost.org/doc/libs/1_78_0/doc/html/boost_asio.html

[2] The msgpack library is described at https://msgpack.org/

```
    // send
    do_send(sendBuffer.data(),
         sendBuffer.size());

    // ACK/NACK information
    ret_cmd_status= get_reply_from_server(
                    ReplReqRep::START_MOTOR,
                    repl_msg);
    return ret_cmd_status;
}
```

When a command is called it's possible to pack the data and send them, waiting eventually the ACK/NACK feedback.

2.3. Client-Server state machine

Two state machines were implemented to make the mechanism more consistent due to the asynchronous protocol, helping high level robotic controller to send the commands in the right way, getting right feedback. The Figure 4 shows the UML state machine diagram of the UDP Server:

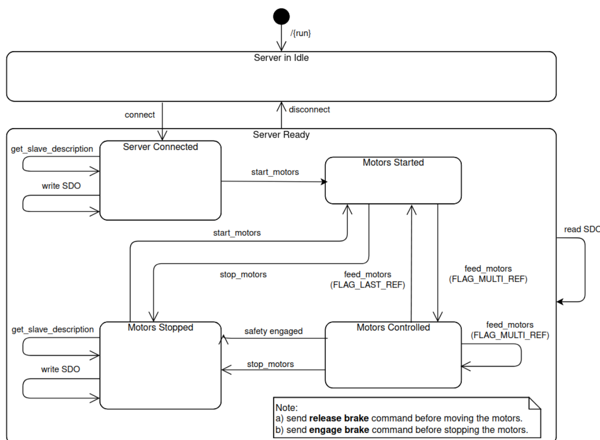

Figure 4. Schematic overview of the UDP sever state machine

The Server is the process that runs with EtherCAT Master and waits that a Client sends the `connect` command. When one of it is detected, its state change into connected where it's possible to receive the commands for managing the real-time slaves (Server Ready state). Furthermore, it communicates its alive state with a specific periodicity. This is useful to understand if the mechanism is still standing or not. Of course you can return to idle state with the `disconnect` command. Staying in the connection state, instead, the server can move in different "operative states", Motors Started, Controlled and Stopped with Client-Server software APIs. An important consideration is related on Motors Controlled state, here, it's possible to move or stop the robot remembering to use the release or engage brakes command in according with robotic controller decisions. The `feed_motors` API is used to send references with different flag options, MULTI REF for continuous robot manipulation and LAST REF for completing the movement and returning back in the Motors Started State. In addition, safety checks were implemented during the robot motion in order to guarantee the communication quality that will be taken up at Section 3. The Server sends continuously the state of the robot, including the state of the motors, sensors and other slaves as soon as it receives the `get_slaves_description` command before entering the operative states or in Motors stopped, where it is also possible to write the service data objects (SDO). Instead, the read SDO API is always available when the server is connected.

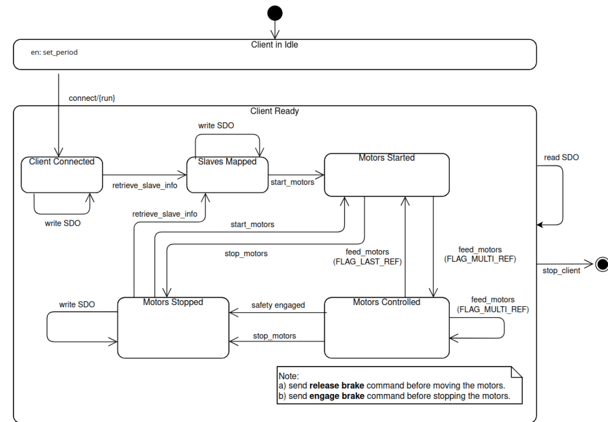

Figure 5. Schematic overview of the UDP client state machine

The Figure 5, instead, presents the UML diagram of the UDP Client. The Client is a thread of the high level application process which is responsible to call the APIs in according with the robot control goal. At first it's necessary to setup the periodicity and then send the `connect` command moving the Client's state machine from idle to connected. Here (Client Ready state), it's possible to receive back the server status synchronizing the Client-Server state machines, verifying also that the communication is still alive. Before entering in the "operative states" (Motors Started, Controlled and Stopped) like the server case, an auto-detection or discovery procedure of EtherCAT slaves is needed, using the `retrieve_slave_info` request. Moving the machine into the Slaves Mapped, the high application (robotic control) can start to read the robot state (motors, sensors or other real-time devices) closing its control loop. Finally it's possible to operate the robot exploiting the same APIs described in the server state machine. About the SDO APIs, read and write, these are useful during the calibration and testing phase, for this reason, the writing operation is available only in some states.

2.4. Client-Server sequence diagram

The following schematic in Figure 6 presents the UML sequence communication diagram of a typical use case

for using the robot.

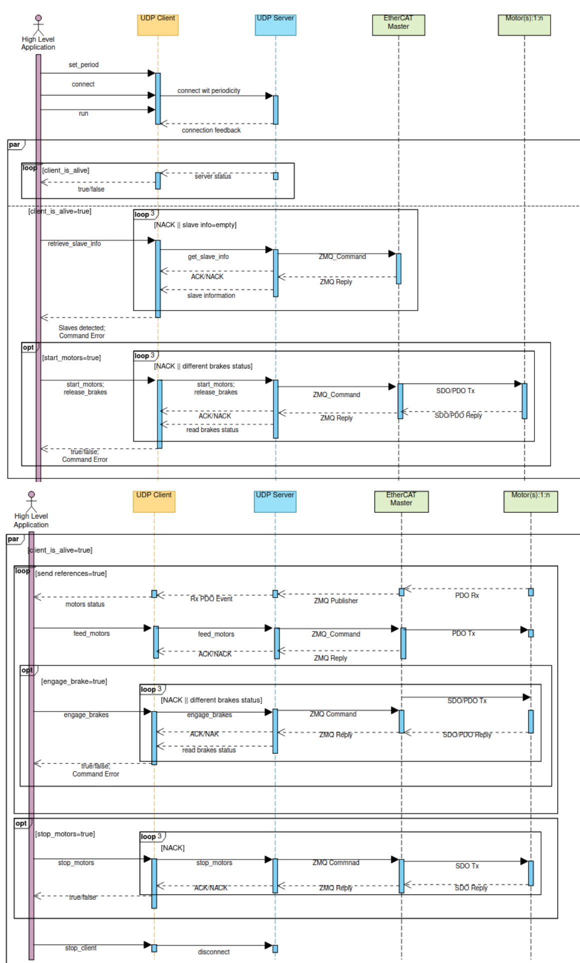

Figure 6. The sequence communication diagram of the overall system

The high level application, for starting the communication flow, needs to set the client periodicity, send the `connect` command and detach a thread calling the `run` API (since it's a blocking call from boost::asio specification), capturing the server events, status, feedback and data (PDOs). At this point the high level application (main process) can operate the robot calling the following commands creating a typical use case:

- retrieve slave info;
- start motor controllers and release brakes;
- send references
- engage brakes (before stopping phase);
- stop motors;
- stop client;

As it can be observed in the diagram, every command is an optional activity based of the success of the previous step. The client tries for three times the commands before getting the right answer from the server. A particular situation is the send references activity where the `feed_motors` API is called until the robot manipulation is completed, reading also the motors status. When this is completed it is then possible to engage the brakes for stopping the motors. The `stop_client` request allows the application to close the client and server communication. Note that this diagram describes a common use case for operating the robot, but it's possible to implement also other sequences for operating the platform, especially for managing the release/engage brakes APIs. For instance, they might be used in the send references loop, the main necessity is is to use the release the brakes command before moving the robot and engage the brakes before stopping the motor controllers.

3. SAFETY

Two safety controls were also developed to verify the communication in terms of communication degradation as well as the healthy operation of the robot actuation system. The `rolling_mean` functions in the boost library [3] are used to evaluate the communication quality in a specific window. The mean in that window is controlled verifying the desired communication frequency for the data exchange along the communication pipeline, activating appropriate recovery actions in the system control if degradation of the communication quality is detected. The robotic controller can choose different communication frequencies using the set period API from the safety frequencies range allowed, greater a minimum frequency level of 225 Hz or less than or equal to 500 Hz. Furthermore, the rolling window size can be set before running the server process, the default size of which is equal to 100.

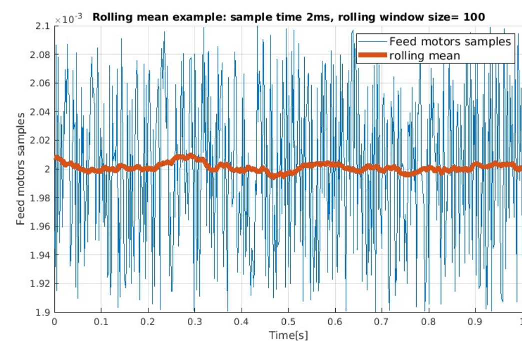

Figure 7. Rolling mean function example

The Figure 7 introduces an example for the rolling mean function calculated on 1s communication having as sample time equal of 2ms, getting the samples randomly in

[3]The rolling mean function is is described at `https://www.boost.org/doc/libs/1_81_0/doc/html/accumulators/user_s_guide.html#accumulators.user_s_guide.the_statistical_accumulators_library.rolling_mean`

that range (blue line), computing the rolling mean with a window equal to 100 (read line).

The Figure 8 shows a typical server log during the send references operation:

```
2ms --> rolling window 100

************************LOG****
-----------------------------------------------------------------
10:30:43.467] [server] [info]  REPL REQ : SET_MOTOR_REFS
[10:30:43.467] [server] [info]  REPL REQ : SET_MOTOR_REFS
[10:30:43.467] [server] [info] mean refs freq mean 0.0020001714700000014
[0MQ Req] connect to ipc:///tmp/ecat_master:5555
[10:30:43.468] [server] [info]  send request reply SET_MOTOR_REFS
[10:30:43.469] [server] [info]  REPL REQ : SET_MOTOR_REFS
[10:30:43.469] [server] [info]  REPL REQ : SET_MOTOR_REFS
[10:30:43.469] [server] [info] mean refs freq mean 0.002001261530000001
[0MQ Req] connect to ipc:///tmp/ecat_master:5555
[10:30:43.470] [server] [info]  send request reply SET_MOTOR_REFS
[10:30:43.471] [server] [info]  REPL REQ : SET_MOTOR_REFS
[10:30:43.471] [server] [info]  REPL REQ : SET_MOTOR_REFS
[10:30:43.471] [server] [info] mean refs freq mean 0.002000142350000001
[0MQ Req] connect to ipc:///tmp/ecat_master:5555
[10:30:43.472] [server] [info]  send request reply SET_MOTOR_REFS
[10:30:43.473] [server] [info]  REPL REQ : SET_MOTOR_REFS
[10:30:43.473] [server] [info]  REPL REQ : SET_MOTOR_REFS
[10:30:43.473] [server] [info] mean refs freq mean 0.001999978140000001
[0MQ Req] connect to ipc:///tmp/ecat_master:5555
[10:30:43.474] [server] [info]  send request reply SET_MOTOR_REFS
[10:30:43.475] [server] [info] periodicActivity tDiff 0.001836719
[10:30:43.475] [server] [info]  REPL REQ : SET_MOTOR_REFS
[10:30:43.475] [server] [info]  REPL REQ : SET_MOTOR_REFS
[10:30:43.475] [server] [info] mean refs freq mean 0.002000213480000001
[0MQ Req] connect to ipc:///tmp/ecat_master:5555
[10:30:43.476] [server] [info]  send request reply SET_MOTOR_REFS
[10:30:43.477] [server] [info]  REPL REQ : SET_MOTOR_REFS
[10:30:43.477] [server] [info]  REPL REQ : SET_MOTOR_REFS
[10:30:43.477] [server] [info] mean refs freq mean 0.002000003240000001
[0MQ Req] connect to ipc:///tmp/ecat_master:5555
[10:30:43.478] [server] [info]  send request reply SET_MOTOR_REFS
[10:30:43.479] [server] [info]  REPL REQ : SET_MOTOR_REFS
[10:30:43.479] [server] [info]  REPL REQ : SET_MOTOR_REFS
[10:30:43.479] [server] [info] mean refs freq mean 0.002000028500000001
[0MQ Req] connect to ipc:///tmp/ecat_master:5555
[10:30:43.480] [server] [info]  send request reply SET_MOTOR_REFS
[10:30:43.481] [server] [info]  REPL REQ : SET_MOTOR_REFS
[10:30:43.481] [server] [info]  REPL REQ : SET_MOTOR_REFS
[10:30:43.481] [server] [info] mean refs freq mean 0.002000024840000001
[0MQ Req] connect to ipc:///tmp/ecat_master:5555
[10:30:43.482] [server] [info]  send request reply SET_MOTOR_REFS
[10:30:43.483] [server] [info]  REPL REQ : SET_MOTOR_REFS
[10:30:43.483] [server] [info]  REPL REQ : SET_MOTOR_REFS
[10:30:43.483] [server] [info] mean refs freq mean 0.002000026470000001
[0MQ Req] connect to ipc:///tmp/ecat_master:5555
[10:30:43.484] [server] [info]  send request reply SET_MOTOR_REFS
```

Figure 8. Rolling mean logging of the UDP Server

The other safety check is related to the detection of complete communication break. In particular in the case that the server detects that the time difference between the last actuation set-points and the actual server periodicity is greater than a specific period the recovery action is activated to safely terminate the operation of the system and bring the robot actuation in idle mode engaging their brakes.

4. USE CASES AND VALIDATION

The proposed architecture was implemented and validated on the MIRROR robot platform, under different control modes including position and impedance control settings. Furthermore, we also validated the proposed architecture and safety features while the robot was controlled in pure torque mode (impedance mode with stiffness and damping gains equal to zero) providing the joint torque references derived from the gravity compensation module.

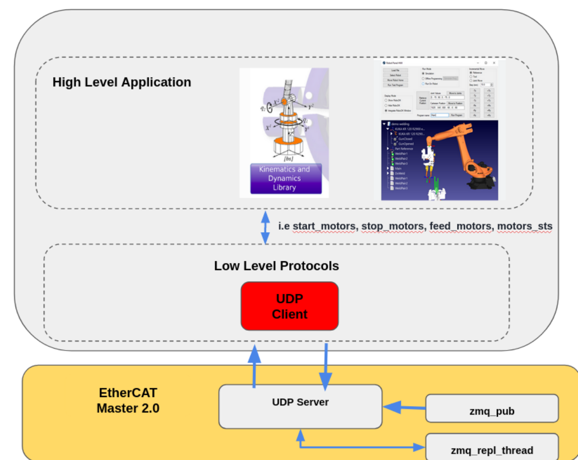

Figure 9. High level schematic of the use cases

Basically two use cases were implemented, one realizes a trajectory generator module that produces joint space references, starting the motors with different control mode and gains, and the other one implements a gravity compensation task using a library called XBot-Interface (ModelInterface) developed by Humanoids and Human Centered Mechatronics (HHCM) research line [LATM23, MLMHT20, LHMT19]. All use cases call the APIs in the same order shown in the UML sequence diagram (Figure 6), they also use an external configuration file where it's possible to setup the UDP protocol, control mode, trajectories (Homing, General trajectory), high level robotic libraries, URDF and SRDF:

```
network:
hostname: localhost
port: 54321
timeout: 1000

###############position###############
#control_mode: position
#position gains
#gains: [200.0,0.0,10.0,0.0,0.0]
###############position###############

###############impedance###############
control_mode: impedance
#impedance gains
# stiffness, damping, tau_p, tau_f, tau_d
gains: [1000.0,10.0,1.0,0.7,0.007]
###############impedance###############


UDP_period_ms: 4
homing_position: {11: -0.75, 12: -1.0, 13: -1.0, 14: -0.75, 15: 1.0, 16: -0.75,
                  21: -0.75, 22: -1.0, 23: -1.0, 24: -0.75, 25: 1.0, 26: -0.75,
                  31: -0.75, 32: -1.0, 33: -1.0, 34: -0.75, 35: 1.0, 36: -0.75}
homing_time_sec: 3
trajectory: {11: 0.0, 12: -1.9, 13: -2.3, 14: 0.0, 15: -0.4, 16: 0.0,
             21: 0.0, 22: -1.9, 23: -2.3, 24: 0.0, 25: -0.4, 26: 0.0,
             31: 0.0, 32: -1.9, 33: -2.3, 34: 0.0, 35: -0.4, 36: 0.0}
trajectory_time_sec: 3
repeat_trj: 3


slave_id_led: [16,26,35]

XBotInterface:
urdf_path: ${PWD}/urdf/mirror.urdf
srdf_path: ${PWD}/srdf/mirror.srdf
joint_map_path: ${PWD}/joint_map/mirror_joint_map.yaml

ModelInterface:
model_type: "RBDL"
is_model_floating_base: "true"
```

Figure 10. Client-Server configuration file

During the send references loop, the joint space trajectory generator produces the joint trajectories, switching n times from homing to the execution of the generated tra-

jectories. The gravity task, instead, having started the motors in impedance mode, regulates smoothly to reduce the joint stiffness and damping gains to zero, leaving a pure torque reference to be tracked by the torque controller running on the robot joints. It then start to send the torque references to compensate the gravity, calculated by the ModelInterface [LATM23, MLMHT20, LHMT19]. As soon as the compensation is interrupted by the user, they gains will be smoothly restored, having back the impedance controller ready to follow the position references received by the task.

It important to report that all use cases were tested with different sample times (2..4ms) having always an Ethernet point to point connection between the real robot and the external machine.

The following flow chart in Figure 11 presents the use of the APIs for testing and validation our MIRROR project partner, the GMV company.

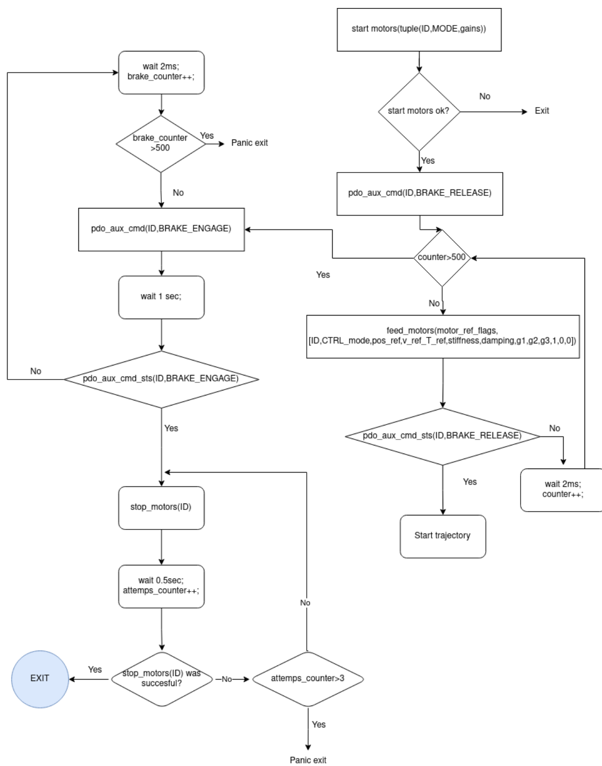

Figure 11. The GMV test procedure

They employ the torque controller loop where the their high level application calculates the gravity compensation torque needed for each joint, having a centralized robotic controller. They initially start the motors in impedance mode, sending the release brake command without waiting the real brake status. Following this, the reference torques command are sent to compensate the gravity due the releasing brake action. If the brake status is equal to released, the torque trajectory is sent, of course, reducing the stiffens and damping to zero, otherwise the stopping procedure is triggered, engaging the brakes and stopping the motors.

4.1. Graphical use interface

Figure 9 introduces an example of GUI developed e for rapid testing and robot monitoring and debugging.

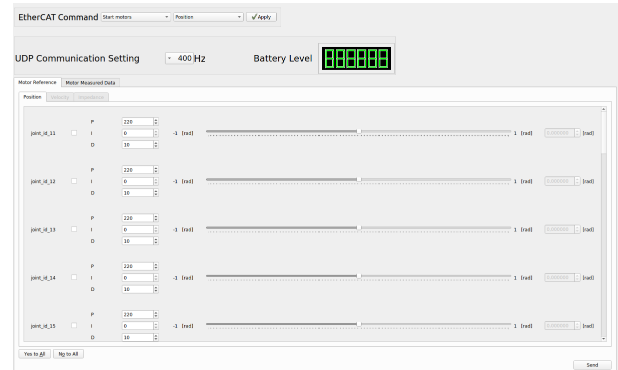

Figure 12. Schematic overview of the MIRROR GUI

In fact, using the Motor Reference page of the GUI it s possible to start the MARM robot joint controllers in position, impedance and idle mode, while also setting the control gains. By Setting up the UDP communication and by pressing the send button interface it's possible to send the references ot the joints moving the joints individually and verifying the telemetry in the Motor Measured Data page of the GUI. The stop reference button can be used to stop sending the references to the joints and eventually stop the controllers.

5. DISCUSSION AND CONCLUSIONS

We are currently working on extending the C++ APIs to also include Matlab/Simulink and eventually Python APIs.

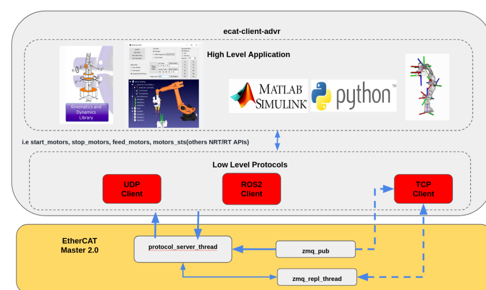

Figure 13. Proposal of next software framework development

This will allow the users to develop and design their high level control strategies in different software environments facilitating fast prototyping and tuning of the system high level control and behaviors. Another goal is to extend the communication protocol from UDP integrating ROS2 in the future. The TCP protocol, instead, was the first implementation done inside the EtherCAT Master (2.0) using

the ZMQ library and google protocol buffer for serialization and de-serialization of data.

An important discussion point is related on the simulation and testing of this software framework. At the moment it can't offer the user the possibility to perform robot model validation or robotic control strategy testing inside a kinematics or dynamics simulator like RViz or Gazebo or others. Of course, this development was out of scope from the MIRROR project, where the HHCM research line was responsible to cover this part [HLR+23]. For this reason, we would develop other features related on the simulation, maybe using ROS2, especially for testing where actually is possible to verify the behaviour only with the terminal log. Another aim is the improvement of the logging phase for the post processing analysis.

ACKNOWLEDGMENTS

The development of the MARM platform is funded by the European Space Agency (ESA) as part of the MIRROR project.

REFERENCES

[HLR+23] E.M. Hoffman, A. Laurenzi, F. Ruscelli, L. Rossini, L. Baccelliere, D. Antonucci, A. Margan, P. Guria, M. Migliorini, S. Cordasco, R Raiola, L. Muratore, J. Estremera, A. Rusconi, G. Sangiovanni, and N.G. Tsagarakis. Design and validation of a multi-arm relocatable manipulator for space applications. In *2023 IEEE ICRA*. IEEE, 2023.

[LATM23] Arturo Laurenzi, Davide Antonucci, Nikos Tsagarakis, and Luca Muratore. The xbot2 real-time middleware for robotics. *Robot. Auton. Syst.*, 163:104379, 2023.

[LHMT19] Arturo Laurenzi, Enrico Mingo Hoffman, Luca Muratore, and Nikos Tsagarakis. CartesI/O: A ROS Based Real-Time Capable Cartesian Control Framework. In *IEEE Int. Conf. Robot. Autom.*, pages 591–596, 2019.

[MLMHT20] Luca Muratore, Arturo Laurenzi, Enrico Mingo Hoffman, and Nikos Tsagarakis. The XBot real-time software framework for robotics: From the developer to the user perspective. *IEEE Robot. Autom. Mag.*, 27(3):133–143, 2020.