Scuola universitaria professionale
della Svizzera italiana

SUPSI

micro ROS

Studente

Jacopo Pagnoncelli

Relatore

Roberto Bucher

Correlatore

Ivan Furlan

Corso di laurea

Elettronica

Modulo

Progetto di laurea

Anno accademico

2020-2021

Indice

Elenco delle figure

1. Abstract

Il progetto si prefigge l'obbiettivo di interfacciare il ROS2 installato su un sistema UBUNTU con il micro ROS presente su un microcontrollore tramite CAN bus.

Per potere connettere i due sistemi viene utilizzato il DDS per creare una connessione tra un client presente sul microcontrollore e un agent sul NUC. In questo modo s riesce ad ottenere la interoperabilità fra i due sistemi.

Si è riuscito a creare un sistema che si collega sfruttando l'ID del CAN bus come indentificativo di ogni device.

2. Introduzione

Negli ultimi anni il mercato dei robot è sempre più crescente grazie alla miniaturizzazione degli IC e alla potenza di calcolo sempre più crescente. L'incremento di funzioni e periferiche disponibili richiede un aumento proporzionale di calcolatori presenti su un singolo robot. La gestione tramite software del sistema e delle singole parti è aumentato in modo esponenziale col passare degli anni fino a diventare insostenibile per una singola azienda. Per sopperire a questo problema è stato creato il Robot Operating System(ROS), un framework utilizzato per la gestione ad un alto livello di astrazione di cluster di calcolatori. Il ROS si basa sul concetto di nodi, delle entità in grado di elaborare e scambiare dati con altri nodi distaccandosi dai componenti hardware che permettono il funzionamento a livello fisico. Il progetto è open-source e questo permette di avere a disposizione numerosi pacchetti(applicazioni per ROS)[] che permettono di abbattere i tempi di sviluppo di un sofware realizzato tramite ROS. Nell'ultimo periodo è stato introdotto il ROS2, il successore del ROS.

La Hilti AG ha recentemente introdotto sul mercato the first Jobsite robot for ceiling drilling application. Nel progetto è stato utilizzato il ROS per una questione di stabilità del framework ma un porting al ROS2 è una opzione possibile grazie ai continui aggiornamenti che lo hanno reso sempre più stabile.
L'obiettivo del progetto è quello di creare un sistema che sfrutti il ROS2 su una macchina con sistema operativo ubuntu e due microcontrollori STM32F4 che utilizzano il micro-ROS. Lo schema logico del progetto è mostrato in fig. 2.1

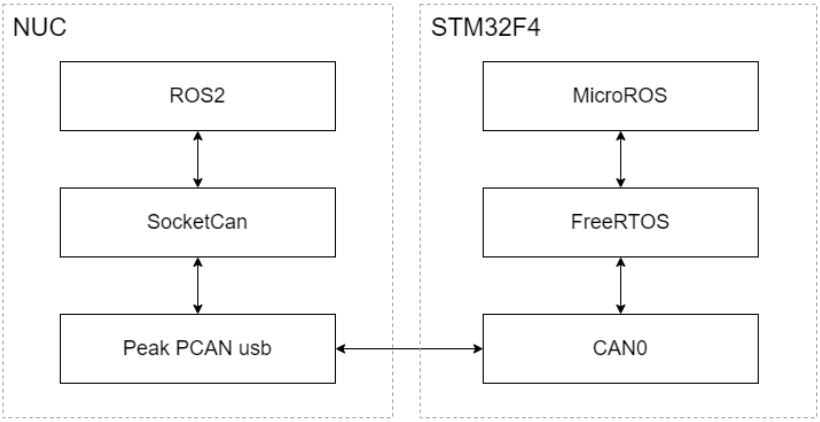


Figura 2.1: Schema logico progetto

I componenti hardware sono:

- Intel NUC
- 2x STM32 nucleo-64
- PEAK PCAN-USB
- 2x LC Technology TJA1050 CAN Transceiver Module

Il canale di comunicazione scelto è il Controller Area Network(CAN). Questo protocollo permette di avere una comunicazione molto robusta grazie alla alta reiettività ai disturbi e alle 5 differenti modalità per detettare un errore. Il protocollo è gestito da parte dei microcontrollori tramite un modulo transceiver che converte il segnale differenziale tipico del CAN bus in un segnale single-ended gestito da una periferica dedicata. Per quanto riguarda il NUC il PEAK PCAN-USB collega il CAN bus ad una porta usb. In seguito viene creato un socket per potere utilizzare il protocollo come se fosse un network generico consentendo l'utilizzo delle funzioni di read e write. Nel ROS2 in seguito viene creato un nodo che sfrutta il meccanismo del socket-can per potere ricevere ed inviare direttamente messaggi tramite il CAN bus. Nei microcontrollori invece viene utilizzato il sistema operativo FreeRTOS per supportare il micro-ROS che è un ramo del ROS2 indirizzato per i microcalcolatori con risorse limitate.

3. CAN bus

Il CAN bus viene utilizzato nel progetto come protocollo di comunicazione per inviare messaggi tra i dispositivi utilizzati. Il CAN bus è stato scelto per la sua affidabilità e per la possibilità di ampliare i dispositivi connessi al bus. In questo capitolo verranno spiegate le parti fondamentali del protocollo per poi analizzare l'implementazione nel progetto.

3.1 Protocollo CAN

Il Controller Area Network(CAN) è uno standard di comunicazione sviluppato dalla Rober Bosch GmbH negli anni 80. Inizialmente è stato usato in ambito automotive per l'alta reiettività ai disturbi. Il protocollo è di tipo broadcast ovvero che ogni messaggio inviato da un nodo che compone il sistema è ricevuto da tutti gli altri ed ogni nodo è in grado di inviare un messaggio. La modalità di trasmissione differenziale permette di utilizzare il protocollo in ambienti fortemente disturbati a livello elettromagnetico. La velocità di trasmissione massima è di 1 Mbit/s fino a 40 m.

Esistono 4 diversi tipi di frame che sono:

- Data frame
- Remote frame
- Error frame
- Overload frame

Data frame: Il formato del frame utilizzato per trasmettere un dato è mostrato in fig. 3.1. Il campo arbitration rappresenta l'ID del messaggio. Infatti, a differenza della maggioranza dei protocolli di comunicazione, l'ID è riferito al messaggio e non al dispositivo che lo invia.

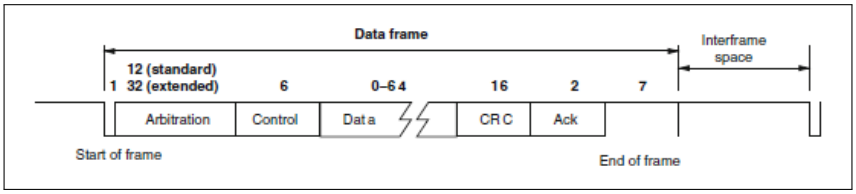


Figura 3.1: CAN data frame

Il numero di bit dell'ID definisce anche il formato del frame che sono:

- Standard frame (ID da 11 bit)
- Extended frame (ID da 29 bit)

Le due versioni vengono definite come CAN 2.0A e 2.0B. I due formati di frame vengono distinti da un bit nel control field. La versione 2.0B supporta anche messaggi di formato

2.0B.In questo progetto viene utilizzato lo standard frame visto il già alto numero di ID unici utilizzabili(2031).

Il data field contiene il dato da inviare e può variare da 0 a 8 byte. Inviare un payload di 0 byte può essere utile per verificare se tutti i nodi presenti nel sistema ricevono il messaggio. Il CRC field viene utilizzato per detettare se il messaggio ricevuto e quello inviato risultano uguali o se qualche bit nella trasmissione è cambiato. L'acknowledgement slot è composto da un bit che serve per verificare che tutti i nodi abbiano ricevuto il messaggio e il secondo bit è un delimitatore per segnare la fine del frame.

Remote frame: serve per richiedere un particolare dato. La struttura del frame è identica a quella di un data frame senza payload. Per differenziarli il bit RTR nell'arbitration field è recessivo. Un altro nodo che riceve questo messaggio potrebbe potrebbe inviare un data frame con lo stesso ID e con il data field contente il dato richiesto.

Error frame: è un frame che viola la lunghezza standard di un frame. Serve per notificare che un nodo ha trovato un errore nel messaggio inviato. Anche se solo un nodo invia questo frame il frame dovrà essere inviato nuovamente.

Overload frame: serve per notificare che un nodo non riesce a stare al passo con la velocità di trasmissione. Questo frame non è più utilizzato.

Bus arbitration: il CAN bus è un protocollo multicast. Questo comporta che due nodi possono tentare di trasmettere un frame nello stesso momento. Il messaggio con l'ID più basso vincerà la corsa e verrà trasmesso. Il motivo è che il protocollo CAN ha un sistema di bit dominante. Il valore logico 0 è dominante rispetto al valore1, che viene definito recessivo. Si può quindi considerare il bus come una AND. Un altro nodo che sta partecipando alla corsa vedendo che il messaggio trasmesso sul canale non è identico a quello che sta trasmettendo entrerà in modalità listener e aspetterà la fine del messaggio per provare ad inviare nuovamente il proprio frame. Un esempio è mostrato in fig. 3.2.

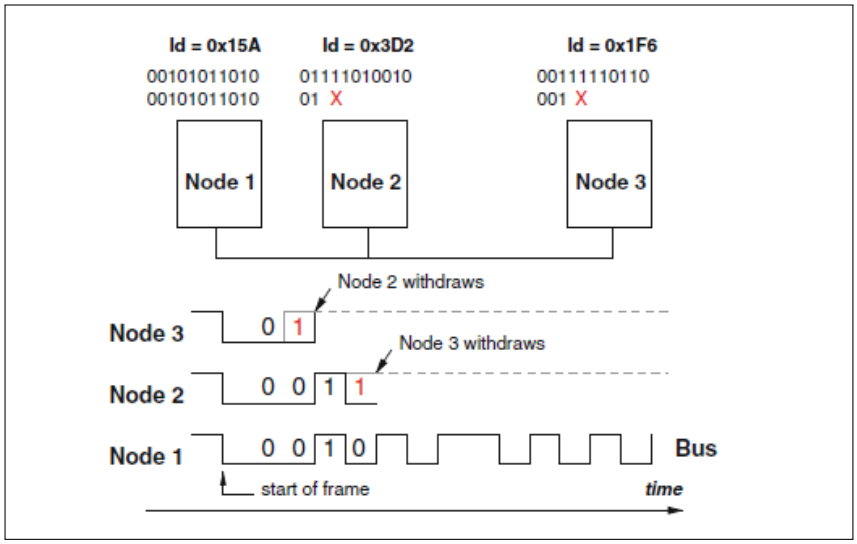


Figura 3.2: Esempio di arbitrazione di un messaggio

Error checking: una delle caratteristiche che contraddistingue il CAN bus è l'estrema capacità di trovare errori di comunicazione. Se un singolo nodo rileva un errore in un messaggio questo viene scartato. I metodi per detettare un errore sono:

- Bit monitorig
- Bit stuffing
- Frame check
- Acknowledgement check
- Cyclic redundancy check

Ogni nodo che sta trasmettendo un messaggio lo controlla per verificare che il contenuto non si modifichi. Questo meccanismo è definito bit monitoring.

Il bit stuffing è una convenzione per cui ad ogni 5 bit trasmessi dello stesso livello venga trasmesso un bit di valore logico opposto. Questo permette ai nodi che sono in ascolto di detettare un errore nel caso in cui vengano trasmessi 6 bit dello stesso valore.

Il frame check consiste nel controllare alcuni bit di un frame che rimangono fissi in tutti i messaggi per trovare un errore di trasmissione.

L'acknowledgement check viene eseguito nell'ACK field. Un nodo che riceve un messaggio imporrà durante l'ACK field un bit dominante. Il nodo che trasmette il messaggio si aspetta di leggere questo bit dominante. Se ciò non accade, verrà segnalato un acknowledgement error.

Il cyclic redudancy check è eseguito sul campo CRC che è usato se ci sono errori di trasmissione del messaggio.

Fault confinement: Ogni nodo ha due error counter: uno per la trasmissione e uno per la ricezione. In base ad ogni errore e dove questo viene rilevato i counter possono essere aumentati o diminuiti. Se un counter aumenta oltre un certo valore alcune funzionalità possono essere disattivate fino al confinamento del nodo stesso. Se successivamente il nodo riuscirà a ricevere o trasmettere potrà essere riammesso nel bus.

Bit timing: Ogni singolo bit inviato tramite CAN bus è diviso in 4 segmenti visibili in fig. 3.3. Ogni segmento è composto da uno o più time quanta. Questa è l'unità di tempo elementare del protocollo. I 4 segmenti sono definiti come:

- Syncronization segment
- Propagation segment
- Phase segment 1
- Phase segment 2

Il syncronization segment è sempre di 1 quanto e serve per sincronizzare il clock dei nodi.

Il Propagation segment serve per compensare il propagation delay lungo il bus. Varia in base alla lunghezza del canale.

I due phase segments vengono utilizzati per compensare i ritardi di fase. Tra questi due segmenti avviene il sample point ovvero il momento in cui viene campionato il valore del bit.

Questi valori devono essere concordi per tutti i nodi per ottenere la sincronizzazione di tutti gli attori sul bus.

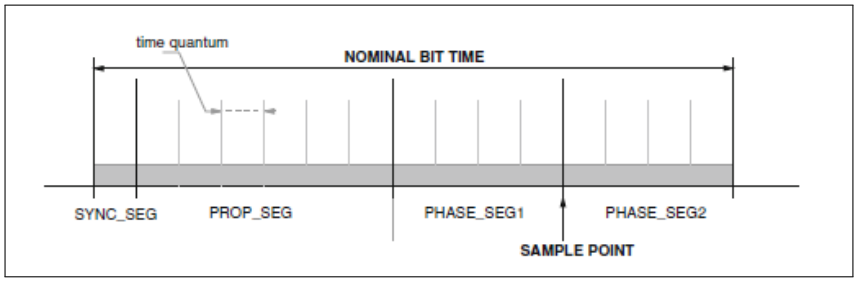


Figura 3.3: CAN bit frame

3.2 CAN su STM32F446RE

Il protocollo CAN bus nel microcontrollore STM32F446RE è supportato sia in versione 2.0A che 2.0B. Sono presenti 2 periferiche in grado di gestire il protocollo per aumentare il grado di robustezza. La maggior parte delle operazioni è eseguita tramite hardware per permettere la gestione delle periferiche anche con la CPU con poco tempo di calcolo a disposizione.

3.2.1 Struttura generale

Nell'STM32F446RE esistono due periferiche dedicate al CAN bus. Esse però non sono in grado di interfacciarsi direttamente al bus ma hanno bisogno di un CAN transceiver (fig. 6.1).

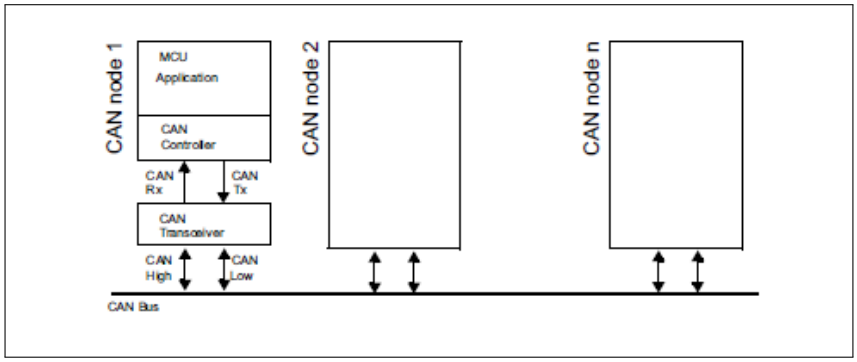


Figura 3.4: Inferfacciamento MCU con CAN newtork

Il transceiver è necessario per convertire il segnale differenziale in uno single-ended e poter indirizzare il segnale nel pin corretto in base alla ricezione o invio del dato. Il transceiver scelto è prodotto dalla LC technology ed utilizza il TJA1050 che consente

ad una sola periferica di interfacciarsi. Le due periferiche presenti sul microcontrollore vengono definite master e slave. Questo nome è riferito alla gestione dei 512 byte di SRAM. Infatti il master è in grado di accedere alla memoria mentre l'accesso per lo slave è gestito dal master. Questo comporta che il master dovrà essere attivo per permettere allo slave di utilizzare tutte le funzionalità. Una implementazione spesso usata è quella di usare lo slave come rindondanza rispetto al master. In questo modo viene aumentata l'affidabilità del sistema.

Tranmit: ogni periferica ha 3 mailboxes utilizzate per inviare un CAN frame. Una mailbox viene inizializzata nello stato empty, quando al suo interno viene scritto il frame entra in stato pending. Se è la mailbox con la priorità più alta il dato al suo interno verrà inviato. La priorità è definita in primo luogo dal ID del messaggio con l'ID 0x00 che ha la priorità maggiore. Se gli ID dei frame contenuti in due mailboxes è identico la mailbox con il valore minore verrà schedulata prima.

Receive: I messaggi in arrivo vengono gestiti totalmente via hardware tramite due FIFO per periferica. Ogni FIFO è in grado di contenere fino a 3 messaggi. Per poi potere essere scritto nella SRAM dovrà coincidere con almeno un filtro.

Filters: I filtri sono una componente fondamentale del protocollo visto che tutti i messaggi devono essere ricevuti per il controllo di errori. Il filtraggio avviene in base all'ID del frame in arrivo. Vengono messi a disposizione 28 banchi per i filtri. Se viene utilizzata una sola periferica sono disponibili solo i primi 14 banchi. Le due modalità che si possono utilizzare sono:

- Identifier list
- Identifier mask

Nella prima modalità viene semplicemente confrontato il frame in ricezione con l'ID inserito nel filtro. Se i due valori coincidono il messaggio viene salvato nella SRAM.

$$(\text{DATA} \ \& \ \text{MASK}) = \text{ID}$$

Con DATA che corrisponde all'ID frame e MASK ad una maschera inserita via software. ID e MASK sono da composti da due campi da 16 bit e per questo si possono utilizzare due scale per filtrare ID da 11 o 29 bit. Nel caso di un ID da 29 bit si può utilizzare solo la scala da 32bit unendo i due campi delle variabili mentre per ID da 11 bit si può utilizzare anche la scala da 16 bit per avere due filtri in un singolo banco(fig. 3.5).

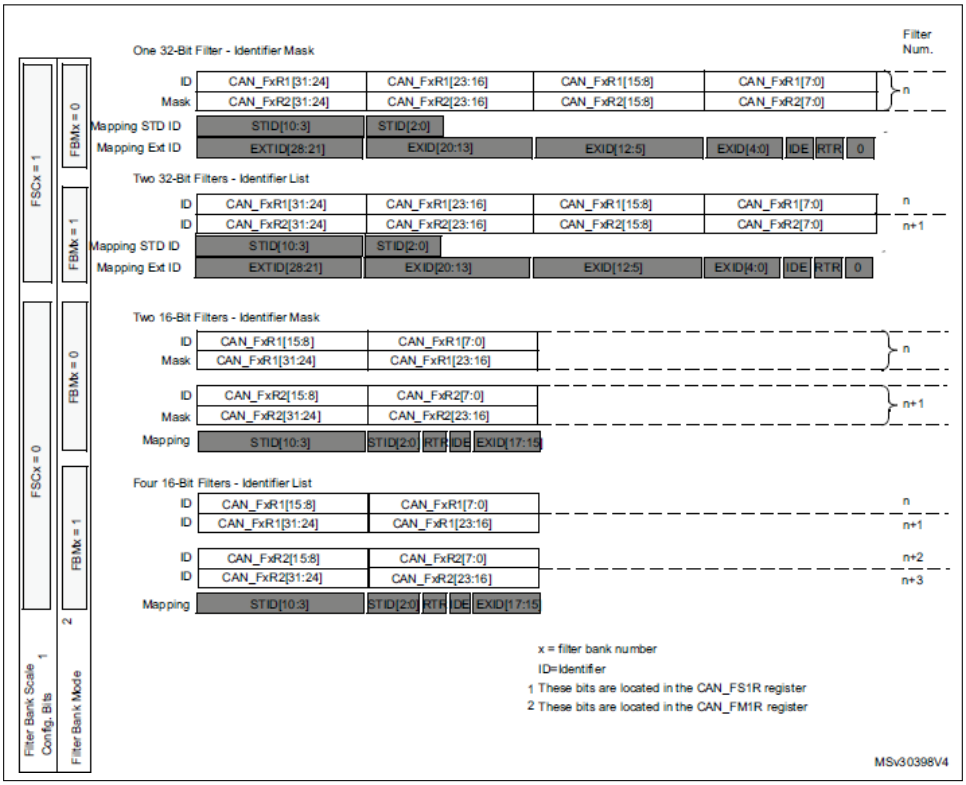


Figura 3.5: Configurazione filtri STM

3.2.2 Implementazione

Per potere testare il CAN bus si utilizza il programma STM32CubeIDE. Inizialmente si crea un progetto utilizzando il microcontrollore STM32F446RE. Per attivare la periferica CAN bisogna:

1. Aprire il file .ioc sotto Connectivity->CAN1 attivare la periferica(Master Mode).
2. Sotto Parameter Settings impostare i Bit Timings Parameters come in tab. 3.1 per ottenere una velocità di trasmissione di 500 Kbit/s.
3. Attivare l'interrupt CAN1 RX0 interrupt che si trova in NVIC Setting.

Frequency	16 Mhz
Prescaler	2
Time quantum	125 ns
Time Quanta Segment 1	13 Times
Time Quanta Segment 2	2 Times
ReSyncronization Jump Width	1 Time

Tabella 3.1: CAN1 segment configuration

La configurazione software del CAN bus viene effettuata attraverso varie funzioni HAL (fig. 6.1). A seguito della configuraione si potrà usare la funzione di transmit e la receive callback sarà inizializzata.

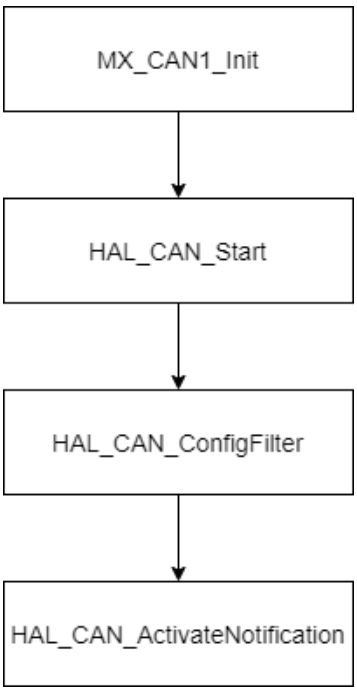


Figura 3.6: Inizializzazione periferica CAN

Impostazione filtro: l'impostazione di un filtro è obbligatoria. L'assenza di un filtro provoca il blocco di tutti i messaggi e quindi non ci saranno interrupt. La funzione per impostare un filtro è HAL_CAN_ConfigFilter. La struttura CAN_FilterTypeDef definita nel file stm32f4xx_hal_can.h contiene le caratteristiche di un filtro. Nell'esempio viene mostrato un filtro che accetta tutti i messaggi.

```
CAN_FilterTypeDef CAN_Filters;

CAN_Filters.FilterIdHigh = 0x0000;
CAN_Filters.FilterIdLow = 0x0000;
CAN_Filters.FilterMaskIdHigh = 0x0000;
CAN_Filters.FilterMaskIdLow = 0x0000;
CAN_Filters.FilterFIFOAssignment = CAN_FilterFIFO0;
CAN_Filters.FilterBank = 0;
CAN_Filters.FilterMode = CAN_FILTERMODE_IDMASK;
CAN_Filters.FilterScale = CAN_FILTERSCALE_32BIT;
CAN_Filters.FilterActivation = ENABLE;
CAN_Filters.SlaveStartFilterBank = 0;
```

Transmit: i messaggi vengono inviati tramite la funzione HAL_CAN_AddTxMessage. Gli argomenti della funzione sono un puntatore alla struttura che contiene la configurazione della periferica CAN, un puntatore alla struttura che contiene le informazioni relative al frame da inviare, un vettore contenente il payload e una variabile in cui verrà inserita la mailbox usata per salvare il messaggio. La struttura CAN_TxHeaderTypeDef contiene i campi:

- StdId : specifica l'ID a 11 bit del frame
- ExtId : specifica l'ID a 29 bit del frame
- IDE : specifica se il frame è 2.0A o 2.0B

- RTR : specifica se inviare un DATA o un REMOTE frame
- DLC : specifica la lunghezza in byte del payload
- TransmitGlobalTime : specifica se usare parte del payload per indicare il tempo di invio del messaggio

```
HAL_StatusTypeDef
HAL_CAN_AddTxMessage(
  CAN_HandleTypeDef *hcan,
  CAN_TxHeaderTypeDef *pHeader,
  uint8_t aData[],
  uint32_t *pTxMailbox)
```

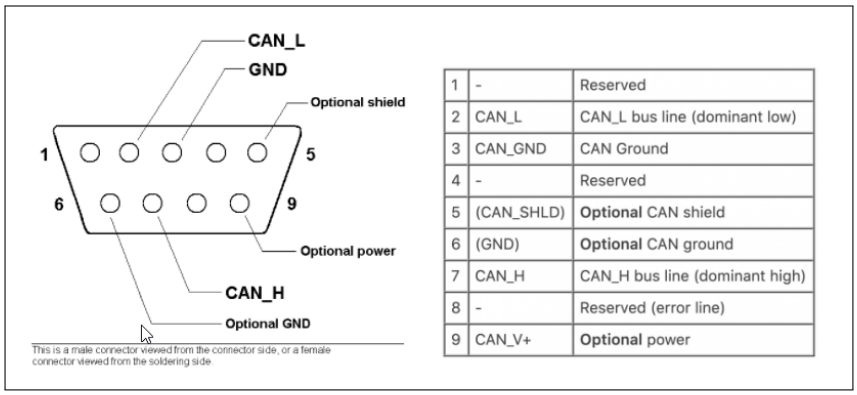
Receive: I messaggi vengono ricevuti tramite interrupt. Ogni volta che un messaggio arriva e viene memorizzato in una FIFO viene chiamata la funzione HAL_CAN_RxFifo0MsgPendingCallback. Dentro questa callback viene chiamata la funzione HAL_CAN_GetRxMessage per potere memorizzare il frame. Gli argomenti sono similari alla funzione per la trasmissione fatta eccezione per i campi Timestamp e FilterMatchIndex della struttura CAN_RxHeaderTypeDef. Un campo serve per memorizzare il momento in cui è arrivato il frame mentre il secondo indica il numero del filtro che ha permesso la memorizzazione nella SRAM.

3.3 CAN su Ubuntu

Il microcontrollore comunca tramite il CAN bus con un NUC con sistema operativo Ubuntu via usb. Viene utilizzato il PEAK PCAN-USB per eseguire questo cambio di interfaccia. Poi tramite software verrà creato il socket-CAN, un layer che permette l'utilizzo della periferica come se fosse un network e quindi di rendere più portatile il sistema.

3.3.1 PCAN-USB

Il PCAN-USB è un connettore sviluppato dalla PEAK-System in grado di interfacciare un sistema CAN bus con un pc tramite usb. Il connettore ha come standard il 9-pin DSUB(fig. 3.7). I pin utilizzati sono il CAN_L ed il CAN_H che corrispondono al bus differenziale definito dal protocollo.



1	-	Reserved
2	CAN_L	CAN_L bus line (dominant low)
3	CAN_GND	CAN Ground
4	-	Reserved
5	(CAN_SHLD)	**Optional** CAN shield
6	(GND)	**Optional** CAN ground
7	CAN_H	CAN_H bus line (dominant high)
8	-	Reserved (error line)
9	CAN_V+	**Optional** power

Figura 3.7: CAN 9-pin DUSB connector

Installazione driver Il driver per potere utilizzare il PCAN-USB è già presente sui sitemi Linux in modalità netdev. Questa modalità permette di utilizzare il socket can altrimenti non disponibile. Per verificare che sia presente nell' OS si usa il comando da shell:

```
$ grep PEAK_ /boot/config-`uname -r`
```

Un altro controllo necessario è vedere se una volta collegato il connettore alla porta usb si può vedere se il dispositivo è inizializzato tramite il comando:

```
$ lsmod | grep ^peak`
```

Se il comando non restituisce nessun output è necessario installare il driver manualmente. Dopo aver scaricato ed estratto il driver i comandi da shell necessari per installarlo sono:

```
$ make netdev
$ sudo make install
$ modprobe pcan
```

Per vedere se l'installazione è riuscita usare il comando:

```
$ cat /proc/pcan

*---------------- PEAK-System CAN interfaces (www.peak-system.com)
 ------------------
*---------------- Release_20210505_n (8.12.0) Aug 10 2021 13:12:27
 ------------------
*---------------- [mod] [isa] [pci] [pec] [dng] [par] [usb] [pcc]
 ------------------
*------------------------ 1 interfaces @ major 235 found
 ---------------------------
*n -type- -ndev- --base-- irq --btr- --read-- --write- --irqs-- -errors-
 status
32    usb   can0  ffffffff 000 0X001C 00000000 00000000 00000000 00000000
 0X0000
```

I device connessi tramite usb vengono inizializzati partendo dal numero 32 come si vede dall'output. Inoltre è stata creata l'interfaccia per il socketCAN che corrisponde a can0. Un'altra informaione utile che si può estrarre è la velocità di comunicazione che di default è di 500 kbit/s. Un ultimo controllo che si può effettuare per vedere se il device è stato creato è tramite il comando:

```
$ ls /dev | grep can
```

3.3.2 SocketCAN

Il driver appena installato si basa sui character devices e questo permette esclusivamente di inviare o ricevere frame tramite CAN bus. Un'altra limitazione è che solo un processo è in grado di accedere al device. Inoltre tra i vari driver esistono leggere differenze che rendono difficile il porting di API da un dispositivo all'altro.
Il socketCAN elimina queste limitazioni usando un sistema simile al protocollo TCP/IP. Viene creato un nuovo protocol family chiamato PF_CAN contenente il CAN_RAW protocol e vengono aggiunti vari driver di dispositivi CAN. Questo porta alla possibilità da parte di una applicazione di ricevere o inviare frame tramite funzioni usate per i socket

convenzionali portando ad una maggiore astrazione. Inoltre i protocolli di comunicazione permettono l'implementazione di filtri e aumenta la portabilità tra firmware diversi.

Le funzionalità che il CAN networking device driver apporta sono:

- il kernel inizializza e configura il network.
- I frame ricevuti vengono portati fino al layer superiore.
- I frame in uscita vengono portati fino al layer inferiore ed inviati attraverso il bus

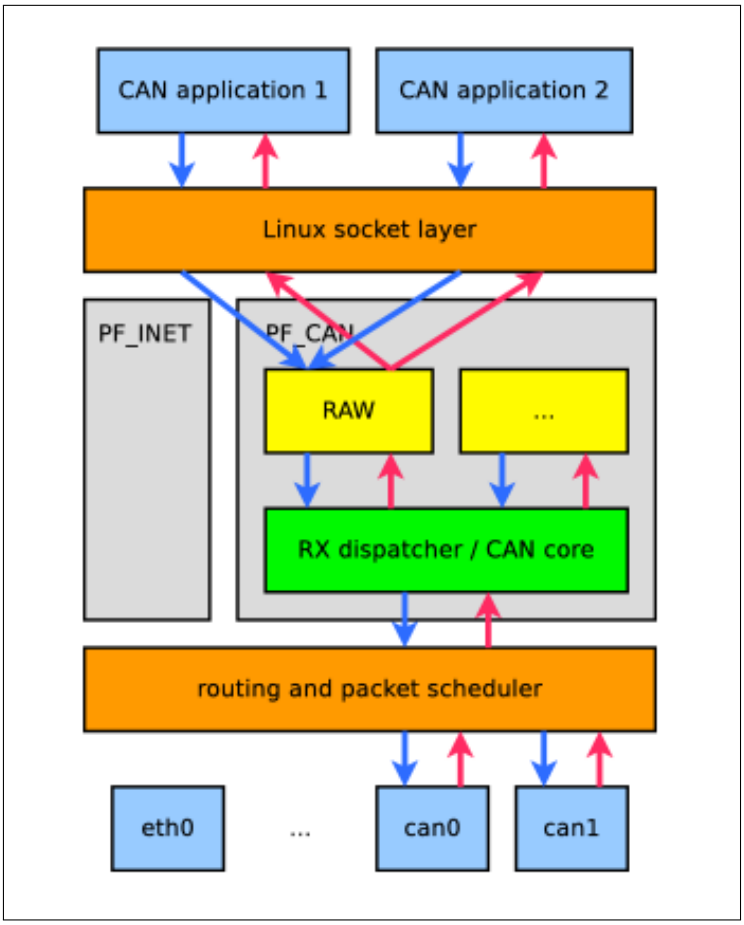


Figura 3.8: SocketCAN layer diagram

Inizializzazione

Per potere inizializzare il network bisogna prima installare il driver PEAK-linux in modalità netdev ed inizializzare il CAN-bus. A questo punto si inizializza il network tramite i comandi da shell:

```
$ sudo ip link set up can0
$ ifconfig
```

Il primo comando serve per creare il network mentre il secondo viene per controllare che l'inizializzazione sia stata portata a termine in modo corretto. La corretta configurazione del network porta all'output:

```
can0: flags=193<UP,RUNNING,NOARP>  mtu 16
        unspec 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00  txqueuelen
    10  (UNSPEC)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Per ottenere maggiori informazioni si può utilizzare il comando:

```
  $ ip -details -statistics link show can0
2: can0: <NOARP,UP,LOWER_UP,ECHO> mtu 16 qdisc pfifo_fast state UP qlen 10
  link/can
  can <TRIPLE-SAMPLING> state ERROR-ACTIVE restart-ms 100
  bitrate 500000 sample_point 0.875
  tq 125 prop-seg 6 phase-seg1 7 phase-seg2 2 sjw 1
  pcan: tseg1 1..16 tseg2 1..8 sjw 1..4 brp 1..64 brp-inc 1
  clock 8000000
  re-started bus-errors arbit-lost error-warn error-pass bus-off
  41         17457      0          41         42         41
  RX: bytes  packets  errors  dropped overrun mcast
  140859     17608    17457   0       0       0
  TX: bytes  packets  errors  dropped carrier collsns
  861        112      0       41      0       0
```

Tra i vari dati forniti i valori dei segmenti del bit devono essere concordi per tutti i nodi del bus. Per potere vedere tutti i parametri configurabili si utilizza l'istruzione:

```
$ ip link set can0 type can help
```

4. ROS2

Il Robot Operating System(ROS) è un insieme di framework open source utilizzato per scrivere software per robot. La possibilità di partire da una base di librerie funzionanti e non da zero rende lo sviluppo di un robot molto più rapida e permette uno sviluppo di un robot in tempi sostenibili. Inoltre la somma e la complessità di tutti i task richiesti da un robot per essere definito tale sono oltre le possibilità della maggior parte delle istituzioni. Per questi motivi il ROS è largamente utilizzato. Dal 2018 viene utilizzata la seconda versione chiamata ROS2 che porta varie migliorie al framework iniziale.

Un sistema di più pacchetti(le librerie del ROS2) che lavorano in sincrono crea una struttura definita "ROS graph". Gli elementi che compongono questa struttura vengono chiamati nodi. Un nodo è l'elemento fondamentale che serve per svolgere un compito specifico. Ogni nodo può inviare o ricevere dati da altri nodi tramite:

- topics
- services
- actions

Topic Questo metodo di comunicazione più utilizzato. Ha una funzione molto simile ad una richiesta server-client. Un nodo pubblica un dato in un topic e ogni nodo iscritto riceverà quel messaggio. La comunicazione può essere point-to-point, one-to-many, many-to-one e many-to-many(fig. 4.3).

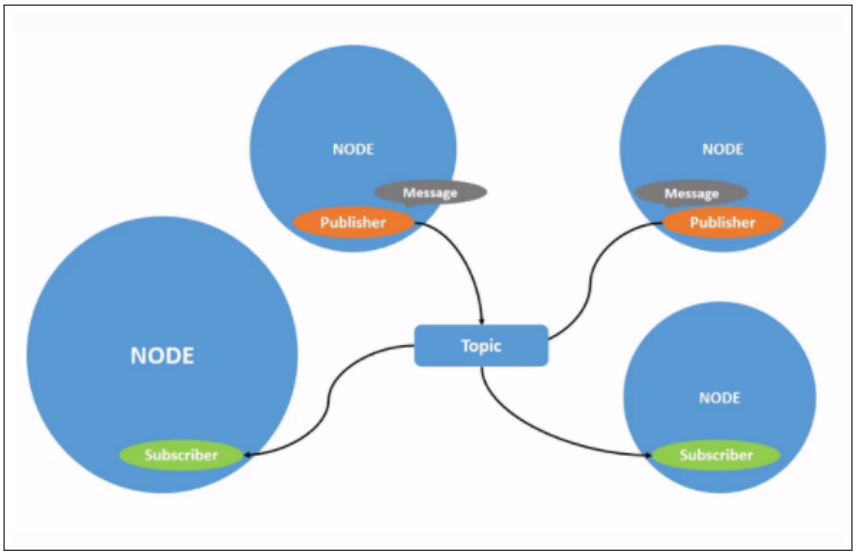


Figura 4.1: ROS topic many-to-many

Service Il servizio è basato su un modello call-and-resposta. Viene fornito un dato solo quando arriva una richiesta da un client. Può esserci solo un server che invia i dati mentre i client non hanno un limite prestabilito.

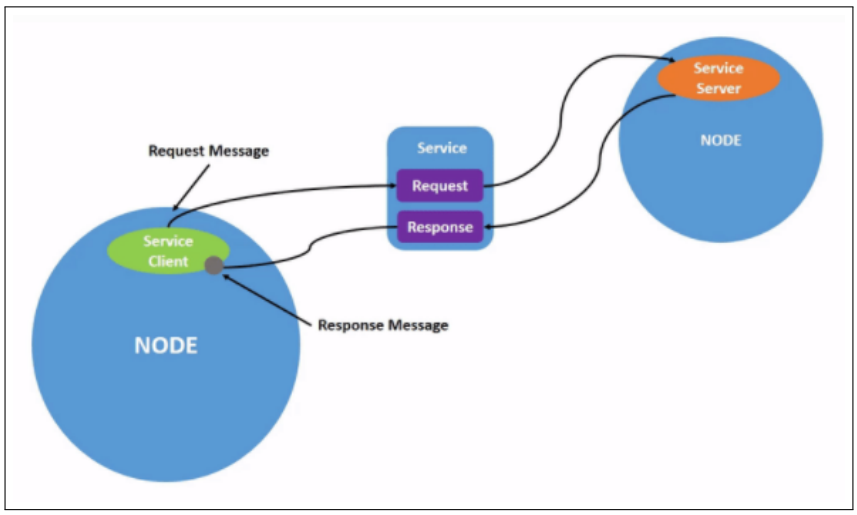


Figura 4.2: ROS service

Azioni vengono utilizzate quando un task ha un tempo computazionale elevato. é diviso in tre parti: un obbiettivo, un feedback ed un risultato. Una azione si basa su un topic e su due servizi e il workflow è:

1. Il client richiede al server di eseguire una opearazione(goal request).
2. Il server informa il client che ha eseguito l'operazione(goal response).
3. Il client richiede il risultato della operazione(result request).
4. Il server invia i dati(feedback topic.)
5. Il server informa il client che ha finito la trasmissione dei dati(result response).

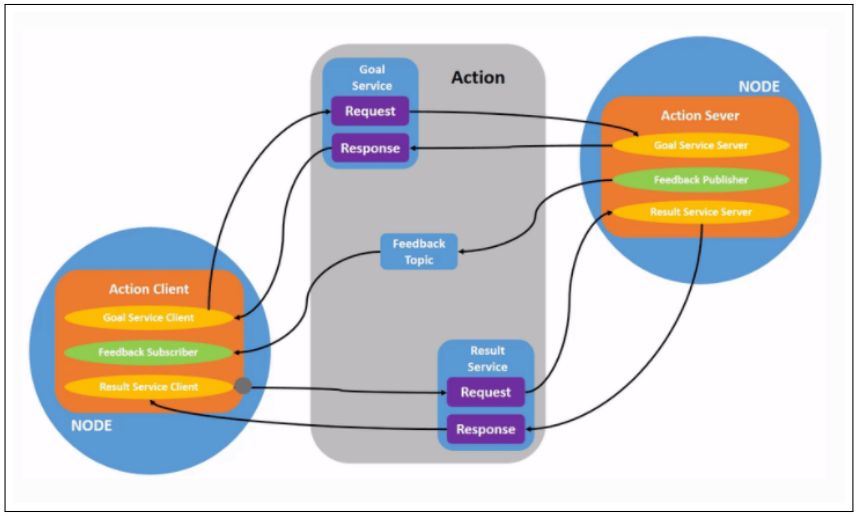


Figura 4.3: ROS action structure

5. Micro-ROS

Il micro-ROS è un framework sviluppato sulla base del ROS2. Ha lo scopo di integrare nel sistema anche microcontrollori con risorse limitate che altrimenti non sarebbero in grado di utilizzare il ROS2. Il micro-ROS è nato a causa della grande quantità di microcontrollori presenti in un robot che gestiscono la quasi totalità del layer hardware. Questo rende lo sviluppo software molto lungo e di difficile gestione ed espansione.

Il micro-ros riesce ad integrare una struttura di layer che arrivano fino al livello applicativo e nel contempo richiede una quantità minima di risorse del sistema su cui è installato. Per fare ciò delega la maggiore parte delle operazioni al ROS2 tramite il micro XRCE-DDS. Questo middleware permette di creare un protocollo di trasmissione tra micro ROS e ROS2 creando un client e agent al loro interno. In questo modo il client comunica con l'agent che a sua volta comunica con il DDS. Il global-data space è lo spazio in cui tutte le entità del ROS2 esistono e comunicano. Utilizzando il XRCE le entità create nel micro-ROS vengono integrate in questo spazio e possono quindi essere utilizzate come se fossero nel dispositivo principale con una potenza di calcolo e risorse maggiori rispetto ad un microcontrollore.

Per permettere di creare tutte le entità del ROS2 è stata creata la libreria rclc che implementa le funzioni normalmente utilizzate in c++ in c. Questo permette di utilizzare un linguaggio più a basso livello e quindi vicino al microcontrollore per avere un controllo maggiore sulle flusso del programma. I vari timer e entità generati con questa libraria sono gestiti da un executor, che implementa il concetto di priorità all'interno del ROS2. Questo permette lo sviluppo di applicazioni real time con un buon determinismo sul tempo di esecuzione già garantito dal sistema operativo sul quale viene eseguito.

6. Implementazioni

I due framework descritti nei capitoli precedenti molto spesso convivono nello stesso sistema e quindi è necessario un metodo di comunicazione. Il metodo più utilizzato è per via seriale. La facilità di utilizzo è una dei pregi maggiori di questo tipo di comunicazione grazie all'invio continuo di un numero predefinito di bit seguito da un singolo bit di controllo. Purtroppo questo metodo può essere soggetto ad errori che nelle applicazioni tipiche del ROS non sono accettabili. Per questo motivo il progetto ha come obiettivo quello di utilizzare come protocollo di comunicazione tra ROS2 e micro-ROS il CAN bus. Il protocollo tramite la gestione di 4 tipologie di errore presenta un possibilità di errore non detettato tendente allo zero. Inoltre nel protocollo è definito un algoritmo per escludere nodi dal bus nel caso di ripetuti errori. Oltre a ciò l'utilizzo di una trasmissione differenziale permette una buona reiettivo ai disturbi. Questi fattori fanno del CAN bus un ottimo sistema di comunicazione per il ROS.

6.1 Prima implementazione

Un primo approccio al problema è quello di utilizzare le caratteristiche dei nodi del ROS per potere creare un layer superiore al CAN bus. La **??** mostra uno schema concettuale della implementazione del CAN bus con il ROS. Vengono creati due nodi, uno sul NUC e uno sulla NUCLEO, che tramite l'utilizzo di un topic per la ricezione e trasmissione gestisce la comunicazione. Per la parte di invio dei dati viene creato un topic in cui i nodi della applicazione possono pubblicare. Il nodo CAN_Bridge è iscrito a questo topic e si occupa di trasmettere i dati ricevuti secondo le modalità del dispositivo in cui si trova. Analogamente per la ricezione esiste un topic apposito in cui chi vuole ricevere dati da nodi presenti nell'altro dispositivo deve iscriversi. Il CAN_Bridge si occupa di pubblicare i dati in arrivo dal CAN bus.

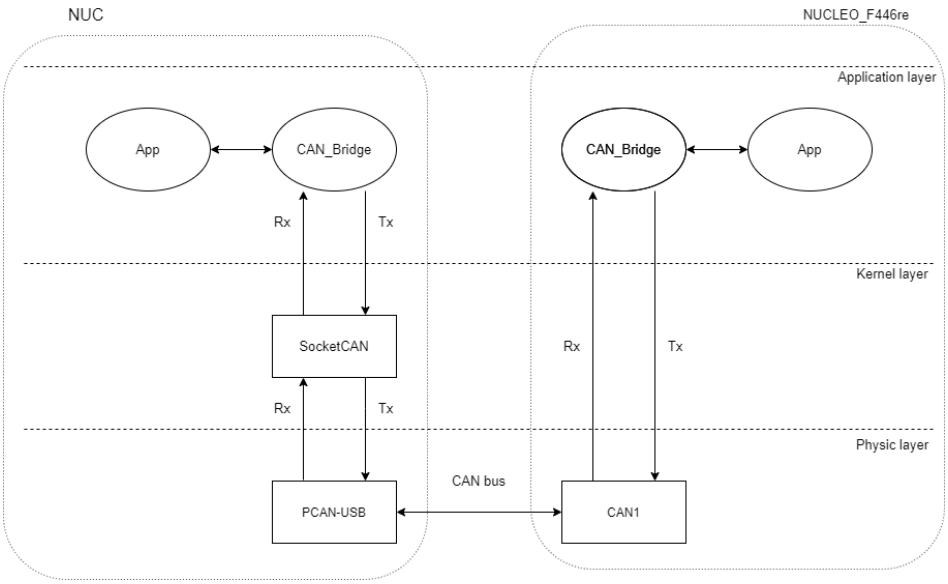


Figura 6.1: Layers interfacciamento CAN

La struttura della applicazione con i relativi nodi è mostrata in fig. 6.2. Tutti i nodi

utilizzano il topic come metodo di comunicazione. Questa è una limitazione dovuto alla organizzazione dei nodi. Le operazioni disponibili sono:

- Rilevamento del valore della tensione misurata da un ADC sulla board.
- Settaggio da parte del NUC di un PWM.

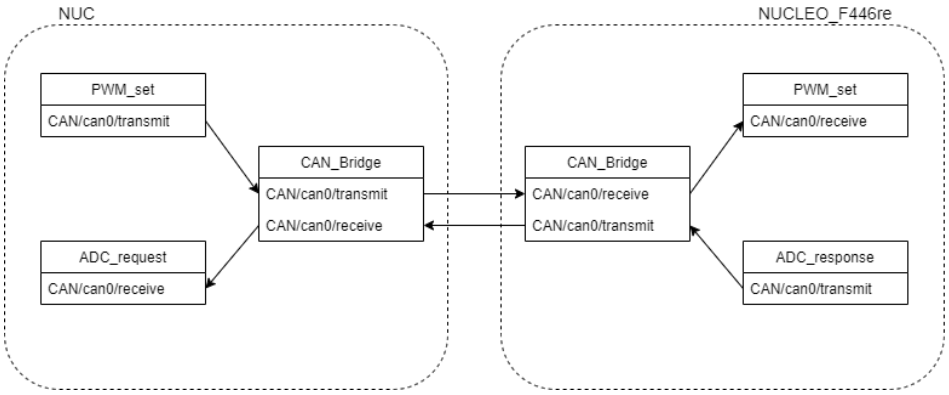


Figura 6.2: Struttura generale implementazione

Come tipo di dato utilizzato per la comunicazione viene creata una struttura adatta alla trasmissione tramite protocollo CAN. Questa struttura purtroppo limita la grandezza del payload a 8 byte.

6.1.1 CAN_Bridge

Il nodo CAN_Bridge rappresenta il nodo centrale di questa implementazione. Il pacchetto utilizzato è chiamato ros2can_bridge([]). Purtroppo il pacchetto non è mantenuto da circa 4 anni e quindi ha bisogno di modifiche per poterlo utilizzare.

- In can_msgs/package.xml: alla riga 3 cambiare il package format da 2 a 3. Aggiungere <member_of_group>rosidl_interface_packages</member_of_group> dopo le righe contenenti <exec_depend>.
- In ros2socketcan_bridge/src/ros2socketcan.h: cancellare i riferimenti ai servizi del pacchetto msgs. Nella riga 53 #include "can_msgs/srv/can_request.hpp", cancellare riga 89 rclcpp::service::Service<can_msgs::srv::CanRequest>::SharedPtr server_ros2can, eliminare da riga 118 a 121 la funzione void ros2can_srv,
- In ros2socketcan_bridge/src/ros2socketcan.cpp: nelle funzioni create_publisher, create_publisher, create_subscription aggiungere dalla riga 23 a 25 come secondo argomento il valore 10. Questo numero rappresenta il valore di quality of service utilizzato.

Per utilizzare il pacchetto bisogna buildarlo tramite il colcon build e poi refreshare i pacchetti del ROS. A questo punto basterà utilizzare il comando ros2can_bridge per far partire il pacchetto.

```
$ colcon build
$ . install/local_setup.bash
$ ros2can_bridge
```

Vengono creati 2 topic definiti come receive e transmit. Il nodo ROS2can_bridge Nel topic receive ascolta il CAN bus e quando arriva un frame lo pubblica. Invece nel topic transmit il nodo si iscrive e ogni nodo che vuole inviare un frame lo pubblica in questo topic. Il ROS2can_bridge si occupa di inviarlo tramite CAN bus. Questo pacchetto nel caso del sistema UBUNTU utilizza per le operazioni di lettura e scrittura un socket per collegarsi col CAN bus. Nel caso del microcontrollore ST invece vengono utilizzate le funzioni HAL specifiche per il CAN.

6.1.2 Svantaggi

Il sistema anche se facilmente configurabile presenta gravi lacune e problematiche. La più limitante risulta essere la gestione dei messaggi. Un nodo per ricevere un messaggio specifico deve iscriversi al topic receive ma in questo topic vengono pubblicati tutti i messaggi in arrivo rendendo di fatto la comunicazione di tipo broadcast. Per ciò serve che un nodo sia in grado di discriminare ogni messaggio e scegliere quelli utili. Questo può essere fatto utilizzando il campo ID presente nel protocollo CAN ma risulta molto macchinoso da utilizzare. Inoltre anche il passaggio di informazioni risulta complicato. Ogni nodo dovrebbe essere in grado di decodificare in modo corretto il payload rispetto al tipo o struttura utilizzato. Questo rende meno efficace la struttura stessa del ROS che prevede la possibilità di inviare messaggi di ogni tipo con facilità. Oltre a ciò l'ampliamento delle funzionalità presente in un nodo risulta sempre più complesso a causa dell'aumento dei filtri da utilizzare. Quindi questa soluzione risulta funzionante ma non abbastanza efficace e risulta necessario trovarne un'altra.

6.2 Seconda implementazione

L'obbiettivo della seconda implementazione è quello di permettere l'utilizzo libero di ogni tipo di dato astraendosi dal metodo di comunicazione. La possibilità inoltre di non dovere passare obbligatoriamente da un nodo di supporto per arrivare ad un terzo risuterebbe un ottimo miglioramento rispetto alla prima implementazione. Per potere compiere questa operazione bisogna portare i nodi presenti nel micro-ROS all'interno del DDS utilizzato dal ROS2.

6.2.1 DDS

Il DDS(Data istribution Service) descrive un modello di distribuzione data centrico tramite un modello publish-subscribe in grado di stabilire una distribuzione di informazioni efficiente ed affidabile per sistemi real-time. Il DDS API descrive l'applicazione che crea un canale tra il publisher ed i subscriber. Una volta che un dato viene pubblicato esso viene direttamente scritto nella cache locale del subscriber. Viene adibito uno spazio globale definito DDS Domain in cui il DDS aggiunge publisher e subscriber e crea il canale di comunicazione tra di essi(fig. 6.3).

Per il ROS2 è stato scelto come Middleware di default il DDS di Eprosima[]. Questa scelta è stata presa a causa della similitudine di utilizzo degli elementi publishers-subscribers. Il ROS2 delega al DDS la crazione e la gestione di tutti gli elementi del ROS. Questo permette di mantenere il ROS ad un livello di astrazione alto e al contempo utilizzare un metodo di comunicazione già ampliamente testato e quindi affidabile.

L'utilizzo del DDS necessita una buona potenza di calcolo e risulta oneroso sotto il punto

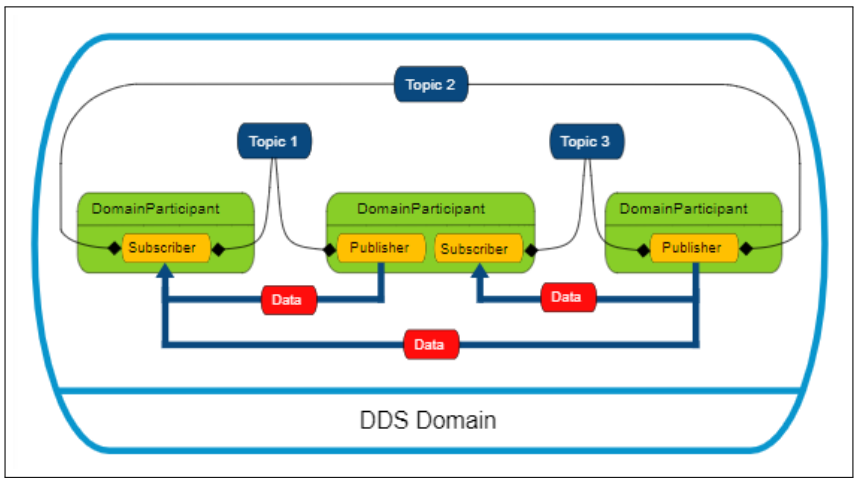


Figura 6.3: DDS global space

di vista della memoria utilizzata. Per questi motivi non risulta particolarmente adatto al micro-ROS che viene utilizzato su piattaforme embedded che di solito scarseggiano in questi due ambiti. Quindi è stato ideato il Micro XRCE-DDS[], il middleware standard per il micro-ROS. L'utilizzo delle risorse è limitato, per esempio una applicazione publisher-subscriber necessita di meno d 75 kB di memoria flash e intorno ai 3 KB di RAM per operare correttamente. Il middleware utilizza un client che gestisce il micro-ROS e un agent presente nella macchina in cui è presente il ROS2(fig. 6.4).

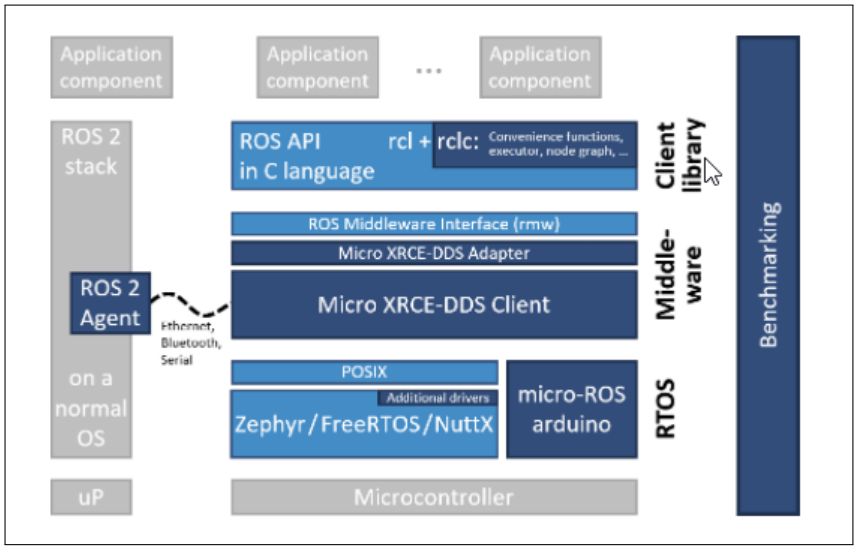


Figura 6.4: Implementazione XRCE-DDS

Queste due entità permettono al microcontrollore di eseguire solo le operazione meno onerose. Il client può richiedere di eseguire una operazione da parte dell'agent, il quale una volta completata l'operazione restituirà il risultato di essa. Tramite queste operazioni l'agent(e di conseguenza il micro-ROS) può accedere al DDS domain abilitando così l'interoperabilità tra i due sistemi coinvolti. Le operazioni che possono essere compiute sono:

- Creazione di una sessione: questa è la prima operazione da svolgere per potere eseguire le successive.

- Cancellazione di una sessione: è l'ultima operazione da svolgere e non permette altre operazioni eccetto la creazione di una nuova sessione.
- Creazione di una entità: esiste la possibilità di creare una entità sul micro-ROS direttamente dal dispositivo su cui è presente l'agent.
- Cancellazione di una entità: la sessione può chiaramente anche cancellare le entità precedentemente create
- Richiesta dati: l'agent dopo aver ricevuto una richiesta preleva il dato richiesto dal DDS data space e lo fornisce all'agent.

L'implementazione del middleware permette di ovviare alla quasi totalità delle limitazioni della prima implementazione. I protocolli di comunicazione a livello di trasporto già implementati sono la seriale, UDP e TCP ma si può aggiungere un modalità customizzata.

6.2.2 Custom transport layer

Per potere inviare dati il XRCE-DDS utilizza un transport layer che definisce il protocollo a cui sarà affidato il compito di trasmettere e ricevere i dati. Per potere aggiungere il CAN bus come custom transport bisogna modificare per agent e client le modalità di creazione e distruzione del canale, la lettura e scrittura del messaggio e la creazione del messaggio stesso.

Pacchetto dati

Il pacchetto usato del middleware può essere al massimo di 64 byte. Nel pacchetto vengono forniti oltre ad un payload l'indirizzo del device che vuole inviare il messaggio e l'indirizzo di arrivo del device. Questo pacchetto deve essere integrato nel payload del protocollo 2.0. Purtroppo la grandezza del payload del CAN2.0 è di massimo 8 byte. Bisogna quindi impostare la comunicaione come framing. Questa modalità permette al middleware di ricostruire il pacchetto aspettato anche se diviso.

Questa accorgimento può essere evitato introducendo il CAN FD(CAN flexible datarate) come protocollo. Infatti questo protocollo con i 64 byte massimi di payload risulta più semplice da gestire rispetto al CAN2.0(fig. 6.5). Purtroppo questo protocollo è poco diffuso sui microcontrollori: al momento l'unica famiglia della ST ad utilizzarlo è l'STMF7. Comunque il CAN2.0 può essere utilizzato come metodo di trasporto al costo di una percentuale più bassa di bit utili a causa dell'overhead aggiuntivo ripetto al CAN FD.

Invio di dati

Il processo per inviare un dato è mostrato in fig. 6.6. I passaggi per permettere la trasmissione sono:

- il publisher richiede al client di inviare un dato.
- il client serializza il dato all'interno di un XRCE message che viene memorizzato in un output stream buffer.
- il client chiama lo stream framing protocol.

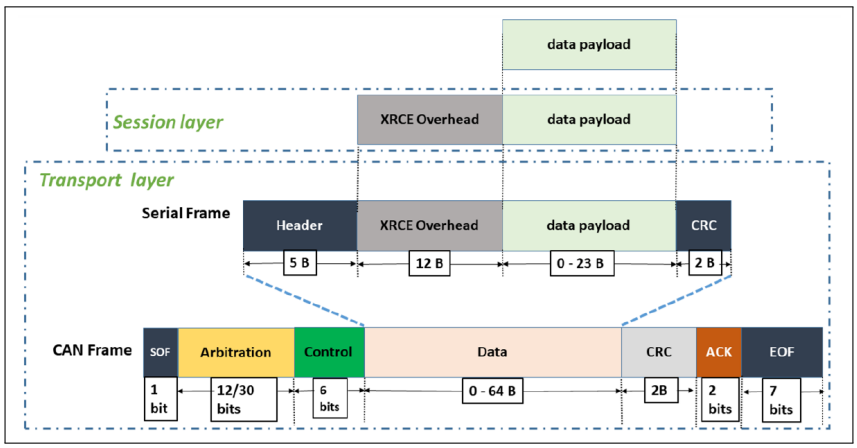


Figura 6.5: Frame CAN FD su micro-ROS XRCE

- lo stream protocol aggiunge l'overhead al XRCE message. Se è stata selezionata la modalità framing il messaggio viene diviso. Lo spazio di memoria utilizzato per questa operazione è chiamato framing buffer.
- quando il framing buffer è pieno viene chiamato il device transport che si occupa di inviare il dato tramite il trasporto utilizzato.

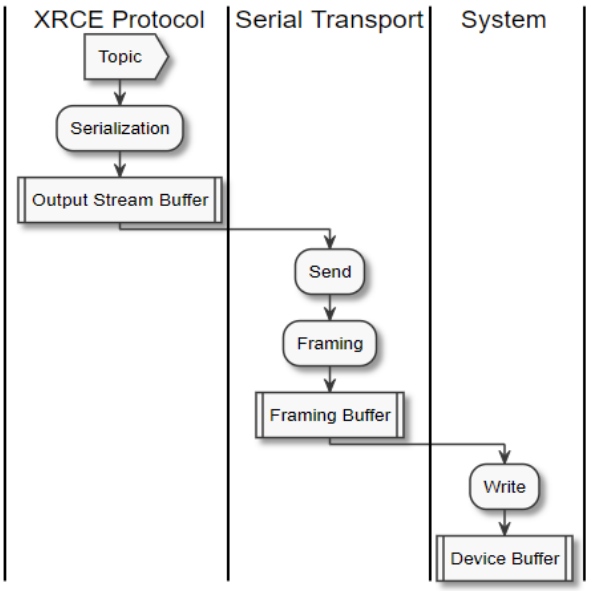


Figura 6.6: Workflow sending data

Ricezione di dati

Per potere ricevere un dato vengono eseguite le operazioni mostrate in fig. 6.7che sono:

- il subscriber invia un richiesta al client.
- il client chima lo stream framing protocol che legge dal device buffer il dato da ricevere spacchettando il frame. Il valore viene messo nell'unframing buffer
- ogni volta che l'unframing buffer è pieno i dati vengono spostati nell'input stream buffer. Questa operazione viene ripetuta finché l'input stream buffer è pieno.

- quando l'input stream buffer è pieno il client deserializza il topic dallo stream buffer.

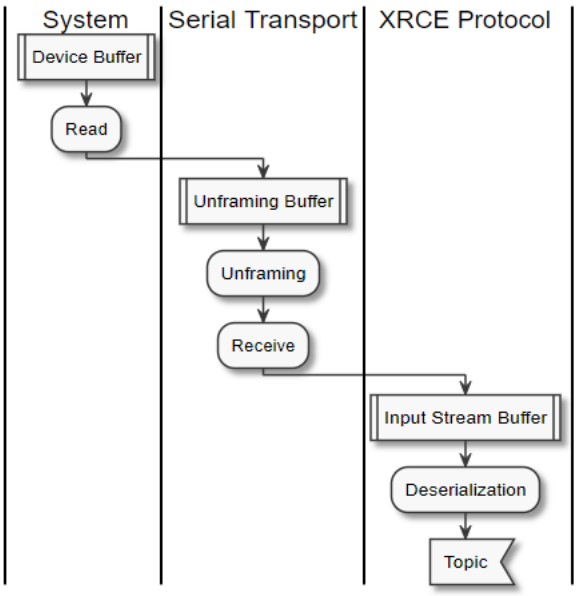


Figura 6.7: Workflow receive data

6.2.3 Custom client

Il micro-XRCE-DDS client è l'API per poter generare un client nel microcontrollore. Essendo il client presente sul microcontrollore gestisce il CAN bus direttamente con le funzioni HAL. Per potere settare un trasporto personalizzato bisogna implementare quattro funzioni:

- funzione di apertura del trasporto
- funzione di chiusura del trasporto
- funzione di invio di un dato
- funzione di ricezione di un dato

Open function

Questa funzione ha lo scopo di aprire ed inizializzare il custom transport. Il valore di ritorno deve essere un booleano che indica la riuscita o meno dell'operazione. Inizialmente si attiva il modulo CAN tramite funzione HAL_Start. I filtri vengono inizializzati per potere accettare solo i dati con l'ID riferito alla board. Quindi il campo ID che nel protocollo CAN viene utilizzato per i messaggi in questo caso viene utilizzato per identificare il device. Escludendo tutti gli altri ID si evitano errori di collisione nel bus se sono presenti più client. Infine si attiva la callback relativa alla ricezione di messaggi.

Close function

Questa funzione serve per terminare il custom transport e ritorna un valore booleano in caso di riuscita. Viene usata la funzione HAL_Stop per compiere questa operazione.

Write function

La funzione di write serve per inviare dati con il custom transport. Il costrutto che definisce la funzione è:

```
size_t my_custom_transport_write(
        uxrCustomTransport* transport,
        const uint8_t* buffer,
        size_t length,
        uint8_t* errcode);
```

Gli argomenti passati sono:

- un puntatore ad un elemento uxrCustomTransport. In questa variabile vengono passati gli argomenti indicati durante la creazione del transport. In questo caso viene passata la struttura che gestisce il CAN.
- un puntatore ad un array di uint8_t che contiene il dato da inviare.
- il numero di byte da scrivere.
- un valore di errcode per indicare se l'operazione è andata a termine.

Il ritorno della funzione indica il numero di byte inviati. Se il protocollo di comunicazione non è in grado di inviare per intero il numero di byte richiesti bisogna utilizzare la modalità stream-oriented come in questo caso. La modalità di invio viene mostrata in fig. 6.8.

Inizialmente viene settato l'ID del frame coerentemente con l'ID statico assegnato alla board. Questo permette di identificare la provenienza del messaggio ed evitare che altri client considerino il messaggio inviato dall'agent. Dopo di che viene calcolato il valore del payload da inviare rispetto ai lunghezza dei byte ancora da inviare. Se un messaggio da inviare è di lunghezza superiore ad 8 viene inviato a gruppi di 8 byte per poi modificare la lunghezza del payload dell'ultimo CAN frame. Infine si controllano che i byte inviati corrispondano a quelli riciesti negli argomenti passati.

Read function

La funzione di read serve per potere leggere i dati in ingresso provenienti dal master. Il prototipo viene difinito come:

```
size_t my_custom_transport_read(
        uxrCustomTransport* transport,
        uint8_t* buffer,
        size_t length,
        int timeout,
        uint8_t* errcode);
```

Gli argomenti che vengono passati sono simile alla funzione di write, ad esclusione del fatto che bisogna scrivere il numero di byte specificati da length nel buffer invece di leggerli. Inoltre viene aggiunto un timeout dopo il quale la funzione ritornerà errore. Come per la funzione write, viene impostata la modalità stream-oriented. La funzione quindi non ha il compito di gestire l'errorcode.

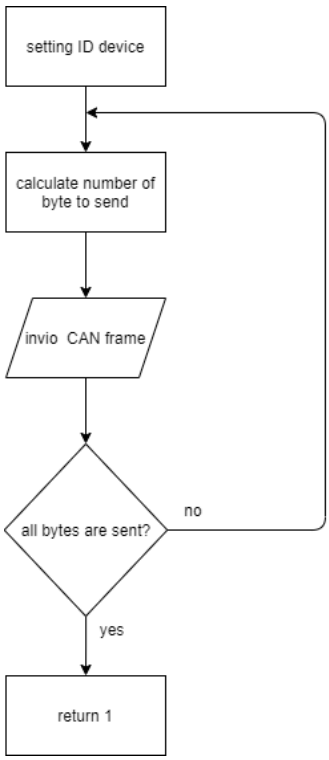


Figura 6.8: Write custom client function

La funzione di lettura opera insieme alla callback della FIFO del CAN. Ogni volta che la callback è chiamata e quindi un CAN frame è stato accettato viene scritto nella SRAM. Il payload viene memorizzato in un buffer e viene incrementato una variabile globale chiamata tail che indica la posizione nel buffer in cui l'ultimo byte è stato memorizzato.

Ogni volta che viene chiamata la funzione di read una variabile globale chiamata head viene confrontata con tail. Se è inferiore, un nuovo dato è arrivato e viene scritto nel buffer passato come argomento. Questo permette di memorizzare i dati in arrivo e passarle al middleware in modo asincrono.

6.2.4 Custom agent

L'agent è l'entità che si occupa di gestire le richieste del client. Come transport layer viene utilizzato il socket CAN. Per potere utilizzare un client personalizzato bisogna creare:

- un custom endpoint
- una funzione di apertura del trasporto
- una funzione di chiusura del trasporto
- una funzione di lettura di un dato
- una funzione di scrittura di un dato

Custom endpoint

Questo oggetto gestisce gli endpoint parameters. L'agent ha bisogno di sapere con quale client interagire. Quindi per ogni client che vuole creare una sessione viene aggiunto un elemento di tipo CustomEndPoint. Per questa applicazione viene usato un uint32_t per contenere l'identificativo dei membri. Visto che viene utilizzato l'ID del CAN bus per discriminare i vari membri si è scelto questo tipo per permettere l'utilizzo sia del CAN2.0A che del CAN2.0B.

Open function

Per aprire il custom transport viene utilizzato il socketCAN. Questo metodo permette di astrarre la funzione dal layer hardware e utilizzare le funzioni posix per potere scrivere e leggere sul file descriptor che si crea.

```
int natsock = socket(PF_CAN, SOCK_RAW, CAN_RAW);
strcpy(ifr.ifr_name, "can0");
ioctl(natsock, SIOCGIFINDEX, &ifr);

addr.can_family  = AF_CAN;
addr.can_ifindex = ifr.ifr_ifindex;

bind(natsock,(struct sockaddr *)&addr,sizeof(addr));

stream.assign(natsock);
```

Close function

Questa funzione serve per terminare il trasporto. Viene utilizzata la funzione close con argomento il file descriptor del socket creato tramite la funzione di apertura.

Write function

La funzione di write serve per potere scrivere nel custom transport. Il prototipo della funzione corrisponde a:

```
eprosima::uxr::CustomAgent::SendMsgFunction my_custom_transport_write = [&](
    const eprosima::uxr::CustomEndPoint* destination_endpoint,
    uint8_t* buffer,
    size_t length,
    eprosima::uxr::TransportRc& transport_rc) -> ssize_t);
```

I parametri della funzione corrispondono a :

- destination_endpoint: un membro CustomEndPoint che indica a chi è diretto il messaggio.
- buffer: puntatore alla zona di memoria in cui è contenuto il dato da inviare.
- length: indica il numero di byte da inviare
- transport_rc: valore che indica la riuscita o meno della operazione. Da usare se è stata scelta la modalità packet-oriented.

Se il numero di byte da inviare supera la lunghezza massima del payload del CAN bus vengono inviati più frame di seguito. Inizialmente viene settato il frame per inviare messaggi standard,con ID a 11 bit e viene impostato l'ID in baase al valore dell'endpoint di destinazione. Dopo di che si copia nel payload una porzione di buffer di massimo 8 byte per poi inviare tutto il frame tramite la funzione write. Questa operazione è ripetuta fino a raggiungere il numero di byte da inviare richiesti dall'argomento length. Questa operazione è mostrata nel listato.

```
while ( word_write!=0 && byte_write!=-1){
    if ((frame.can_dlc = word_write % 8) == 0)frame.can_dlc = 8;
    memcpy(&frame.data,ptr,frame.can_dlc);
    byte_write = write(natsock,&frame,sizeof(struct can_frame));
    word_write -= byte_write;
    ptr +=  byte_write;
}
```

Read function

Il costrutto della funzione usata per potere leggere del trasporto è simile a quella di scrittura. Il prototipo si presenta come:

```
eprosima::uxr::CustomAgent::RecvMsgFunction my_custom_transport_read = [&](
        eprosima::uxr::CustomEndPoint* source_endpoint,
        uint8_t* buffer,
        size_t length,
        int timeout,
        eprosima::uxr::TransportRc& transport_rc) -> ssize_t);
```

La differenza principale consiste nel soure endpoint. In questa variabile bisognerà informare l'agent da quale entità arriva il messaggio. Viene usato il frame ID come metodo per distinguere la provenienza del messaggio. Il fatto che sia quasi sempre necessario inviare più CAN frame per potere avere un XRCE message completo aumenta la resistenza agli errori.

6.2.5 Configurazione

Per potere utilizzare il sistema è necessario innanzitutto scaricare il package contenente il setup del microros tramite il comando da shell:

```
$ git clone -b $ROS_DISTRO https://github.com/micro-ROS/micro_ros_setup.git
    src/micro_ros_setup
```

Per poi aggiornare le dipendenze e buildare il pacchetto.

```
$ sudo apt update && rosdep update
$ rosdep install --from-path src --ignore-src -y
$ colcon build
$ source install/local_setup.bash
```

Per potere ottenere il micro-ros sulla board NUCLEO sono necessari 4 passaggi:

- creazione della repository in cui vengono scaricati tutti i pacchetti necessari alla compilazione del micro-ros, dei setup per le board supportate e del middleware.
- configurazione per la compilazione. In questo passaggio viene scelta l'applicazione da eseguire e la board che si desidera utilizzare.
- creazione dell'eseguibile .bin buildato.
- il file .bin viene inserito nella board.

I passaggi descritti vengono semplifcati da dei file bash presenti nella repository.Per potere creare la repository per la board utilizzata si utilizza il comando:

```
# Create step
$ ros2 run micro_ros_setup create_firmware_ws.sh freertos nucleo_f446re
```

Dopo questo passaggio bisogna configurare il metodo di trasporto, l'app che si vuole utilizzare e il nome che viene assegnato all'agent. Queste configurazioni vengono svolte col comando:

```
# Configure step
$ ros2 run micro_ros_setup configure_firmware_ws.sh ping_pong -d 0 -t
    custom
```

Dopo aver modificato le funzioni per impostare il custom trasport è necessrio buildare la repository per potere creare il file .bin tramite il compilatore.

```
# Build step
$ ros2 run micro_ros_setup build_firmware.sh
```

Per ultimo bisogna flashare il file .bin nel microcontrollore. Dopo questo passaggio il lato client è pronto e premendo il tasto di reset il programma partirà.

```
# Flash step
$ ros2 run micro_ros_setup flash_firmware.sh
```

Per potere creare ed eseguire l'agent bisogna scaricare e buildare il pacchetto tramite i comandi:

```
$ ros2 run micro_ros_setup create_agent_ws.sh
$ ros2 run micro_ros_setup build_agent.sh
$ source install/local_setup.bash
```

Per potere eseguire l'agent si utilizza il comando ros2 specificando il device /dev/pcan32 da cui si aprirà il custom trasport.

```
ros2 run micro_ros_agent micro_ros_agent serial --dev /dev/pcan32
```

7. Conclusioni

L'integrazione del CAN bus all'interno del sistema ROS come canale di comunicazione tra ROS2 e micro ROS risulta avere un interesse applicativo grazie al protocollo robusto e con un ottimo sistema del controllo dell'errore.

L'utilizzo del socketCAN risulta essere la scelta migliore per l'interfacciamento con il ROS2 grazie all'implementazione di un layer aggiuntivo che permette l'utilizzo del framework a prescindere dall'hardware utilizzato per il CAN bus. Per il micro ROS l'interfacciamento sul microcontrollore STM32F446re risulta facilitato dalle funzioni HAL e dalla gestione della periferica principalmente hardware.

La prima implementazione anche se funzionante ha mostrato numerose lacune. L'utilizzo dei filtri del protocollo CAN come identificativo dei topic è risultato da subito macchinoso e di difficile gestione. Inoltre l'utiizzo del driver socketCAN_bridge ha obbligato ad avere una comunicazione di tipo broadcast e non più peer to peer. Questo ha richiesto di utilizzare oltre ai filtri hardware anche una decodifica software che rallenta tutto il sistema.

Notando che la prima implementazione risultava insufficente si è deciso di optare per un cambio di approccio. Si è deciso di implementare il micro ROS XRCE e così di ottenere un exploit del sistema. Il middleware permette di aggiungere un client sul micro ROS e un agent sul ROS2 che usano come layer di trasporto il CAN bus. LA comunicaione tra agent e client permette di aggiungere le entità create nel microcontrollore a quelle create col ROS2 nel DDS space così da ottenere una interoperatività a livello applicativo. Il risultato purtroppo risulta non funzionante a causa della difficoltà implementativa dell'agent e del poco tempo rimasto doo la prima implementazione.