# 1 UDP Client and Server architecture



Client and Server exploit the UDP protocol to exchange data, in particular, boost::asio for the network protocol and msgpack for packing and unpacking data.

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.

## 1.1 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 main process that will use it. This concept will be brought back in the next session (5.3.10-11)**

## 1.2 Command handler

As already said 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);

**// send**

do\_send(sendBuffer.data(), sendBuffer.size() );;

**// ACK/NACK information**

ret\_cmd\_status = get\_reply\_from\_server(ReplReqRep::START\_MOTOR,repl\_msg);

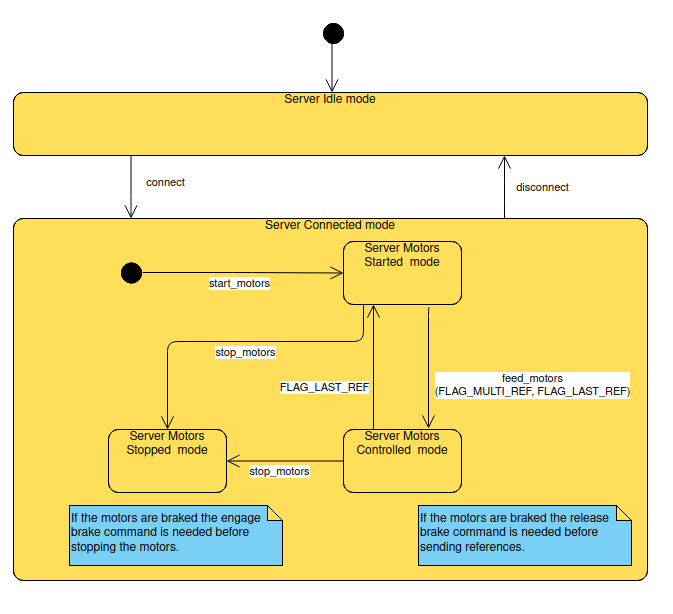
return ret\_cmd\_status;

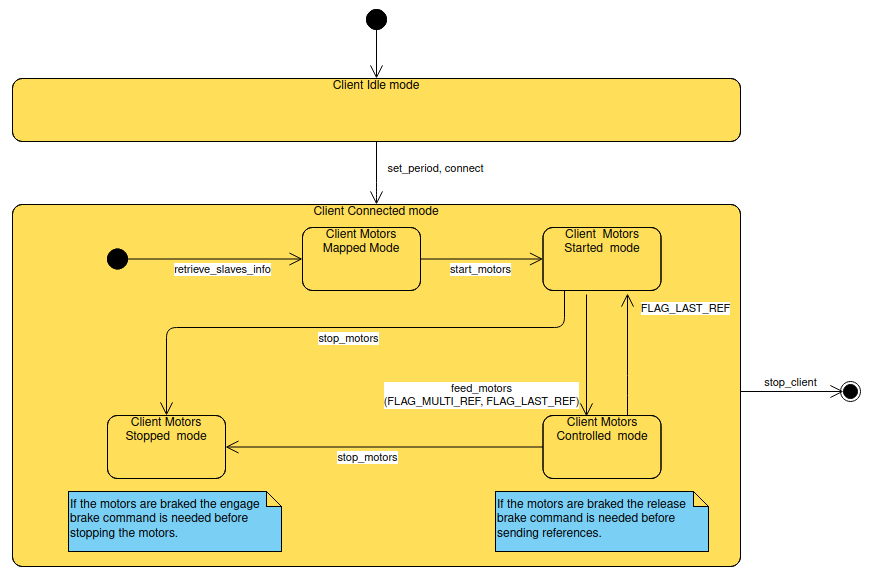
}

**Note: the ACK/NACK information is important for the client for receiving back feedback on IMPORTANT commands like start or stop motors.**

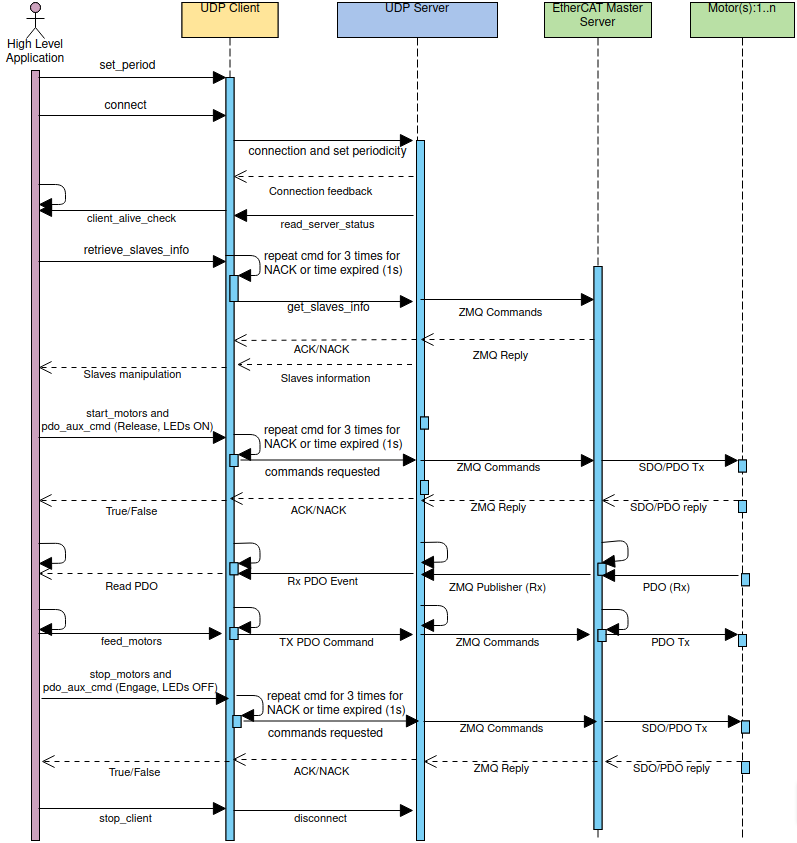
## 

## 1.3 UDP Server and Client STM (State machine)





## 1.4 UDP Client and Server sequence diagram



# 2 Client API (command and event handlers)

## 2.1 connect

Start connection with the UDP server.

**client.connect()**

## 2.2 disconnect

Disconnection with the UDP server.

**client.connect()**

## 2.3 quit\_server

Stop/Kill the UDP server (NOT USED).

**client.quit\_server()**

## 2.4 retrieve\_slaves\_info

Retrieve slaves (motor, power board, etc…) information, slave ID, type and EtherCAT slave position.

**cmd\_status=client.retrieve\_slaves\_info(slave\_info)**

OUTPUT:

**slave\_info**: Slave ID, type (motor, power board, etc…) and EtherCAT slave position.

**cmd\_statu**: boolean value if the retrieve slave info command was received using ACK/NACK feedback coming from the server.

## 2.5 retrieve\_rr\_sdo

Read/Write slaves (motor, power board, etc…) SDO information.

**cmd\_status=client.retrieve\_rr\_sdo(slave\_id,rd\_sdo,wr\_sdo,rr\_sdo\_info)**

INPUT:

**slave\_id**: slave identifier.

**rd\_sdo**: read sdo, string vector.

**wr\_sdo:** write sdo, vector of tuples (string, value).

OUTPUT:

**rr\_sdo\_info**: SDO information (only for reading).

**cmd\_statu**: boolean value if the retrieve rr sdo command was received using ACK/NACK feedback coming from the server.

## 2.6 start\_motors

Start motor for all motors specified by ID with control mode and gains.

**motor\_started=client.start\_motors(motors\_start);**

INPUT:

**motors\_start:** vector of tuples having these fields: **motor ID, control mode 3B (position) or D4 (impedance) and gains (position or impedance gains).**

**position gains:**

| **Gain type** | **Description** | **Default value** | **Valid range** | **Precaution** |
| --- | --- | --- | --- | --- |
| Kp | Proportional gain of the PID position controller. | 200.0 | 50-3000 | Modification of the default gain setting may result to system instability and is not recommended |
| Ki | Integral gain of the PID position controller. | 0.0 | 0.0 | Modification of the default gain setting may result to system instability and is not recommended |
| Kd | Derivative gain of the PID position controller. | 10.0 | 0-60.0 | Modification of the default gain setting may result to system instability and is not recommended |
| G1 | Additional gain parameter for future use | 0.0 | 0.0 | Not used. |
| G2 | Additional gain parameter for future use | 0.0 | 0.0 | Not used. |

**Note!!!:**

**For the first three joints starting from the shoulder (shoulder yaw, shoulder pitch and elbow pitch): Kp<=3000, Kd<=60**

**For the last three joints (forearm yaw, wrist pitch and wrist yaw): Kp<=2000, Kd<=40.**

**impedance gains**

| **Gain type** | **Description** | **Default value** | **Valid range** | **Precaution** |
| --- | --- | --- | --- | --- |
| Kp | Stiffness gain of the impedance controller. | 500.0 | 50-2000 | Modification of the default gain setting may result to system instability and is not recommended |
| Kd | Damping gain of the impedance controller. | 10.0 | 0-50.0 | Modification of the default gain setting may result to system instability and is not recommended |
| tau\_p | Inertia scaling gain of torque controller. | 1.0 | 1-1.5 | Modification of the default gain setting may result to system instability and is not recommended. DON’T CHANGE THIS VALUE. |
| tau\_f | Friction compensator gain of torque controller. | 0.7 | 0-0.85 | Modification of the default gain setting may result to system instability and is not recommended. DON’T CHANGE THIS VALUE. |
| tau\_d | Damping gain of torque controller. | 0.007 | 0.005-0.008 | Modification of the default gain setting may result to system instability and is not recommended. DON’T CHANGE THIS VALUE. |

OUTPUT:

**motor\_started:** boolean value if the start motor command was received using ACK/NACK feedback coming from the server.

## 2.6 stop\_motors

Stop all motors.

**motor\_stopped=client.stop\_motors();**

OUTPUT:

**motor\_stopped:** boolean value if the stop motor command was received using ACK/NACK feedback coming from the server.

## 2.7 pdo\_aux\_cmd (release brake)

Release brakes for all motors specified by ID.

**brake\_status= client.pdo\_aux\_cmd(brake\_cmds);**

INPUT:

**brake\_cmds:** vector of tuples having these fields: **motor ID, release brake command.**

OUTPUT:

**brake\_status:** boolean value if the brake release command was received using ACK/NACK feedback coming from the server.

## 2.8 pdo\_aux\_cmd (engage brake)

Engage brakes for all motors specified by ID.

**brake\_status= client.pdo\_aux\_cmd(brake\_cmds);**

INPUT:

**brake\_cmds:** vector of tuples having these fields: **motor ID, engage brake command.**

OUTPUT:

**brake\_status:** boolean value if the brake engage command was received using ACK/NACK feedback coming from the server.

**Note: The brake has to be engaged when the motor velocity is less than 0.02 rad/s.**

## 2.9 pdo\_aux\_cmd (LED ON)

Switch ON the LEDs for all motors specified by ID.

**led\_status= client.pdo\_aux\_cmd(led\_cmds);**

INPUT:

**led\_cmds:** vector of tuples having these fields: **motor ID, LED ON command.**

OUTPUT:

**led\_status:** boolean value if the LED ON command was received using ACK/NACK feedback coming from the server.

## 2.10 pdo\_aux\_cmd (LED OFF)

Switch OFF the LEDs for all motors specified by ID.

**led\_status= client.pdo\_aux\_cmd(led\_cmds);**

INPUT:

**led\_cmds:** vector of tuples having these fields: **motor ID, LED OFF command.**

OUTPUT:

**led\_status:** boolean value if the LED OFF command was received using ACK/NACK feedback coming from the server.

## 2.11 feed\_motors

Send references to the motors.

**client.feed\_motors(std::make\_tuple(motor\_ref\_flags, motors\_ref));**

INPUT:

**motors\_ref:**  vector of tuples having these fields:

1. motor ID.
2. control mode 3B (position) or D4 (impedance mode).
3. position reference (rad).
4. velocity reference (rad/s).
5. torque reference (Nm).
6. position gains (Kp, Ki, Kd, G1, G2) or impedance gains (Kp, Kd, tau\_p, tau\_f, tau\_d) based on the control mode.
7. op,idx and aux (Reserved for internal scope).

**motor\_ref\_flags:**  FLAG\_MULTI\_REF, FLAG\_LAST\_REF (NOT USED).

## 2.12 stop\_client

Stop UDP client.

**client.stop\_client()**

## 2.13 get\_motors\_status

Get motor status of all motors useful for telemetry.

**motors\_status\_map= client.get\_motors\_status();**

**OUTPUT:**

**motors\_status\_map**: motors status map having like key, motor id, and value motor\_status with these follow fields:

**link position:** link side position reading of the motor.

**motor position:** motor side position reading of the motor.

**link velocity:** link side velocity reading of the motor.

**motor velocity:** motor side velocity reading of the motor.

**torque:** torque sensor reading of the motor.

**motor temp:** motor temperature.

**board temp:** board temperature.

**fault:** fault of the motor.

**rtt:** round trip time.

**op\_idx\_ack:** operational id acknowledge.

**aux:** auxiliary.

**cmd\_aux\_sts:**  auxiliary command status brake and LED status).

**Note: Fault value is an uint16 value having a specific bit mask to identify the fault:**

| **m3\_rxpdo\_pos\_ref** | **0** |
| --- | --- |
| **m3\_rxpdo\_vel\_ref** | **1** |
| **m3\_rxpdo\_tor\_ref** | **2** |
| **m3\_fault\_hardware** | **3** |
| **m3\_params\_out\_of\_range** | **4** |
| **m3\_torque\_array\_not\_loaded** | **5** |
| **m3\_torque\_read\_error** | **6** |
| **m3\_out\_of\_limits** | **5** |
| **m3\_link\_enc\_error** | **8** |
| **m3\_defl\_enc\_error** | **9** |
| **m3\_temperature\_warning** | **10** |
| **m3\_temperature\_error** | **11** |
| **c28\_motor\_enc\_error** | **12** |
| **c28\_v\_batt\_read\_fault** | **13** |
| **c28\_enter\_sand\_box** | **14** |
| **m3\_safaty\_override** | **15** |

**Note: This function uses the mutex mechanism explained before (1.) to synchronize the UDP client with the main process.**

## 2.14 get\_pow\_status

Get power board status for all power boards useful for telemetry.

**pow\_status\_map= client.get\_pow\_status();**

**OUTPUT:**

**pow\_status\_map**: power board status map having like key, power board id, and value power\_board\_status with these follow fields:

**v\_batt:** voltage battery reading.

**v\_load:** voltage load reading.

**i\_load:** current load reading.

**temp\_pcb:** pcb temperature reading.

**temp\_heatsink:** heat sink temperature reading.

**temp\_batt:** battery temperature reading.

**Note: This function uses the mutex mechanism explained before (1.) to synchronize the UDP client with the main process.**

## 2.15 get\_ft6\_status

Get force/torque sensor status for all force/torque sensor boards useful for telemetry.

**ft6\_status\_map= client.get\_ft6\_status();**

**OUTPUT:**

**ft\_status\_map**: force/torque status map having like key, force/torque board id, and value ft6\_status with these follow fields:

**force\_x:** force value in x.

**force\_y:** force value in y.

**force\_z:** force value in z.

**torque\_x:** force value in x.

**torque\_y:** force value in y.

**torque\_z:** force value in z.

**Note: This function uses the mutex mechanism explained before (1.) to synchronize the UDP client with the main process.**

# 3 Client initialization

This part is very important to use and add the UDP client inside a specific process or thread (of a process):

1. Create a logger.
2. Create a UDP client class with hostname and port.
3. Setup the UDP periodic time.
4. Start a connection.
5. Run the client thread alongside the main process to start the event handler system.

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\* START UDP CLIENT \*\*\*\*\*\*\*\*\*\*\*\*\*\*\* //

createLogger("console","client");

Client client(ec\_client\_cfg.host\_name\_s,ec\_client\_cfg.host\_port);

auto UDP\_period\_ms\_time=milliseconds(ec\_client\_cfg.UDP\_period\_ms);

client.set\_period(UDP\_period\_ms\_time);

client.connect();

// Run asio thread alongside main thread

std::thread t1{[&]{client.run();}};

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\* START UDP CLIENT \*\*\*\*\*\*\*\*\*\*\*\*\*\*\* //

# 4 UDP Communication quality disruption check

Two safety controls were developed to verify UDP communication in terms of losing communication and UDP quality motor reference check. The rolling\_mean functions in the boost library are used to evaluate the samples in a specific window (Default value 50 with the possibility to modify the rolling\_window\_size editing this file: .ecat\_master/configs/udp\_srv.yaml). The mean in that window is controlled and if it’s higher and equal to 225 Hz the recovery action is activated, engaging the brakes and stopping the controllers for all motors.

Working Frequency Range:

**- 1KHz <= freq\_choosen < 500 Hz (1ms <= freq\_choosen < 2ms): avoid these frequencies due the UDP protocol.**

**- 500Hz <= freq\_choosen < 225 Hz (2ms <= freq\_choosen < 4.44ms): safe range.**

**- freq\_choosen >= 225.0 Hz (freq\_choosen >= 4.44ms): safety UDP quality check detection.**

The other safety check is related to losing communication. If the UDP server finds that the difference between the last motors reference and actual UDP server periodicity is greater than 10ms the recovery action is activated, engaging the brakes and stopping the controllers for all motors.

links:

<https://www.boost.org/doc/libs/1_81_0/doc/html/accumulators/user_s_guide.html>

<https://www.boost.org/doc/libs/1_81_0/doc/html/accumulators/user_s_guide.html#accumulators.user_s_guide.the_statistical_accumulators_library.mean>

<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>

# 

# 5 Safety flow for testing

Remind to start the system following this flow:

1. start the EtherCAT Master (repl).
2. start udp\_server.
3. start Mirror client.

**Note: if the EtherCAT Master dies or is stopped using the emergency button, stop ALL processes and restart the system using the flow explained before. If the udp server dies restart it but remind to STOP the Mirror Client.**

# 6 Logging Information

**EtherCAT Master log:**

repl -f .ecat\_master/configs/zipc\_config.yaml l—-----------> **Start REPL Process.**

Build timestamp 2022-06-15T12:06:31 l—-----------> **Build timestamp of the REPL application.**

Git short hash f3e3f8d l—-----------> **Github hash of the REPL repository.**

Using .ecat\_master/configs/zipc\_config.yaml as config file l—-----------> **Configuration file used.**

.... open /proc/xenomai/registry/rtipc/xddp/repl\_in

.... open /proc/xenomai/registry/rtipc/xddp/repl\_out l—-----------> **Opening of REPL software infrastructure.**

.... open /proc/xenomai/registry/rtipc/xddp/repl\_info

.... open /proc/xenomai/registry/rtipc/xddp/emergency

th\_init run\_mode l—-----------> **EthertCAT Communication in run mode (OPERATIONAL).**

[ECat\_master] Using rteth0

[ZMQ\_rep] bind to ipc:///tmp/ecat\_master:5555, zmq\_rcvtimeo\_ms 500

[ZMQ\_rep] Opening xddp /proc/xenomai/registry/rtipc/xddp/repl\_in

[ZMQ\_rep] Opening xddp /proc/xenomai/registry/rtipc/xddp/repl\_out

non\_periodic\_thread ZMQ\_rep, period 1 us

non\_periodic\_thread ZMQ\_rep : start looping ...

[ECat\_master] POWER ON slaves.

[**ECat\_master] 19 EtherCAT slaves identified.**l—-----------> **EthertCAT Slaves identified.**

l—-----------> **Starting of factory phase where all slave are classified and checked.**

[ECat\_master] Request 0x02 state for all slaves

Slave 1 State=0x04 StatusCode=0x0000 : No error... reach State=0x02

Slave 18 State=0x04 StatusCode=0x0000 : No error... reach State=0x02

Slave 19 State=0x04 StatusCode=0x0000 : No error... reach State=0x02

Man: 00000298 ID: 00000032 Rev: 00000001

Man: 00000298 ID: 00000015 Rev: 00000001

Man: 00000298 ID: 00000015 Rev: 00000001

l—-----------> **Factory of a MOTOR.**

>> factory 298 id 0x0015 Cent\_Motor : conf\_addr 1013 pos 19 rev 1 alias 0

support DC 1

topology 1 port act 2

Ibytes 28 Obytes 28

Using config CentAcESC\_X

[PDO\_aux] pdo\_auxes\_map size is 0

.... open /proc/xenomai/registry/rtipc/xddp/NoNe@Motor\_id\_36

.... open /proc/xenomai/registry/rtipc/iddp/NoNe@Motor\_id\_36\_rx\_pdo

init\_transport protoType 0 pipe\_port -1

HW\_config\_high 0x5300 hw\_brake 1

[PDO\_wr\_aux] master\_cmd\_control 0x8003:3

[PDO\_rd\_aux] master\_cmd\_status 0x8002:30

[PDO\_aux] pdo\_auxes\_map size is 2

Ctor pos 19 id 36

Joint serial# 530 Joint robot id 36 HW conf 0x5417

min pos -2.791593 max pos 2.788407

fw\_ver m3 220606xx c28 21122115

>>find\_root id 201

**found 1 powF28M36s**

**found 18 motors**

**found 0 fts** l—-----------> **please verify always the number of slaves that have to be the same with**

**the count of slave classified (motors, power board and etc…).**

**found 0 ftsMsp**

**found 0 foot\_10x5**

**found 0 skin\_8x3**

**found 0 tests**

[ECat\_master] Configure DC 1000000 1000000000

[ECat\_master] Request 0x04 state for all slaves

[ECat\_master] pdo warm up ...

[ECat\_master] Request 0x08 state for all slaves

[ECat\_master] Request 0x08 state for all slaves

[ECat\_master] Calculated workcounter 57

[ECat\_master] ec\_DCtime 3348282639551188440

[ECat\_master] o: 514 i: 526

[ECat\_master] Start ecat\_thread 1000000 ns

non\_periodic\_thread EC\_boards\_repl, period 1 us

non\_periodic\_thread EC\_boards\_repl : start looping …

l—-----------> **Factory COMPLETED.**

[0MQ Pub] Opening xddp\_socket /proc/xenomai/registry/rtipc/xddp/NoNe@Motor\_id\_35

[0MQ Pub] publisher bind to ipc:///tmp/ecat\_master:9036

[0MQ Pub] publishing with id NoNe@Motor\_id\_36

[0MQ Pub] Opening xddp\_socket /proc/xenomai/registry/rtipc/xddp/NoNe@Motor\_id\_36

[0MQ Pub] publisher bind to ipc:///tmp/ecat\_master:9201

[0MQ Pub] publishing with id NoNe@PowBoard\_id\_201

[0MQ Pub] Opening xddp\_socket /proc/xenomai/registry/rtipc/xddp/NoNe@PowBoard\_id\_201

non\_periodic\_thread ZMQ\_Pub\_thread, period 1 us

non\_periodic\_thread ZMQ\_Pub\_thread : start looping ...

l—-----------> **REPL Ready.**

**repl\_loop : handle CTRL\_CMD** l—-----------> **i**f this line is present in the log this means that a control command (start or stop motors) has arrived from the UDP server.

**repl\_loop : handle PDO\_AUX\_CMD** l—-----------> if this line is present in the log this means that a PDO auxiliary command (release and engage the brake) has arrived from the UDP server.

**repl\_loop : handle MOTOR\_PDO\_CMD** l—-----------> if this line is present in the log this means that a Motor PDO command (motor references) has arrived from the UDP server.

[ECat\_master] WARN: wkc 3 != 57 expectedWKC

ESC 1 : topology changed 4 --> 1

ESC 1 : active ports changed f --> 1

ESC 2 : fail read ECT\_REG\_DLSTAT

ESC 3 : fail read ECT\_REG\_DLSTAT

ESC 4 : fail read ECT\_REG\_DLSTAT

ESC 5 : fail read ECT\_REG\_DLSTAT

ESC 6 : fail read ECT\_REG\_DLSTAT

ESC 7 : fail read ECT\_REG\_DLSTAT

ESC 8 : fail read ECT\_REG\_DLSTAT

ESC 9 : fail read ECT\_REG\_DLSTAT l—-----------> **EtherCAT communication CUTTING OFF**

ESC 10 : fail read ECT\_REG\_DLSTAT l—-----------> **via** **emergency button**

ESC 11 : fail read ECT\_REG\_DLSTAT

ESC 12 : fail read ECT\_REG\_DLSTAT

ESC 13 : fail read ECT\_REG\_DLSTAT

ESC 14 : fail read ECT\_REG\_DLSTAT

ESC 15 : fail read ECT\_REG\_DLSTAT

ESC 16 : fail read ECT\_REG\_DLSTAT

ESC 17 : fail read ECT\_REG\_DLSTAT

ESC 18 : fail read ECT\_REG\_DLSTAT

ESC 19 : fail read ECT\_REG\_DLSTAT

[ECat\_master] WARN: wkc 3 != 57 expectedWKC l—-----------> **Note: if the WKC counter is more than 3 this mean EtherCAT communication error.**

ESC 1 : topology changed 4 --> 1

**UDP server log:**

udp\_server

[log hour] [server] [info] Server Started FILL IN THE BLANK

[log hour][server] [info] dump /home/embedded/.ecat\_master/configs/udp\_srv.yaml

[log hour][server] [info] -- ecat\_host : \_NOT\_USED\_

[log hour][server] [info] -- ecat\_uri : ipc:///tmp/ecat\_master l—-----------> **UDP server setup.**

[log hour] [server] [info] -- period\_ms : 10

[log hour][server] [info] -- pub\_base\_port : 9000

[log hour][server] [info] -- repl\_port : 5555

[log hour] [server] [info] REPL REQ : 0 l—-----------> **Get slave information.**

[0MQ Req] connect to ipc:///tmp/ecat\_master:5555

11 15

12 15

13 15

14 15

15 15

16 15

21 15 l—-----------> **Slave ID and type.**

22 15

23 15

24 15

25 15

26 15

31 15

32 15

33 15

34 15

35 15

36 15

201 32

[log hour][server] [info] send slave info reply l—-----------> **Get slave information.**

[log hour][server] [info] msg 23: type: RX\_XT\_MOTOR

header {

stamp {

sec: 1165

nsec: 325320029

}

}

motor\_xt\_rx\_pdo {

link\_pos: 2.47543168

motor\_pos: 2.47545028 l—-----------> **READ PDO of the motors.**

link\_vel: 0

motor\_vel: 0

torque: -38.4348145

temperature: 8229

fault: 0

rtt: 948

op\_idx\_ack: 30

aux: 2.80259693e-45

motor\_temp: 32

board\_temp: 37

pos\_ref: 2.47545028

vel\_ref: 0

tor\_ref: 0

cmd\_aux\_sts: 2

}

As shown before, when REPL REQ is logged means that the UDP client has done a request that can be classified:

[log hour][server] [server] [info] REPL REQ : 0 l—-----------> **Get slave information.**

[log hour][server] [server] [info] REPL REQ : 1 l—-----------> **Start Motors.**

[log hour][server] [server] [info] REPL REQ : 2 l—-----------> **Stop Motors.**

[log hour][server] [server] [info] REPL REQ : 5 l—-----------> **Motors references.**

[log hour][server] [server] [info] REPL REQ : 6 l—-----------> **Release or Engage the brakes.**

[log hour][server] [server] [info] REPL REQ : 9 l—-----------> **SDO command.**