UNIVERSITÀ POLITECNICA DELLE MARCHE

FACOLTÀ DI INGEGNERIA

Corso di Laurea triennale in Ingegneria Informatica e dell'Automazione

Progettazione e sviluppo di nodi ROS per la gestione del CAN-bus in scenari di agricoltura di precisione

Design and Development of ROS nodes to manage CAN-bus in precision agriculture scenarios

Relatore:
Prof. Mancini Adriano

Tesi di Laurea di:
Silveri Nicola

A.A. *2019/2020*

Indice

Abstract

Questo elaborato è il risultato del tirocinio formativo previsto dal mio corso di studi in Ingegneria Informatica e Dell'Automazione Industriale e del successivo approfondimento nell'ambito del lavoro di tesi. Un'esperienza che ho svolto insieme al mio collega Luigi Di Marcantonio e sotto la guida ed i consigli del Dott. Mancini Adriano.

L'argomento trattato in questa tesi riguarda la progettazione e lo sviluppo di un'interfaccia che permetta la comunicazione tra Robot Operating System (ROS) e dispositivi su bus di tipo CAN (Controller Area Network) ovvero uno standard seriale per bus di campo. L'obiettivo dell'elaborato non è solo quello di presentare in maniera precisa il contesto progettuale, ovvero quello dell'agricoltura di precisione, ma anche quello di dimostrare come applicazioni reali possano giovare dell'utilizzo di ROS. Applicazioni nelle quali macchine agricole, attrezzi e sensoristica di bordo possono lavorare sinergicamente sfruttando tutti i benefici dell'ISOBUS[1].

La tesi affronta dapprima il contesto dell'agricoltura di precisione, introducendone i principi, le tecnologie di cui si serve e dimostrando con alcuni esempi pratici come essa sia indispensabile per chiunque oggigiorno voglia condurre un'azienda in questo settore. Si analizzeranno l'evoluzione e la diffusione dell'agricoltura di precisione non solo sul territorio italiano, dove il contesto agricolo ha sempre ricoperto un ruolo fondamentale nell'economia del paese, ma anche oltre confine.

Successivamente verranno presentati tutti gli strumenti ed i metodi che sono stati utilizzati durante lo sviluppo. Partendo da ROS fino alla sensoristica utilizzata da Arduino ogni componente hardware o software verrà esaminata per chiarire al lettore quali siano gli "attori" del progetto.

Si procede poi con quello che risulterà essere il cuore di questo elaborato: la progettazione e lo sviluppo. In questa parte verrà ripercorso tutto il workflow seguito da me e il mio collega sotto supervisione del Dott. Mancini. Questa tesi, differentemente da quella del mio collega Luigi Di Marcantonio, si concentrerà maggiormente sulla parte ad alto livello dell'interfaccia. Verrà inizialmente presentato il progetto con un linguaggio naturale e utilizzando schemi di facile

comprensione. Successivamente si scenderà più in basso con il livello di astrazione privilegiando un linguaggio tecnico. Il capitolo si chiuderà poi con la presentazione e l'analisi del codice prodotto.

Al fine di valutare la qualità finale dell'interfaccia realizzata verrà dedicato un intero capitolo all'analisi dei risultati e verranno discusse le motivazioni che ci hanno portato ad effettuare alcune scelte durante lo sviluppo.

Parte conclusiva dell'elaborato tratterà delle conclusioni tratte da tale progetto e degli eventuali sviluppi futuri.

1. Introduzione

1.1 Contesto di Progetto

Quando parliamo di agricoltura generalmente facciamo riferimento a tutti quei processi di lavorazione della terra che hanno come obiettivo la coltivazione di specie vegetali da utilizzare poi a scopo alimentare. Nel paleolitico gli uomini si nutrivano solo con le risorse alimentari che la terra offriva loro spontaneamente (fossero esse di tipo vegetale o animale). Solo al termine dell'ultimo periodo glaciale, iniziato circa 110.000 anni fa e terminato poco meno di 10.000 anni fa iniziarono ad emergere le prime organizzazioni dell'uomo con l'unico scopo di produrre cibo. Facendo quindi di necessità virtù l'uomo, organizzatosi in gruppi, stava inconsciamente dando il via allo sviluppo di comunità sedentarie, progressivamente strutturate in villaggi e città legate da una sola condizione: la sopravvivenza. Probabilmente, se la terra avesse continuato a fornire risorse alimentari sufficienti nonostante la glaciazione, l'uomo non avrebbe conosciuto il concetto di civiltà chi sa per quanto altro tempo. Ci accorgiamo dunque che l'introduzione del concetto di agricoltura rappresenta uno step evolutivo molto importante per la specie umana. Vale la pena dunque ripercorrere brevemente la storia di questa pratica, dalle origini fino ai giorni nostri.

La storia e lo sviluppo dell'agricoltura vanno di pari passo con la specializzazione dell'uomo. Durante questi 10.000 anni infatti l'uomo non solo ha potuto sviluppare conoscenze e tecniche di coltivazione sempre più avanzate ma ha anche rincorso costantemente lo sviluppo tecnologico. Se si analizza l'evoluzione dell'agricoltura emergono progressivamente alcune tappe considerabili vere e proprie pietre miliari per tale percorso. Si passa da un'agricoltura di sussistenza (ovvero con il solo obiettivo di sfamare i coltivatori e le loro famiglie) ad un'agricoltura estensiva basata su proprietà agricole sempre più ampie e sugli effetti della rotazione delle colture fino ad un'agricoltura di tipo intensiva e meccanizzata.

Oggigiorno chi conduce attività in questo settore deve fronteggiare spesso questioni riguardanti l'introduzione di tecniche di ingegneria genetica, informatica ed automazione, l'ottimizzazione dei processi produttivi, l'eco sostenibilità, bilanci economici e molto altro. Quando si parla di Agricoltura di Precisione (AdP) si fa proprio riferimento a tutte le nuove metodologie e tecnologie che hanno permesso all'agricoltura, in questi ultimi anni, di compiere un ennesimo passo evolutivo.

Nella coltivazione di un campo è molto probabile trovare zone a maggior resa rispetto ad altre. In queste zone il livello produttivo è più alto e tutto dipende da un insieme di fattori quali: una corretta applicazione del concime, una semina ben fatta o fattori del tutto oggettivi; ad esempio una differente composizione del terreno, la presenza di avvallamenti dove ristagna l'acqua o di zone più compattate e quindi meno porose. Riuscire a comprendere cosa genera la variabilità e quindi trovare i mezzi per porvi rimedio quando è possibile, o adattare il processo produttivo in modo da ridurre gli sprechi è uno degli obiettivi che si prefigge l'Agricoltura di Precisione. Alcuni esempi pratici di applicazioni dell'AdP possono essere:

• Sistemi in cui le macchine agricole, tramite sistema GPS, sono in grado di identificare la propria posizione sul campo in tempo reale. Questa tecnologia ha aperto le porta alla guida autonoma e assistita di queste macchine che permettono di ridurre al minimo lo stress a carico dell'operatore, i tempi impiegati e gli sprechi specialmente in termini di carburanti.

Figura 1.1.1 esempio di macchina agricola a guida autonoma
Fonte: Agrieuropa 2018

• Prelievo di campioni di terreno per conoscerne dati come la quantità, la tipologia e i rischi legati a particolari tipi di parassiti presenti in quelle zone; dati che permettono all'agricoltore di pianificare strategie per l'utilizzo calcolato di pesticidi e per delle concimazioni future sicuramente più consapevoli.

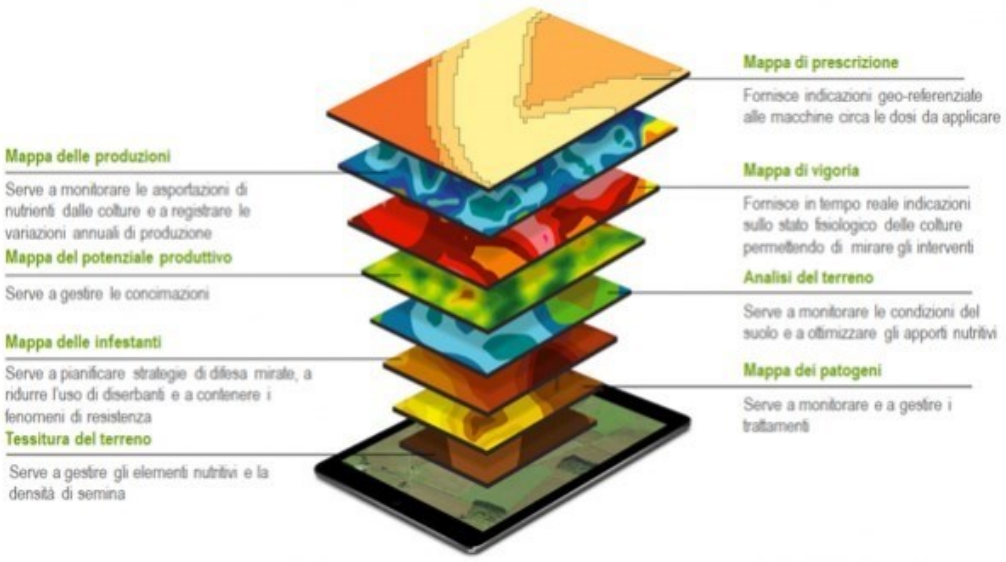


Figura 1.1.2 vari layer di un rilevamento areo di un terreno
Fonte: Agrieuropa 2018, modificata

• Macchine predisposte per una concimazione a rateo variabile, ovvero dotate di sistemi in grado di stabilire in tempo reale, durante l'esecuzione, le zone in cui il prodotto è maggiore e permettendo il rilascio automatico di una quota maggiore di concime in queste zone.

Figura 1.1.3 Spandiconcime a rateo variabile a controllo diretto mediante sensori di vigoria montati frontalmente al trattore Fonte: Agenzia europea per i sistemi satellitari di navigazione globale, 2018

È evidente come le agrotecniche attuali siano ricche di spunti tecnologici sempre più affinati e in costante aggiornamento. Ovviamente ciò può solo rappresentare un punto di partenza: con l'aumentare della popolazione (previsti quasi 9mld di individui) e, conseguentemente, delle esigenze agricole nei prossimi trent'anni qualora le agrotecniche dovessero arrestarsi allo stato attuale sarà necessario l'impiego di un numero sempre maggiore di risorse (come affermato da Carlo Bisaglia[1] nel suo articolo su agriregionieuropa del Giugno 2018). Sviluppare nuove metodologie e agrotecniche in grado di garantire la produzione agricola necessaria in quel periodo con le stesse risorse utilizzate oggi significherebbe già l'aver ottenuto un risultato soddisfacente. Riuscire invece a sviluppare un'innovazione in grado di ottenere le produzioni necessarie tra trenta anni con meno risorse di quelle utilizzate oggi, significherebbe l'aver introdotto un cambiamento strutturale di enorme portata per le generazioni future. "Produrre di più utilizzando meno" è proprio il motto dietro l'ottimizzazione proposta dell'Agricoltura di Precisione. Secondo *CEMA-European Agricultural Machinery* "L'agricoltura 4.0 ha spianato la via per l'evoluzione successiva dove la coltivazione consisterà di operazioni senza pilota e sistemi a decisione autonoma. In questo contesto, è facile capire come l'utilizzo dell'intelligenza artificiale risulti essere l'estensione più naturale che aprirà le porte all'era dell'agricoltura 5.0".

1.2 Progetto AGROSBUS

Prima di entrare nel vivo dell'analisi di quanto prodotto, richiamiamo velocemente quali sono le linee generali del progetto al quale io e il mio collega abbiamo partecipato, i suoi obiettivi e le sue caratteristiche. Il progetto *AGROSBUS*, di cui il Dott. Mancini Adriano è responsabile, vuole indirizzare all'utilizzo di ROS in contesti di agricoltura di precisione reali dove macchine agricole, organi meccanici e sensori di terze parti lavorano sinergicamente sfruttando i benefici *dell'ISO 11783 (ISOBUS)*. L'inclusione di ROS in questi contesti apre nuove vie di sviluppo per tutte le unità di ricerca che creano, testano e rilasciano nuove *feature*. Nel contesto dell'AdP che abbiamo descritto fin ora è chiaro quanto sia importante rendere più semplice l'integrazione di nuove funzionalità, che potrebbero eventualmente beneficiare anche di componenti o pacchetti di ROS già sviluppati. ROS

[1] https://agriregionieuropa.univpm.it/it/content/article/31/53/agricoltura-di-precisione-italia-unopportunita-di-aggiornamento-delle

nasce come soluzione *open-source* adatta a tutti per la realizzazione di applicazioni nel mondo della robotica. ROS facilita ad esempio l'interfacciamento con i dispositivi di bordo delle macchine agricole: i dispositivi potrebbero funzionare in maniera molto più efficiente se fatti interagire con esso, che potrebbe rivestire sia il ruolo di *controller* generale delle attività che vengono svolte sulla macchina ma anche quello di controller dedicato (*ECU*).

ll progetto AGROSBUS è attualmente finanziato come *FTP Project* da ROS-Industrial, un progetto *open-source* che mira a trasferire i valori di ROS all'*hardware* industriale, sviluppando nuovi componenti e migliorando quelli esistenti.

Figura 1.2.1 logo Ros-Industrial Fonte: www.rosindustrial.org

1.3 Obiettivi del progetto

Dopo aver chiarito quanto possa essere vantaggioso un innesto di ROS in questi scenari, riassumiamo brevemente i tre obiettivi che si intende raggiungere con il progetto AGROSBUS:

- inviare ed acquisire dati dall'*Implement Messages Application Layer* (ISO 11783-7) utilizzando un ponte ROS che permetta la comunicazione tra macchina/dispositivo e nodi ROS;
- riutilizzare pacchetti ROS già esistenti per automatizzare la navigazione di un trattore o per il mapping del terreno (e.g. *Simultaneous Localization And Mapping* SLAM);
- rendere più semplice sviluppare, distribuire, testare e applicare prodotti avanzati per scenari di agricoltura 5.0.

Nel nostro lavoro di tirocinio, io e il mio collega ci siamo occupati del primo di questi tre punti appena citati.

2. Strumenti e Metodi

2.1 Robot Operating System

2.1.1 Cos'è ROS

Lo sviluppo di robot che siano in grado di immettersi all'interno di applicazioni reali richiede oggi un grande quantità di *software*, *driver*, strumenti di terze parti e soprattutto di piattaforme di simulazione. Il reperimento di questi strumenti può risultare faticoso e può nei casi peggiori, magari per utenti poco consapevoli, costituire un muro invalicabile per chi voglia approcciarsi alla creazione di robot. Il *framework* ROS raccoglie tutti questi strumenti e gestisce il modo in cui un codice per robot viene sviluppato. Invece di reinventare la ruota ogni volta, ROS aiuta lo sviluppatore ad avere una maggiore consapevolezza di quali sono gli strumenti già disponibili e di come andarli ad includere nel proprio progetto.

Venne inizialmente sviluppato dallo *Stanford AI Laboratory* nel 2007, tuttavia ROS va ben oltre quanto appena detto: esso ricalca sotto tutti gli aspetti l'idea di sistema operativo (da cui l'acronimo) in quanto fornisce tutti i servizi tipici che ci si potrebbe aspettare da qualunque altro di essi come l'astrazione hardware, il controllo dei dispositivi di basso livello, l'implementazione delle funzionalità di uso comune, il passaggio dei messaggi tra i processi, la gestione dei pacchetti e molte altre. ROS ad oggi cresce grazie al contributo della sua comunità di sviluppatori, la sua modularità consente a ciascuno di utilizzare componenti di ROS già concepite integrandole con facilità nel proprio progetto.

La maggior parte degli utilizzatori di ROS si trova sicuramente nei laboratori di ricerca, ma la sua adozione si sta gradualmente spostando in questi anni anche ai settori commerciali, in particolare ai robot industriali e ai *service robot*, come per esempio i robot domestici ma anche verso studenti di ogni età per la robotica educativa.

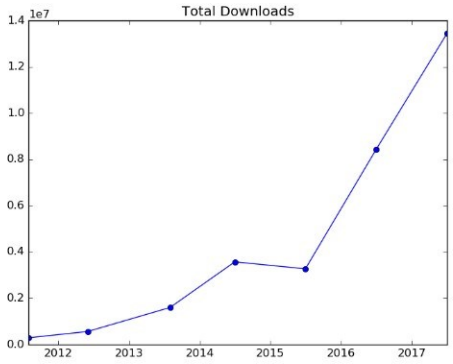


Figura 2.1.1 crescita di ROS stimata in numero di download per anno.
Fonte: www.ros.org/news/rosorg

2.1.2 Concetti e Costrutti di base

I concetti fondamentali dell'implementazione ROS sono i:

- nodi;
- messaggi;
- topic;
- servizi.

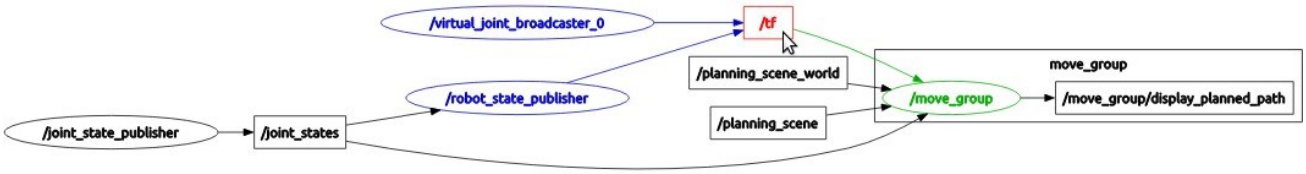


Figura 2.1.2 esempio di progetto realizzato con rqt_graph Fonte: www.wiki.ros.org/rqt_graph

I Nodi sono processi indipendenti, ognuno di essi realizza una funzione specifica e per farlo scambia costantemente informazioni con altri nodi della rete. ROS è progettato per essere modulare su una scala a grana fine: anche progetti concettualmente molto semplici sono spesso il risultato di un'unione tra tanti nodi.

In questo contesto, il "nodo" è intercambiabile con il "modulo software" inteso come componente *software* autonomo e ben identificato, e quindi facilmente riusabile. L'uso del termine "nodo" deriva dalle visualizzazioni dei sistemi basati su ROS in fase di esecuzione: quando molti nodi sono in esecuzione, è conveniente eseguire il rendering *peer-to-peer* delle comunicazioni tramite grafico per avere una visione ampia sul progetto e sui collegamenti logici che interessano i vari nodi; i processi come nodi del grafico e i collegamenti *peer-to-peer* come gli archi.

I nodi comunicano tra loro scambiandosi messaggi. Un messaggio è una struttura dati strettamente tipizzata: sono supportati tipi di dato standard (intero, virgola mobile, booleano, ecc.), così come array o matrici di questi ultimi. I messaggi possono essere composti da altri messaggi e matrici di altri messaggi, nidificati in modo arbitrariamente profondo. Un nodo invia messaggi pubblicandoli (*Publishing*) su un dato *topic*, che è identificato tramite una stringa come (es. "odometria" o "mappa") e i nodi interessati a quel determinato tipo di messaggio sottoscriveranno il *topic* appropriato (*Subscribing*). Possono esistere più *Publishers* e *Subscribers* simultanei per un singolo topic e un singolo nodo può pubblicare e/o iscriversi a più *topic* contemporaneamente. In generale, *Publishers* e *Subscribers* non sono a conoscenza gli uni degli altri ed è per questo che si introduce il *ROS Master*: esso fornisce servizi di denominazione e registrazione al resto dei nodi nel sistema ROS, tiene traccia di tutti i *Publisher*, di tutti i *Subscribers* e dei vari *topic*. Il ruolo del *Master* è quello di consentire ai singoli nodi ROS di localizzarsi l'un l'altro. Una volta individuati tra loro i nodi comunicano tra loro *peer-to-peer*.

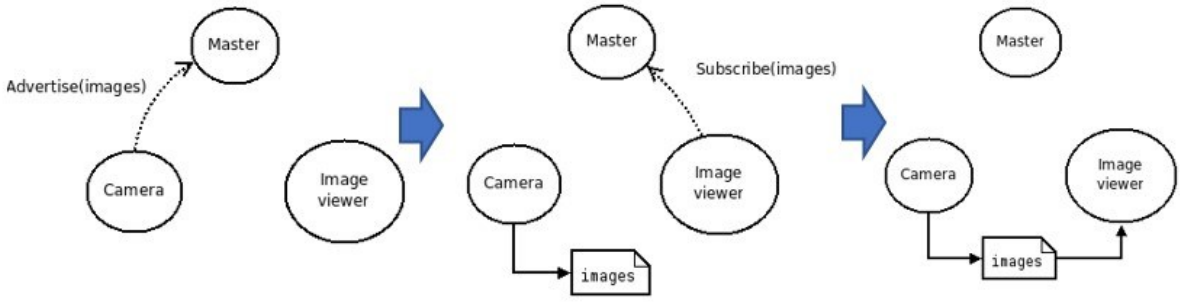


Figura 2.1.3 principio di funzionamento del ROS-Master. Fonte: http://wiki.ros.org/Master modificata

Tuttavia, i grafici sono in genere molto più complessi, spesso contengono cicli e connessioni uno-a molti o molti-a-molti. Sebbene il modello di pubblicazione/abbonamento basato su argomenti sia un paradigma di comunicazione flessibile, il suo schema di *routing "broadcast"* non è appropriato per le transazioni sincrone, che permetterebbero di semplificare la progettazione di alcuni nodi. In ROS lo scambio di messaggi in maniera sincronizzata prevede l'esistenza di un servizio, definito da un nome e da una coppia di messaggi ben definiti: uno per la richiesta e uno per la risposta. Questo è analogo ai servizi Web, che sono predefiniti dagli URL e hanno documenti di richiesta e risposta di tipi ben definiti. Si noti che, a differenza dei *topic*, solo un nodo può fornire un servizio con un nome particolare, proprio come può esserci un solo servizio web in un determinato URL.

2.1.3 Applicazioni reali di ROS

Come già sottolineato ROS sta diventando lo standard per la progettazione e programmazione di robot non solo per la ricerca, ma anche per le aziende che ne fanno uso o che li propongono come prodotti. Alcuni esempi di aziende che sono nate con ROS o che hanno iniziato una conversione verso esso sono le seguenti:

Clearpath Robotics

Clearpath Robotics è una società canadese fondata nel 2009. Il numero di robot che produce nei settori dei veicoli terrestri senza pilota, dei veicoli di superficie senza pilota (sull'acqua) e dei veicoli industriali è sorprendente. I robot dell'azienda sono basati su ROS e possono essere programmati con ROS fin da subito. Questi robot vengono utilizzati nella creazione di applicazioni di terze parti per il rilevamento, l'ispezione, l'agricoltura e la movimentazione dei materiali. Alcuni dei loro robot più noti includono *Jackal UGV*, *Husky UGV*, *Grizzly TUV* e la sua serie recentemente lanciata di robot *Otto* per ambienti industriali.

DIMENSIONS (L x W x H)	1.8 x 1.3 x 1.0 m 68 x 50 x 39 in	MAXIMUM DRAWBAR	A: 1750N / 393lbf B: 7500N / 1682lbf	USER POWER	5 / 12 / 24 / 48 V (Fused)
CURB WEIGHT (A: Base, B: Heavy-Duty)	A: 650kg / 1430lb B: 850kg / 1875lb	TOWING RUNTIME	A: 3hr B: 2hr	ENVIRONMENTAL	IP 66 -10 / +30 °C
TURNING PAYLOAD	A: 350kg / 770lb B: 150kg / 330lb	CRUISING RUNTIME	A: 8hr B: 12hr	BASE SENSORS	ENCODERS, GPS, IMU
CLIMBING PAYLOAD	A: 600kg / 1320lb B: 400kg / 880lb	CRUISING RANGE	A: 29km / 18mi B: 43km / 27mi	DRIVERS/APIs	ROS, C++
MAXIMUM SPEED	A: 4.4m/s / 10mph B: 2.2m/s / 5mph	MAXIMUM INCLINE	30 deg 60%	CONTROL MODES	TELE-OP. SEMI-AUTON. FULL-AUTON.

Figura 2.1.4 Grizzly TUV della Clearpath Robot ics e tabella delle specifiche Fonte: Clearpath Robotics

Pal Robotics

Pal Robotics ha sede a Barcellona ed è stata fondata nel 2004; è l'unica azienda al mondo che costruisce e vende robot umanoidi di dimensioni reali. Non solo un singolo tipo di umanoide, ma diverse tipologie tra le quali: Il robot *Reem*, il robot *Reem-C* e, di recente, il robot *TALOS* nel 2017.

Figura 2.1.5 evoluzione dei robot umanoidi della Pal Robotics
Fonte: Plant Automation Technology

Yujin Robotics

Yujin Robotics è una società coreana specializzata in robot aspirapolvere, essa ricopre un ruolo importante nel mercato in quanto creatrice ufficiale del robot *Kobuki*, ovvero il sistema base del robot *Turtlebot 2*. Il Turtlebot 2 è il robot ROS più famoso al mondo. Quasi ogni sviluppatore ha mosso i suoi primi passi con ROS imparando da quel robot, sia in simulazione che nella realtà. Grazie al suo basso costo, esso consente di entrare facilmente nel mondo ROS senza la necessità di copiose risorse economiche. Yujin ha anche sviluppato un altro robot ROS chiamato *GoCart*, un robot molto interessante per la logistica all'interno degli edifici che può essere utilizzato per inviare pacchi da una posizione nell'edificio ad un'altra, anche in spostamenti che prevedano l'utilizzo di ascensori.

Figura 2.1.6 logo di Yujin Robot e GoCart (robot per il trasporto merce all'interno degli edifici) Fonte: Yujin Robotics

2.1.4 ROS 2.0

L'organizzazione che gestisce e mantiene ROS, sta cercando di rendere il sistema ancora più flessibile e prezioso durante il suo passaggio dalla versione 1.0 alla versione 2.0. La necessità ROS 2.0 include il supporto per sistemi *multi-robot*, comunicazioni in tempo reale, supporto a reti non ideali e maggiori capacità di elaborazione. Questi obiettivi renderebbero ROS non solo più facile da usare, robusto e funzionale, ma anche più facile da essere accettato come standard del settore. Andando avanti, ROS dovrà continuare ad espandere il supporto ai fornitori di sistemi operativi come *Microsoft*, ai fornitori di servizi *in cloud* come *Amazon* e *Google* e ai fornitori di componenti (es. azienda *Robotiq*) e sarà sempre più facile considerarlo riferimento del settore. Ciò non comporterà

la fine dei sistemi proprietari di robotica, ci si aspetta infatti che la coesistenza di sistemi di robotica *open-source* e chiusi continui così come la conosciamo oggi. Negli ultimi anni, importanti operatori del settore come *Comau* e *NVIDIA* hanno lanciato piattaforme *hardware* di robotica *open-source* basate su ROS. È previsto dunque un coinvolgimento sempre maggiore delle principali società nel campo della robotica che porteranno sicuramente più attenzione alla comunità di ROS, più sviluppatori e più utenti finali.

2.2 Arduino

Figura 2.2.1 vista dall'alto di una scheda Arduino-UNO ed il suo IDE di sviluppo ufficiale
Fonte: wikipedia.org/Arduino-IDE modificata

Arduino è una piattaforma *hardware* composta da una serie di schede elettroniche dotate di un microcontrollore, ideata e sviluppata nel 2003 da alcuni membri dell'*Interaction Design Institute* di Ivrea come strumento per la prototipazione rapida e per scopi hobbistici, didattici e professionali. Con Arduino si possono realizzare in maniera relativamente rapida e semplice piccoli dispositivi come controllori di luci, di velocità per motori, sensori di luce, automatismi per il controllo della temperatura e dell'umidità e molti altri progetti che utilizzano sensori, attuatori e comunicazione con altri dispositivi. Tutto il *software* a corredo del microcontrollore è libero, e gli schemi circuitali sono distribuiti come risorse libere. L'ambiente di sviluppo integrato (Figura 2.2.1) permette anche ai novizi di lavorare con Arduino, in quanto i programmi sono scritti in un linguaggio di programmazione semplice e intuitivo, chiamato *Wiring*, derivato dal C e dal C++, liberamente scaricabile e modificabile. I programmi in Arduino vengono chiamati *sketch*. Grazie alla base *software* comune, per la comunità Arduino è stato possibile negli anni sviluppare programmi e librerie per connettere, a questo *hardware*, più o meno qualsiasi oggetto elettronico, *computer*, sensori, display o attuatori ed è oggi possibile fruire di un *database* di informazioni vastissimo.

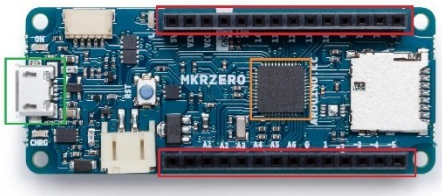


Figura 2.2.2 Arduino MKR Zero Fonte: Arduino Store

In generale una scheda di Arduino consiste in un microcontrollore, evidenziato in arancione in Figura 2.2.2 a 8-bit prodotto dalle *Atmel*, con diverse componenti aggiuntive come ad esempio le due file di pin, evidenziate in rosso o la porta microUSB, evidenziata in verde che consentono l'interazione della scheda con vari dispositivi di *input* e *output*.

Ruolo particolarmente importante per la nostra applicazione è stato rivestito dagli *Shields*. Queste schede aggiuntive permettono alle nostre schede base di essere espanse con ulteriori funzionalità (servizi di geolocalizzazione quali il GPS, Internet tramite GSM e molti altri) per l'interazione con il mondo esterno.

2.2.1 Serie Arduino MKR

La serie MKR può essere la scheda di riferimento per lo sviluppo IoT anche per coloro con minima esperienza pregressa nel *networking*. La maggior parte delle persone conosce le dimensioni di Arduino Uno (2,7 x 2,1"), ma poiché la serie MKR è stata progettata per lo sviluppo IoT, il suo *form factor* è molto compatto per adattarsi a quasi tutti i progetti con una dimensione di 2,42 x 0,98" per tutte le schede MKR. Le schede facenti parte di questa serie hanno tutte lo stesso numero di pin I/O. Le schede sono dotate di un totale di 22 pin I/O digitali di cui 12 pin PWM. Includono inoltre 7 pin per l'ingresso analogico e 1 pin per l'uscita analogica.

2.2.2 Arduino nel nostro progetto

Figura 2.2.3 Can Shield (shorturl.at/eCENW)

Figura 2.2.4 MKR Zero ed MKR Can Shield insieme (shorturl.at/kLXZ1)

Figura 2.2.5 MKR Zero (shorturl.at/dglv6)

Figura 2.2.6 MKR WAN 1300 (shorturl.at/dglv6)

Nel lavoro svolto, sono state utilizzate due schede Arduino, in particolare un Arduino ***MKR Zero*** (Fig. 2.2.3) ed un Arduino ***MKR WAN 1300*** (Fig. 2.2.4), accoppiate entrambe con uno Shield, l'***MKR CAN Shield*** (Fig. 2.2.5) di cui parleremo più avanti.

Tabella 1 Arduino MKR Zero Datasheet Fonte: Arduino Store

Microcontroller	SAMD21 Cortex-M0+ 32bit low power ARM MCU
Board Power Supply (USB/VIN)	5V
Supported Battery(*)	Li-Po single cell, 3.7V, 700mAh minimum

DC Current for 3.3V Pin	600mA
DC Current for 5V Pin	600mA
Circuit Operating Voltage	3.3V
Digital I/O Pins	22
PWM Pins	12 (0, 1, 2, 3, 4, 5, 6, 7, 8, 10, A3 - or 18 -, A4 -or 19)
UART	1
SPI	1
I2C	1
Analog Input Pins	7 (ADC 8/10/12 bit)
Analog Output Pins	1 (DAC 10 bit)
External Interrupts	8 (0, 1, 4, 5, 6, 7, 8, A1 -or 16-, A2 - or 17)
DC Current per I/O Pin	7 mA
Flash Memory	256 KB
Flash Memory for Bootloader	8 KB
SRAM	32 KB
EEPROM	no
Clock Speed	32.768 kHz (RTC), 48 MHz
LED_BUILTIN	32

Tabella 2 Arduino MKR WAN 1300 Datasheet Fonte: Arduino Store

Microcontroller	SAMD21 Cortex-M0+ 32bit low power ARM MCU
Board Power Supply (USB/VIN)	2x AA or AAA
Supported Battery(*)	Li-Po single cell, 3.7V, 700mAh minimum
DC Current for 3.3V Pin	600mA
Circuit Operating Voltage	3.3V
Digital I/O Pins	8
PWM Pins	12 (0, 1, 2, 3, 4, 5, 6, 7, 8, 10, A3 - or 18 -, A4 -or 19)
UART	1
SPI	1

I2C	1
Analog Input Pins	7 (ADC 8/10/12 bit)
Analog Output Pins	1 (DAC 10 bit)
External Interrupts	8 (0, 1, 4, 5, 6, 7, 8, A1 -or 16-, A2 - or 17)
DC Current per I/O Pin	7 mA
Flash Memory	256 KB
Flash Memory for Bootloader	8 KB
SRAM	32 KB
EEPROM	no
Clock Speed	32.768 kHz (RTC), 48 MHz
LED_BUILTIN	32
Full-Speed USB Device and embedded Host	
Antenna power	2dB
Carrier frequency	433/868/915 MHz
Working region	EU/US
Length	67.64 mm

2.2.3 Arduino MKR CAN Shield

Il *CAN Shield MKR* ha anch'esso il *form factor* MKR, permette di espandere le capacità della nostra scheda Arduino, dandoci la possibilità di interfacciarla con il protocollo CAN, utilizzato nella stragrande maggioranza dei veicoli, specialmente agricoli. Per il collegamento abbiamo a disposizione, oltre ai classici *header* per l'accoppiamento con una ulteriore scheda Arduino MKR, 4 pin come possiamo vedere sulla sinistra dalle seguenti immagini:

Figura 2.2.3.1
MKR CAN Shield vista dall'alto
Fonte: Arduino Store

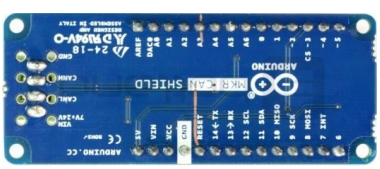


Figura 2.2.3.2
MKR CAN Shield vista dal basso
Fonte: Arduino Store

- Vin 7V+24V
- CAN-L
- CAN-H
- GND (Ground)

Andando quindi a collegare questi pin ai corrispondenti cavi presenti in un veicolo o per semplicità ad una porta OBD (Interfaccia presente nei veicoli utilizzata per la diagnostica che include anche i quattro pin appena citati (Fig. 2.2.3.1 e 2.2.3.2)) siamo in grado sia di ricevere che di leggere i dati scambiati tra tutti i nodi collegati alla rete CAN (*ABS*, *ESP*, Radar anticollisione, ecc.) e sfruttarli a nostro piacimento.

Tabella 2 Arduino MKR CAN Shield Datasheet Fonte: Arduino Store

Protocol	CAN Bus
Interface	SPI
Circuit Operating Voltage	3.3 V
Controller	Microchip MCP2515
Transceiver	NXP TJA1049
Buck converter	Texas Instruments TPS54232
Vin (screw connector)	7 V - 24 V
Vin (header)	5 V
Compatibilty	MKR size

2.3 CAN BUS

Il *Controller Area Network*, noto anche come *CAN-bus*, è uno standard seriale per bus di campo, sviluppato da Robert Bosch nel 1986. L'esigenza di far comunicare i molti dispositivi elettronici presenti all'interno delle automobili (l'*Antilock Braking System* ABS, il *Traction Control System* TCS, il controllo del climatizzatore e la chiusura centralizzata sono solo alcuni esempi) e la complessità di questi, avrebbe portato ad un aumento insostenibile di collegamenti dedicati ed una duplicazione dei singoli sensori necessari a più dispositivi, con conseguente aumento dei costi di produzione e soprattutto notevole ingombro fisico. Per questi motivi è nato il *CAN-bus*, il quale consente a controllori, sensori e attuatori di comunicare l'uno con l'altro, offrendo anche bassi costi di progettazione e implementazione, immunità ai disturbi elettromagnetici, facilità di configurazione e modifica e rilevamento automatico degli errori di trasmissione.

Contrariamente a quanto accade nelle reti *Address-based*, in cui ogni computer ha il suo indirizzo di rete attraverso il quale un mittente invia ad uno specifico indirizzo (destinatario) un pacchetto, nel *CAN-bus* un singolo nodo (mittente) invia un messaggio sulla rete, quest'ultimo, invece di essere indirizzato ad uno specifico nodo destinatario, sarà ricevuto da tutti i nodi collegati, saranno poi quest'ultimi a decidere se il messaggio ricevuto può essere utile e quindi sfruttato per i propri compiti o meno. Questo consente ad un singolo sensore di un nodo di poter essere sfruttato da altri nodi, quindi senza la necessità che su quest'ultimi venga montato lo stesso sensore. Possiamo quindi considerare il *CAN-bus* un protocollo *Message-based* anziché *Address-based* e questa sua caratteristica offre diversi vantaggi in molti campi applicativi come quelli dell'*Automotive*: un radar anticollisione, rilevata la possibilità di pericolo, può inviare un messaggio di allarme sul *CAN-bus* in modo tale che altri nodi, ricevuta l'informazione, possano usufruirne. Per esempio: il sistema di frenata automatica, una volta ricevuto il messaggio dal radar anticollisione, agirà attivando automaticamente i freni, mentre l'*Engine Control Unit* (ECU) può agire chiudendo le valvole a farfalla del sistema di aspirazione e gli iniettori, in modo tale da inibire la funzione dell'acceleratore qualora il conducente lo mantenga premuto. Un altro vantaggio derivato dal *Message-based*, è il fatto che un nodo aggiuntivo può essere inserito nella rete senza il bisogno di riprogrammare le altre interfacce.

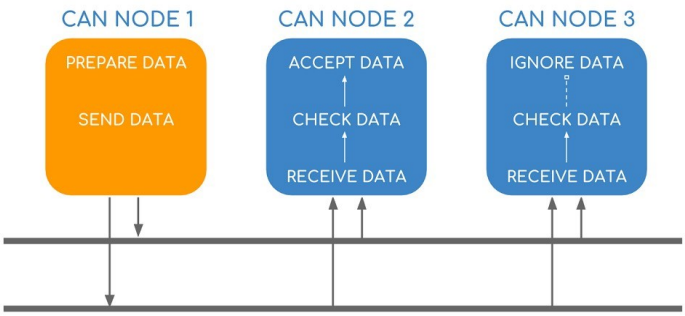


Figura 2.3.1 Struttura nodi CAN Fonte: *CSS Electronics-CAN BUS EXPLAINED (2019)*

2.3.1 Struttura dei Messaggi CAN

Il *CAN-bus* definisce 4 tipi di messaggi chiamati anche frame; essi possono avere due formati definiti dagli standard **CAN 2.0A** (*base frame*) e **CAN 2.0B** (*extended frame*): dove troviamo rispettivamente nel primo l'identificatore a 11 bit mentre nel secondo a 29bit. Il messaggio più comune è il *DATA-frame* ed è usato per trasmettere dati tra i nodi

- *Base frame:* con 11 bit di identificazione.

Figura 2.3.1.1 Struttura di un base frame CAN 2.0A Fonte: *CSS Electronics-CAN BUS EXPLAINED (2019)*

- *Extended frame:* con 29 bit di identificazione.

Figura 2.3.1.2 Struttura di un extended frame CAN 2.0B Fonte: *CSS Electronics-CAN BUS EXPLAINED (2019) modificata*

La struttura del *DATA-frame*, come possiamo vedere dalle due immagini è composta da 8 campi:

- ***Start Of Frame SOF*** - 1 bit
  - ✦ Indica l'inizio di un *frame*.

Campo arbitraggio:

- ***CAN-ID*** – 11 bit / 29bit
  - ✦ Contiene l'identificatore del *frame*, più il valore è basso, più è alta la priorità. ○ ***Remote Transmission Request RTR*** – 1bit
  - ✦ Indica se un nodo sta inviando dati (*DATA-frame*) o se sta richiedendo dati dedicati da un altro nodo (*REQUEST-frame*). Campo di controllo:
- ***Control*** – 6 bit
  - ✦ composto da 6 bit di cui gli ultimi 4 costituiscono il *Data Length Code DLC* che codifica il numero di *byte* di un dato contenuti nel messaggio (*0-8 byte*). Se si tratta di un *base-frame* (2.0A), allora il primo bit del campo di controllo è il bit di *Identifier Extention IDE*, che specifica il tipo di formato (*IDE*=0 standard (2.0A), *IDE*=1 esteso (2.0B)).
- ***Data*** – 0 / 64 bit
  - ✦ Contiene i *byte* che codificano l'informazione trasmessa con il messaggio (il dato). Il campo può avere un'estensione che varia da 0 a 64 bit (solo multipli di 8) in base a quanto specificato nel sottocampo DLC. I bit che compongono i singoli *byte* del campo vengono trasmessi partendo sempre dal più significativo.
- ***Cyclic Redundancy Check CRC*** – 16 bit
  - ✦ Viene utilizzato come sistema per il rilevamento degli errori di trasmissione ed è composto da 15 bit e da un bit recessivo: il *CRC Delimiter*, utilizzato per indicare, una volta terminata la trasmissione dei 15 bit, che il bus è pronto per gestire la fase di *acknowledgment* seguente.

- ***Acknowledge ACK*** – 2 bit
  - Durante *l'ACK* il nodo trasmittente invia un bit recessivo (1) e qualunque nodo della rete abbia ricevuto il messaggio in modo corretto risponde con un bit dominante (0); Il nodo trasmittente in questo modo riceve la conferma dell'avvenuta ricezione da parte di almeno un nodo e chiude la trasmissione con un bit recessivo che costituisce *l'ACK Delimiter.*
- ***End Of Frame EOF*** – 7 bit recessivi
  - Indica la fine del *frame*.

2.3.2 Arbitraggio del Bus

Il CAN è un protocollo *CSMA/CD* (*Carrier Sense Multiple Access/Collision Detection*), ciò significa che i nodi della rete devono monitorare il bus attendendo che questo si porti in stato di *Idle* prima di tentare di trasmettere un messaggio (*Carrier Sense*). Quando il bus è in *Idle* ogni nodo può tentare di inviare le proprie informazioni "impossessandosi" del bus stesso (*Multiple Access*). Se due nodi iniziano la trasmissione nello stesso momento questi rileveranno una "collisione" e si comporteranno di conseguenza (*Collision Detection*).

Per poter supportare un arbitraggio di tipo *non-destructive bitwise* un protocollo deve definire due stati del bus corrispondenti ai due livelli logici 0 e 1 in cui uno stato sia in grado logicamente ed elettricamente di sovrascrivere sempre l'altro; per questo motivo il livello logico che viene sovrascritto prende il nome di stato recessivo mentre l'altro è detto dominante. Il protocollo CAN definisce il livello logico 0 (livello basso di tensione) come dominante (*dominant* bit = d) ed il livello logico 1 (livello alto di tensione) come recessivo (recessive bit = r). I due *bus state* così definiti sono gestiti dai nodi sul bus con il meccanismo del *Wired-AND* con la conseguenza che il bit dominante (0 logico) sovrascrive sempre quello recessivo (1 logico), solo se tutti i nodi della rete trasmettono un bit recessivo quindi anche il bus si troverà in quello stato. Con questo meccanismo di pilotaggio ogni nodo della rete che inizia a trasmettere il messaggio è in grado di riconoscere, "leggendo" lo stato del bus, se altri nodi con identificatore a più elevata priorità hanno contemporaneamente iniziato una trasmissione. Se un nodo durante la trasmissione del campo arbitraggio invia un recessivo e legge il bus in stato dominante riconosce che c'è almeno un altro nodo che sta trasmettendo un messaggio a più elevata priorità e come conseguenza termina la trasmissione ed entra in modalità di "*listening*".

Un nodo che perde l'arbitraggio prova a ritrasmettere il proprio messaggio appena il bus si libera nuovamente. L'arbitraggio viene vinto dal nodo che trasmette il messaggio a più elevata priorità e cioè quello con l'identificatore più piccolo. Quindi il messaggio il cui contenuto nell'identificatore ha il numero binario più piccolo vince la contesa e continua a trasmette sul bus per primo. Per esempio, il numero binario a 4 bit "0011" è più piccolo del numero "0101", è quindi anche il primo che vincerà la contesa. Il messaggio con ID uguale a 0 è quello a priorità massima.

3. Progettazione e Sviluppo

3.1 Presentazione concettuale dell'Interfaccia

Come già anticipato, l'obiettivo del progetto realizzato durante l'esperienza di tirocinio era quello di realizzare un'interfaccia che permettesse di collegare una rete con *CAN-bus*, quindi classica se si parla di macchine agricole e AdP, con un pacchetto ROS e di realizzare una comunicazione bidirezionale. La realizzazione di questa interfaccia ha evidenziato le potenzialità di ROS e sono state gettate le basi per lo sviluppo futuro del progetto *AGROSBUS*. Prima di entrare nel dettaglio tecnico della realizzazione verranno presentati i vari livelli che lo compongono.

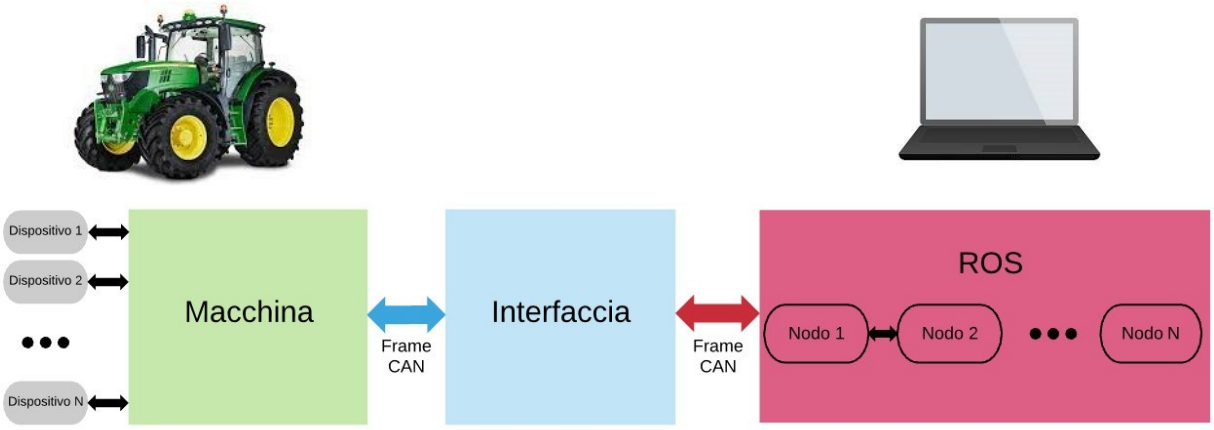


Figura 3.1.1 Struttura a livelli dell'interfaccia progettata Fonti: evo-tune.it, altroconsumo (rielaborata)

Livello Macchina: rappresenta l'astrazione della macchina agricola e del *CAN-bus* che ne collega tutti dispositivi. A bordo sarà presente della sensoristica dedicata ad esempio al controllo delle temperature motore, al controllo dello sterzo, all'analisi del terreno ecc. Qualunque proprietà della macchina che sia importante conoscere o controllare durante il suo funzionamento ha una sua sensoristica dedicata e verrà messa in comunicazione con ROS.

Livello ROS: acquisisce, elabora e fornisce risultati in merito ai dati provenienti dal *CAN-bus*. Su questo livello avranno sede tutti i Nodi necessari al controllo dei dispositivi della macchina e i loro collegamenti logici. Ciascun nodo svolge delle operazioni ben definite (approccio modulare), comunica con gli altri e coopera per realizzare controlli più complessi, che coinvolgano magari più variabili contemporaneamente. Esempio pratico: nella pubblica amministrazione, spesso una procedura può interessare diversi organi indipendenti come amministrazioni centrali, ministeri del tesoro e degli interni, amministrazioni periferiche dello stato, enti locali come i comuni e anche le aziende sanitarie, oppure che collegano unità periferiche con gli enti centrali della stessa amministrazione; allo stesso modo ad esempio un algoritmo di *obstacle-avoidance*[2] potrebbe coinvolgere più nodi ROS come ad esempio, il nodo che si occupa della geolocalizzazione, il radar per i controlli sulle distanze, un nodo-controller per la pedaliera e un nodo per il controllo dello sterzo.

L'interfaccia: si occuperà di trasferire dati in maniera bi-direzionale tra gli altri due livelli. Includerà delle procedure necessarie per andare ad intercettare i *frame* che viaggiano su *CAN-bus*. Questi ultimi, una volta intercettati, dovranno essere smistati verso il nodo appropriato sul livello ROS che svolgerà le dovute azioni di controllo sul frame. Allo stesso modo, un nodo che voglia comunicare risultati a della sensoristica sul *CAN-bus*, per realizzare ad esempio azioni correttive, produrrà dei dati e tramite l'interfaccia essi saranno smistati verso i la porzione di rete appropriata.

3.2 Approccio emulativo

Per migliorare l'efficienza e la rapidità dello sviluppo e non avendo purtroppo a disposizione un *CAN-bus* di una macchina agricola reale, si è scelto costruire l'interfaccia su un modello in scala di quella che potrebbe essere una situazione reale. Abbiamo incluso nel progetto della sensoristica Arduino che potesse emulare il comportamento di alcuni dispositivi a bordo di una macchina agricola. I sensori che abbiamo utilizzato sono i seguenti:

- **Radar ad ultrasuoni HC-RS04**

Dispone di 4 pin: *Vcc (+5V), Trigger, Echo, GND*.
Funzionamento: Si invia un impulso alto sul pin *Trigger* per almeno 10 microsecondi, a questo punto il sensore invierà il ping sonoro e aspetterà il ritorno delle onde riflesse, il sensore risponderà sul pin *Echo* con un impulso alto della durata corrispondente a quella di viaggio delle onde sonore.

Figura 3.2.1
Radar Ultrasonico HC-SR04 (vista frontale)
Fonte: Arduino Store

2 procedure che permettono a robot in movimento di evitare ostacoli lungo un percorso. Spesso vengono utilizzati insieme ad altre procedure come quelle per il path-finding o per l'obstacle-detection. Un esempio di algoritmo di obstacle-avoidance è i "The bubble rebound obstacle avoidance algorithm for mobile robots".

• **Sensore di temperatura e umidità DHT11**

In questo sensore il componente di rilevamento dell'umidità è un substrato di contenimento dell'umidità, con elettrodi applicati sulla superficie. Quando il vapore acqueo viene assorbito dal substrato, gli ioni vengono rilasciati dal substrato e quindi aumenta la conduttività tra gli elettrodi. Il cambiamento di resistenza tra i due elettrodi è proporzionale all'umidità relativa. Il DHT11 misura anche la temperatura con un sensore di temperatura NTC (termistore[3]) montato in superficie e incorporato nell'unità.

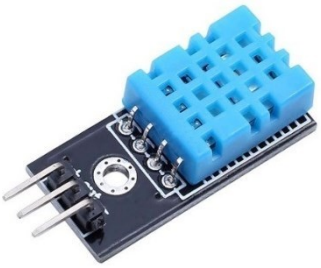


Figura 3.2.2 Sensore di umidità e temperatura DHT11
Fonte: Arduino

Per il collegamento di questi sensori utilizzeremo una piccola breadboard e una scheda ***MKR-Zero*** sulla quale incorporeremo anche uno dei due ***MKR Can-Shield*** che ci permetterà di includere tutte le caratteristiche del protocollo di comunicazione CAN. Di seguito la schematizzazione dei collegamenti:

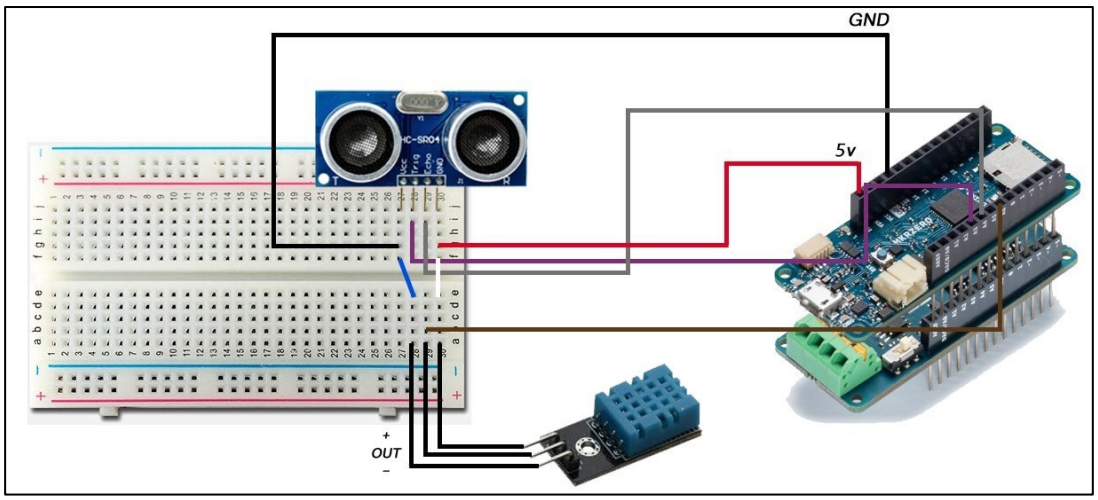


Figura 3.2.3 realizzazione del livello 'Macchina' dell'interfaccia, sulla sinistra i due sensori e sulla destra l'MKR-Zero montato al di sopra del MRK Can-Shield Fonte: Arduino Store, modificata

Con questo schema abbiamo ottenuto una realizzazione del livello macchina. Uno *sketch* Arduino, che vedremo più avanti, caricato sulla *MKR-Zero,* si occuperà di raccogliere i dati generati del radar HC-SR04 ed i dati relativi a temperatura e umidità del sensore DHT11. I valori letti verranno inseriti in differenti *Frame* CAN e verranno spediti, mediante il protocollo CAN, verso il secondo *CAN Shield*. D'ora in avanti faremo riferimento questa coppia di schede *MKR-Zero* e *CAN Shield* con il termine "***Sender".***

3. nell'elettronica ed in generale nell'automazione, è un resistore il cui valore di resistenza varia in maniera significativa con la temperatura. Il termine deriva dalla combinazione delle parole termico e resistore.

3.3 Livello intermedio dell'interfaccia

Per poter spedire i pacchetti CAN dal primo al secondo *Shield* andremo a collegarli insieme utilizzando gli appositi ingressi ***CAN-H*** *e* ***CAN-L*** (Fig. 2.2.3.2).
Di seguito il collegamento dei due *Shield*:

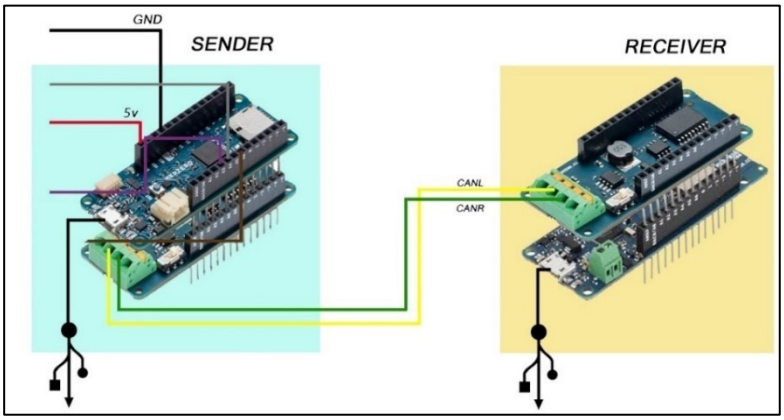


Figura 3.3.1 collegamento delle schede costituenti il Sender con il Receiver (MKR WAN 1300 + secondo MKR CAN Shield) Fonte: Arduino Store, modificata

Al di sotto del secondo ***CAN-Shield*** collegheremo la scheda ***MKR WAN 1300*** sulla quale caricheremo un secondo *sketch* di Arduino, che vedremo sempre più avanti. D'ora in avanti faremo riferimento a questa seconda coppia di schede con il termine "***Receiver***". Differentemente dalla scheda MKR-Zero, la *MKR WAN 1300* questa scheda dovrà comportarsi da *buffer*, sia per i pacchetti provenienti dal ***Sender***, sia per i pacchetti di risposta provenienti da ROS. La *MKR WAN 1300* predisporrà dunque delle zone di memoria che verranno utilizzate per compensare differenze di velocità tra la ricezione e la trasmissione di dati. Per realizzare il *buffer* abbiamo scelto come struttura dati una coppia di code circolari sulle quali andremo continuamente a memorizzare e a prelevare *Frame* CAN.

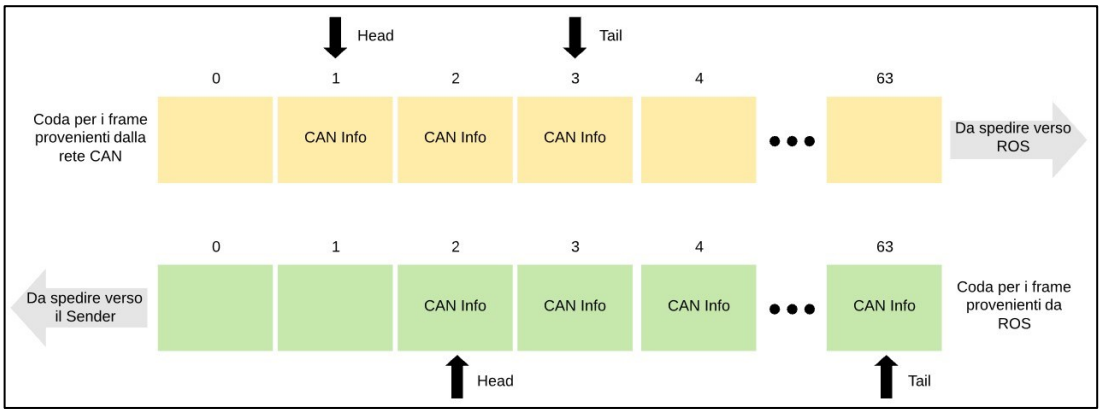


Figura 3.3.2 coppia di code circolari disposte dal Receiver per realizzare la funzione di buffer in entrambe le direzioni

3.4 Collegamento con ROS

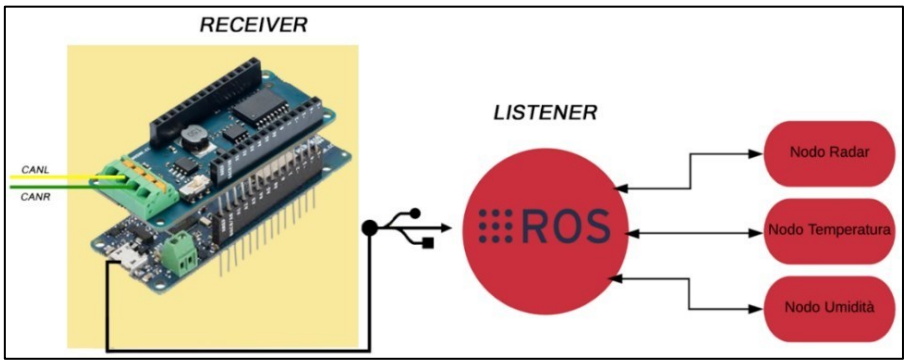


Figura 3.4.1 schema esplicativo del collegamento tra il Receiver e ROS Fonte: Arduino Store, modificata

Nel trasferimento dei *Frame* CAN verso ROS andremo ad includere delle informazioni aggiuntive come la lunghezza del pacchetto CAN che chiameremo ***payload*** e un campo che memorizzi l'istante temporale nel quale il pacchetto è sopraggiunto al ***Receiver*** (campo ***timestamp***). Il campo *payload* sarà un campo numerico tra 1-8 e indicherà la lunghezza della sezione "Data" del *Frame* CAN (si vedano Fig. 2.3.1.1 e Fig. 2.3.1.2). Per semplicità tratteremo solo Frame CAN con campo Data di lunghezza 16bit; si vuole far notare che tale semplificazione non si traduce in una costrizione per l'interfaccia ma andranno soltanto rivalutate le procedure che provvedono alla creazione dei CAN Frame che vogliamo ricordare avviene sul ***Sender***[4]. Di seguito un'immagine che riporta la struttura dell'informazione che verrà scambiata tra ROS e Arduino:

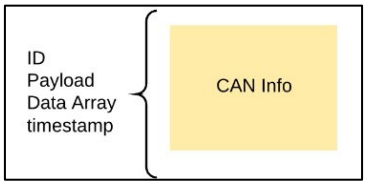


Figura 3.4.2 struttura dati di un singolo elemento Can Info

Anche in questo caso, volendo privilegiare una progettazione modulare, utilizzeremo un unico nodo ROS, che chiameremo ***Listener***, che si occuperà di smistare i pacchetti provenienti dal ***Receiver*** verso il nodo ROS corretto. Questa ridistribuzione dei pacchetti verrà effettuata sulla base del ID del *Frame* che servirà a identificare univocamente i dispositivi mittenti sul *CAN-bus*. In questo modo, qualora si volessero aggiungere ulteriori dispositivi alla rete CAN, sarà sufficiente aggiungere la libreria associata al nuovo dispositivo al ***Listener***, predisporre i collegamenti ed aggiungere eventuali librerie sul ***Sender*** e assicurarsi che non avvengano conflitti sulla generazione degli ID.

La comunicazione nel verso opposto a quello seguito fino ora avviene ripercorrendo gli stessi step ma in ordine inverso quindi non si ritiene necessario un ulteriore approfondimento. Nello sviluppo delle varie componenti il mio lavoro è stato concentrato sulla creazione del codice del *Sender* e quindi del suo interfacciamento a basso livello con i sensori e la parte del *Receiver* che permette di comunicare via CAN, mentre il mio collega si è occupato del completamento del *Receiver e* dei collegamenti con *ROS*.

4 Nel futuro sviluppo il Sender, rappresentando difatti un'emulazione di una rete CAN, verrà sostituito da dispositivi che permettano un collegamento diretto alle reti CAN reali. Un esempio di adattatore che permetterebbe di sostituire il Sender è il PCAN-USB, prodotto della Peak System.

3.5 Preparazione dell'ambiente di sviluppo

Prima di procedere con l'approfondimento degli script specifichiamo che il sistema operativo utilizzato durante lo sviluppo è *Ubuntu* nella sua versione 18.04.

L'IDE Arduino utilizzato è nella sua versione 1.8.6 e può essere direttamente recuperato dalla sua pagina ufficiale. Agli *sketch* andremo ad abbinare diverse librerie esterne che verranno discusse durante la presentazione degli script perché vengano contestualizzate al meglio.

La versione di ROS utilizzata è **Melodic**, che può essere installata seguendo la procedura indicata dalla pagina ufficiale di ROS [*Fonte: wiki.ros.org/melodic/Installation*]. Una volta installato ROS è necessario andare a predisporre un ambiente di lavoro *catkin* e creare un nuovo *ROS package* che nel nostro caso è stato chiamato ***"agrosbuspkg"***. Per le procedure di creazione e compilazione del pacchetto ROS appena creato si continui a far riferimento alla documentazione ufficiale.

3.4.1 Modulo rosserial_arduino

Usando il pacchetto **rosserial_arduino**, è possibile usare ***ROS*** direttamente con l'IDE Arduino. Il modulo rosserial fornisce un protocollo di comunicazione ***ROS*** che funziona su UART[1] di Arduino. Permette ad Arduino di essere un vero e proprio nodo ***ROS*** che può pubblicare ed abbonarsi direttamente ai messaggi ***ROS***. Questa procedura sarà necessaria solo per il ***Receiver*** che sarà a differenza del ***Sender*** a contatto diretto con ***ROS***. I collegamenti verso ***ROS*** sono implementati come libreria vera e propria importabile dall'IDE Arduino. Come tutte le altre librerie, ros_lib funziona inserendo la sua implementazione nella cartella librerie dello sketchbook.

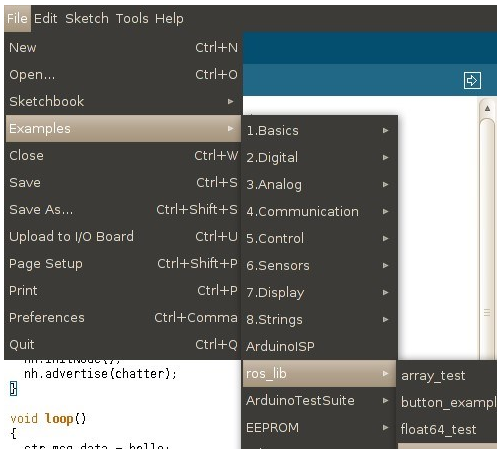


Figura 3.5.1.1 verifica del corretto collegamento di Arduino con ROS Fonte: rosserial_arduino/Tutorials/Arduino IDE Setup

[1] *Universal Asynchronous Receiver Transmitter*: è un dispositivo hardware che converte flussi di bit di una comunicazione parallela in un formato seriale asincrono (o viceversa).

3.4.2 Definizione di un custom message

Nella sezione 3.3 abbiamo descritto in maniera informale le informazioni che, oltre al campo Data e all'ID del *Frame* CAN, vorremmo trasferire verso ROS: ***payload*** e ***timestamp***. Queste variabili comporranno una struttura dati che verrà utilizzata sia dal *Receiver* che da ROS; nel caso del *Receiver* ci basterà definirla nello *sketch* mentre per ROS dovrà essere definita direttamente nel *package*. ROS mette a disposizione una serie di tipi dato standard con i quali, a seconda delle esigenze, possono essere aggregate informazioni eterogenee tra loro correlate. I tipi dato standard in ROS sono i seguenti:

Primitive Type	Serialization	C++	Python2	Python3
bool (1)	unsigned 8-bit int	uint8_t (2)	bool	
int8	signed 8-bit int	int8_t	int	
uint8	unsigned 8-bit int	uint8_t	int (3)	
int16	signed 16-bit int	int16_t	int	
uint16	unsigned 16-bit int	uint16_t	int	
int32	signed 32-bit int	int32_t	int	
uint32	unsigned 32-bit int	uint32_t	int	
int64	signed 64-bit int	int64_t	long	int
uint64	unsigned 64-bit int	uint64_t	long	int
float32	32-bit IEEE float	float	float	
float64	64-bit IEEE float	double	float	

Figura 3.5.2.1 Primitive di dato utilizzabili nei messaggi ROS Fonte: wiki.ros.org/msg

```
fieldtype1 fieldname1
fieldtype2 fieldname2
fieldtype3 fieldname3
```

Ad esempio:

```
int32 x
int32 y
```

Figura 3.5.2.2 esempio di custom message che rappresenti le coordinate di un punto in due dimensioni Fonte: wiki.ros.org/msg

Nel nostro caso, il tipo di messaggio verrà chiamato ***RosInfo*** e la sua definizione è la seguente:

```
uint32 id   uint8
length uint8[]
value uint64
timestamp
```

Inseriremo tale definizione in un *file .msg* in una cartella appositamente creata in ***agrosbuspkg***. Si vuole ricordare al lettore che ogni modifica alle cartelle, come può essere ad esempio questa definizione di messaggio, deve essere resa effettiva aggiornando il file CMakeLists.txt (aggiornamento delle dipendenze di pacchetto) e compilata utilizzando il comando ***catkin_make*** nel *catkin workspace.* Per maggiori informazioni sulla procedura si rimanda a *wiki.ros.org/rosmsg*.

Completata tale procedura tutti gli script del pacchetto ***agrosbuspkg*** potranno utilizzare il tipo di messaggio ***RosInfo*** semplicemente importandone l'header. Nel paragrafo 3.4.1 abbiamo sottolineato come il *Receiver* verrà inizializzato come vero e proprio nodo ROS; per permettere anche a quest'ultimo di importare la definizione di *RosInfo* andremo a generare l'header di ***RosInfo*** direttamente nella cartella contenente tutte le librerie di Arduino. Per farlo ROS mette a disposizione la seguente istruzione eseguibile da terminale:

```
> rosrun rosserial_client make_library.py path_to_libraries your_package
```

In questo caso:

```
> rosrun rosserial_client make_library.py /home/luigi/Arduino agrosbuspkg
```

Per verificare che la generazione dell'*header file* sia avvenuta correttamente è sufficiente controllare che all'interno di Arduino/libraries/ros_lib sia presente una cartella con il nome del pacchetto specificato nell'istruzione precedente. La cartella conterrà al suo interno ***RosInfo.h***, come in questo caso:

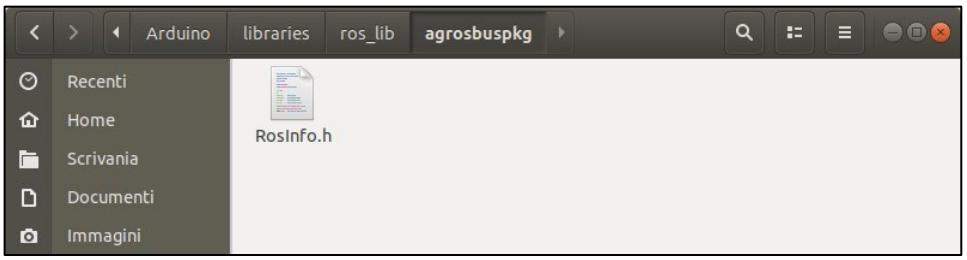


Figura 3.5.2.3 copia dell'header file del messaggio RosInfo tra le librerie di Arduino

A questo punto il setup dell'ambiente è completato ed è possibile andare a sviluppare il codice per le varie componenti partendo da quello per il ***Sender***.

3.5 Sviluppo dello sketch CanSender.ino

L'obiettivo di questo *sketch*, da me elaborato, è quello di leggere in maniera continua i dati generati dai sensori collegati agli ingressi I/O che abbiamo introdotto precedentemente, nel nostro caso, dal radar ad ultrasuoni (Fig. 3.2.1) e dal sensore di umidità e temperatura (Fig. 3.2.2), quindi prendere questi dati, inserirli in un frame CAN ed immette quest'ultimo sulla rete CAN mediante l'***MKR CAN Shield.***

Analizziamo nel dettaglio le parti del suo codice:

3.5.1 Librerie

```
//Main library_____________________________
#include <CAN.h>
//Sensors library_________________________
#include <DHT.h>                        // temperature and humidity sensor library
#include <DHT_U.h>                      // temperature and humidity sensor library
#include <HCSR04.h>                     //Ultrasonic distance sensor library
//________________________________________
```

Figura 3.5.1.1 importazione librerie

La libreria principale che permette al nostro Sender di comunicare con il CAN è Arduino-CAN (reperibile su Github), che permette, come ci suggerisce il nome, di poter pilotare il CAN Shield, le successive librerie sono quelle relative ai sensori che abbiamo collegato e che permettono ad Arduino di gestire i dati in ingresso generati dai vari sensori, quindi nel nostro caso saranno le

rispettive librerie per l'HC-SR04 e DHT11, nel caso in cui andassimo ad aggiungere nuovi o diversi sensori, basterà includere le rispettive librerie.

```
#define DHTPIN A2                                   //temperature and humidity sensor PINs
#define DHTTYPE    DHT11                            //temperature and humidity sensor type
DHT_Unified dht(DHTPIN, DHTTYPE);                   //Type and PIN
UltraSonicDistanceSensor distanceSensor(A4, A3);//Ultrasonic distance sensor used PIN
```

Figura 3.5.2.3 Settaggio parametri dei sensori

Come possiamo vedere in figura, nel nostro caso abbiamo dovuto definire alcune impostazioni necessarie per far funzionare i 2 sensori come i rispettivi PIN dell'MKR Zero dove i sensori sono collegati e nel caso del DHT11 la tipologia di sensore.

Per il DHT11 abbiamo utilizzato la libreria messa a disposizione da Adafruit (https://github.com/adafruit/DHT-sensor-library), mentre per l'HC-SR04 la libreria di Martinsos (https://github.com/Martinsos/arduino-lib-hc-sr04)

3.5.2 Sensori

3.5.2.1 Introduzione ai sensori

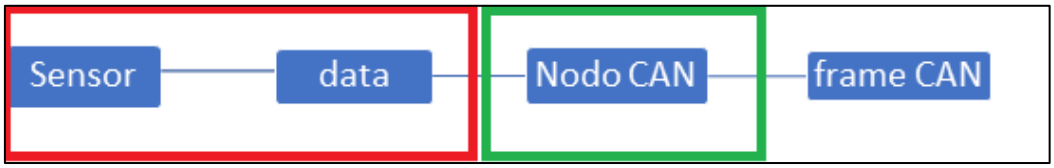


Figura 3.5.2.1

Parlando in generale, un sensore ha il compito di calcolare una grandezza fisica e di associarne un valore da utilizzare come output; questo output potrà poi essere sfruttato per compiere diverse operazioni.
Nel nostro progetto abbiamo come obbiettivo quello di prende l'output di questi sensori mediante le rispettive librerie, manipolare quando necessario il dato ed infine di generare dei frame CAN contenti i rispettivi valori dell'output ed un ID univoco da noi scelto, mediante la funzione *Sender,* che analizzeremo più avanti.
Con un po' di immaginazione possiamo dire ciò che facciamo non è altro che virtualizzare sensori reali, facendoli diventare di fatto dei nodi CAN virtuali che comunicano sulla rete con un proprio ID univoco.

Facendo riferimento alle immagini e guardando il codice possiamo osservare lo struct di un nodo virtuale

```
typedef union sensorS {
  unsigned char buf[2];
  unsigned short int value;
}sensorS;
```

Figura 3.5.2.2 Struct dati

Ogni nodo virtuale corrispondente ad un determinato dato, avrà una propria struttura che andrà a contenere i relativi dati, che andremo a definire:

```
sensorS Humidity;
sensorS Humidity_Ext;
sensorS Temperature;
sensorS Temperature_Ext;
sensorS Radar;
```

Figura 3.5.2.3 Generazione struct per ogni dato ricevuto dai sensori

3.5.4 Nodi CAN virtuali

3.5.4.1 Introduzione ai Nodi virtuali

Prima di analizzare i nodi virtuali utilizzati nel nostro progetto, li introduciamo in maniera generale prendendo come esempio i due sensori reali (rettangolo rosso nella figura) che abbiamo utilizzato nel nostro progetto (3.2), il DHT11 e HC-SR04; il concetto di base è che, poiché un sensore può generare N dati in uscita, possiamo creare un nodo virtuale per ogni N-esimo dato che riceviamo in input; nel nostro progetto, ad esempio, il DHT11, che è un unico sensore, genera come dati in uscita temperatura ed umidità; nel CanSender abbiamo preso questi due dati, e li abbiamo utilizzati per simulare più nodi CAN virtuali differenti, tutti con il proprio ID univoco, quindi sulla rete CAN risulteranno, non un nodo ma più nodi.

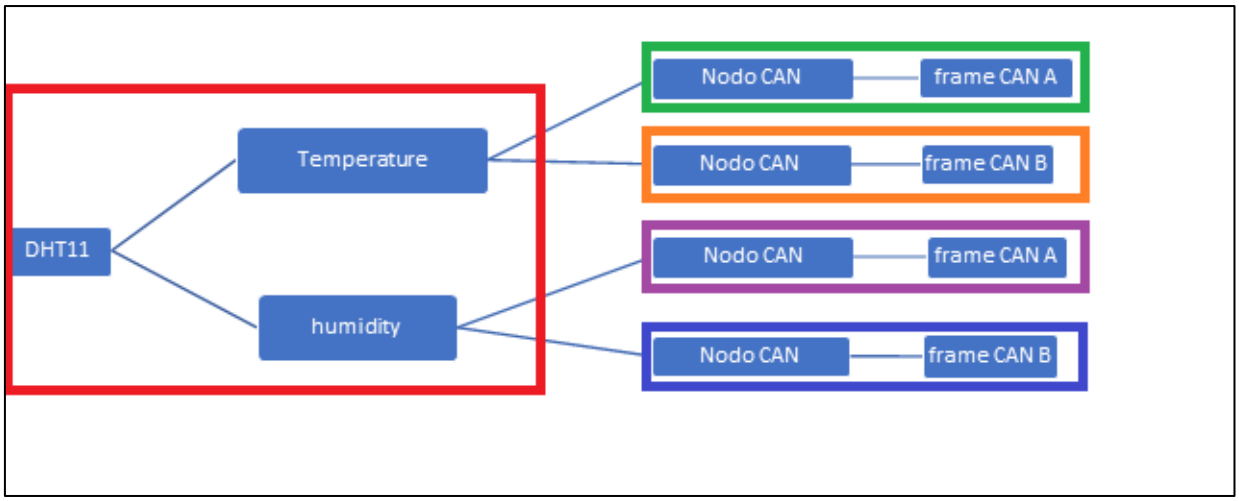


Figura 3.5.4.1

Osservando questo schema, abbiamo nel riquadro rosso il sensore reale che genera 2 dati in uscita; dato che siamo in grado di generare frame sia CAN A che CAN B, per ciascuno di questi dati

generiamo sia un frame CAN A che un frame CAN B, ognuno col proprio ID, quindi sulla rete risulteranno esserci ben quattro nodi CAN.

Questi nodi virtuali, che di fatto sono funzioni *void*, hanno tutte una forma comune che possiamo riassumere composta da 3 elementi principali (fig):

```
void sensor_name()
{
  data_sensor = data_sensor_acquisition;
  encodeUINT16(data_sensor, exponentiation of a 10, &Struct_sensor_name.value ;
  sender(id , Struct_sensor_name.buf, len) or extended_packet_sender(id , Struct_sensor_name.buf, len);
}
```

Figura 3.5.4.2

Osserviamoli nel dettaglio:

- 1) data_sensor_acquisition (acquisizione dati): l'obiettivo di questo primo elemento è quello di raccogliere i dati che ci interessano dal sensore; per fare ciò ci vengono incontro le funzioni messe a disposizione delle librerie relative ai rispettivi sensori; ricordiamo che un sensore può generare anche più dati come nel caso del DHT11, quindi dovremo sceglie quale, tra i dati in ingresso, prelevare.

- 2) data_sensor_convertion (conversione dati)

```
void encodeUINT16(float value, int digits, unsigned short int *valueDST){
  *valueDST = (unsigned short int)(value * pow(10,digits));
  return;
}
```

Figura 3.5.4.3

  Poiché i dati generati dai sensori possono essere in formato float, per semplificarne l'utilizzo ed ottimizzare lo spazio di memoria utilizzato nei frame CAN, mediante la funzione encodeUINT16 (3.5.4.3) moltiplichiamo il valore float per una potenza di 10 in modo tale da, eliminare la virgola prima di tutto, e mantenere una certa precisione del dato, sarà poi compito del receiver effettuare l'operazione inversa mediante la funzione decodeUNIT16, per riottenere il dato originale; questa potenza di 10 caratterizzerà il singolo sensore, quindi, se sul sender abbiamo scelto una potenza X per il sensore A, sul receiver dovrà essere utilizzata la medesima potenza X per effettuare il corretto decode del sensore A, ogni sensore avrà la propria "potenza" di codifica e decodifica.

- 3) sender o extended_packet_sender (invio dati): l'ultimo elemento che compone il nostro sensore virtuale si occupa di passare il dato convertito dal secondo elemento ed

inserito nello struct del sensore, insieme all'ID del sensore che scegliamo arbitrariamente e alla lunghezza del dato stesso.
Nel caso in cui questi dati (id, data, len) venissero passati alla funzione sender verrà generato un frame CAN 2.0 A, se invece venissero passati alla funzione extended_packet_sender verrà generato un frame CAN 2.0 B.

3.5.4.2 Nodi CAN virtuali nel nostro progetto

```
void temperature_sensor()
{
 sensors_event_t event;
 dht.temperature().getEvent(&event);
 Serial.print("Sending temperature data packet... ");
 if (isnan(event.temperature)) {
   Serial.println(F("Error reading temperature!"));
 }else {
 Serial.print(event.temperature);      //TEST
 encodeUINT16(event.temperature, 1, &Temperature.value);
 Serial.println(" ValueDST: ");
 Serial.print(Temperature.value);
 sender(0x12 , Temperature.buf, 2);
 Serial.println(" Temperature CAN_A sent");
 }
}
```

Figura 3.5.4.1

```
void extended_temperature_sensor()
{
 sensors_event_t event;
 dht.temperature().getEvent(&event);
 Serial.print("Sending temperature data packet... ");
 if (isnan(event.temperature)) {
   Serial.println(F("Error reading temperature!"));
 }else {
 Serial.print(event.temperature); //TEST
 encodeUINT16(event.temperature, 1, &Temperature_Ext.value);
 Serial.println(" ValueDST extended packet temperature: ");
 Serial.print(Temperature.value);
 extended_packet_sender(0x12 , Temperature.buf, 2);
 Serial.println(" Temperature data CAN_B sent");

 }
}
```

Figura 3.5.4.2

```
void humidity_sensor()
{
 sensors_event_t event;
 dht.humidity().getEvent(&event);
 Serial.print("Sending humidity data packet ");
 if (isnan(event.relative_humidity)) {
     Serial.println(F("Error reading humidity!"));
 }else {
 Serial.print(event.relative_humidity); //TEST
 encodeUINT16(event.relative_humidity, 1, &Humidity.value);
 Serial.println(" ValueDST Humidity: ");
 Serial.print(Humidity.value);
 sender(0x13 , Humidity.buf, 2);
 Serial.println(" Humidity data CAN_A sent ");

 }
}
```

Figura 3.5.4.3

```
void extended_humidity_sensor()
{
 sensors_event_t event;
 dht.humidity().getEvent(&event);
 Serial.print("Sending humidity data packet ");
 if (isnan(event.relative_humidity)) {
     Serial.println(F("Error reading humidity!"));
 }else {
 Serial.print(event.relative_humidity); //TEST
 encodeUINT16(event.relative_humidity, 1, &Humidity_Ext.value);
Serial.println(" ValueDST humidity: ");
 Serial.print(Temperature.value);
 extended_packet_sender(0x15 , Humidity_Ext.buf, 2);
 Serial.println(" Humidity data CAN_B sent");
 }
}
```

Figura 3.5.4.4

```
void anti_collision_radar()
{
 double distance = distanceSensor.measureDistanceCm();
 Serial.println(distance);
 encodeUINT16(distance, 2, &Radar.value);
 sender(0x14 , Radar.buf, 2);
 Serial.println(" Radar data CAN_A sent");
}
```

Figura 3.5.4.5

Nelle immagini qui riportate (3.5.4.1, 3.5.4.2, 3.5.4.3, 3.5.4.4, 3.5.4.5) possiamo osservare le funzioni relative ai nostri sensori virtuali *temperature_sensor*, *extended_temperature_sensor*, *humidity_sensor*, *extended_humidity_sensor* e anti_collision_radar, che sono tutte composte dai tre elementi suddetti; osserviamo che ogni sensore ha il suo ID univoco e nel caso dei 4 sensori virtuali associati al sensore

DHT11, per la conversione del dato da *float* a *int* abbiamo la stessa potenza di 10, mentre per il sensore virtuale associato al HC-SR04 la potenza di 10 è diversa.

Abbiamo così ottenuto 3 nodi CAN 2.0 A e 2 nodi CAN 2.0 B che inviano i rispettivi frame sul CAN-bus.

3.5.5 Funzione setup

```
void setup() {
  Serial.begin(9600);
  while (!Serial);
  //Sensors Setup
   dht.begin();
  sensor_t sensor;
  dht.temperature().getSensor(&sensor);
  dht.humidity().getSensor(&sensor);
  delayMS = sensor.min_delay / 1000;
  //Simulation type--------------------
  Serial.println("CAN Sender");
  Serial.println("Selected simulation type: ");
  Serial.print(simulation_type);
  if(simulation_type==0)
    Serial.println(" = Continuous sending of packages.");
  if(simulation_type==1)
    Serial.println(" = Send packets if sensor values change.");
  if(simulation_type==2)
    Serial.println("= Radarandom selected delay and sensor.");
  //----------------------------------------
  // start the CAN bus at 500 kbps
  if (!CAN.begin(500E3)) {
    Serial.println("Starting CAN failed!");
    while (1);
  }
```

Figura 3.5.5.1

La funzione `setup()` (3.5.5.1), in qualsiasi sketch Arduino, viene chiamata all'avvio. Il suo scopo è quello di inizializzare variabili, selezionare modalità dei pin, iniziare a usare le librerie e molto altro necessario per il corretto funzionamento del Sender

Possiamo osservare nell'immagine *(3.5.5.1)*:

- La funzione Serial.begin Che permette di osservare gli output generati per il debug su un computer collegato via cavo usb all'*MKR Zero*

Nel rettangolo rosso i settaggi necessari per prelevare i dati dai singoli sensori, in questo caso è stato necessario solo impostare il DHT11

Per rendere la simulazione della rete CAN più realistica o meno, sono state aggiunte delle opzioni che permettono di selezionare 3 modalità con cui il *Sender* può inviare i CAN frame, entreremo nel dettaglio più avanti.

Nel rettangolo blu andremo ad inizializzare il CAN Shield con il *bit rate* desiderato.

3.5.6 Sender

```
void sender(int id , unsigned char* data, unsigned short int len)
{
  int i=0;
  CAN.beginPacket(id);
  while(i<len)
  {
    CAN.write(data[i]);
    i++;
  }
  CAN.endPacket();
}
```

Figura 3.5.6.1

```
void extended_packet_sender(int id , unsigned char* data, unsigned short int len)
{
  int i=0;
  CAN.beginExtendedPacket(id);
  while(i<len)
  {
    CAN.write(data[i]);
    i++;
  }
  CAN.endPacket();
}
```

Figura 3.5.6.2

Analizzando il codice, noteremo due funzioni *sender* (3.5.6.1) e *extended_packet_sender* (3.5.6.2) che, se richiamate passando gli opportuni dati, si occuperanno di generare i pacchetti, rispettivamente, CAN A e CAN B e di inviarli sul CAN-bus.

Per comprendere al meglio come lavorano le due funzioni *Sender* riprendiamo le figura che abbiamo utilizzato per descrivere i CAN Frame.

Figura 3.5.6.3
Fonte: CSS Electronics-CAN BUS EXPLAINED (2019) modificata

Figura 3.5.6.4

Entrando nel dettaglio, partiamo dai dati che passeremo alle due funzioni, come possiamo vedere dalla figura seguente (3.5.6.5) , sono:

```
(int id , unsigned char* data, unsigned short int len)
```

Figura 3.5.6.5

- ***Id***: l'id che passiamo alla funzione *Sender* è univoco per ogni sensore, esso verrà sfruttato dai comando ***CAN.beginPacket,*** per i *Base frame, e* ***CAN.beginExtendedPacket*** per gli *Extended frame*, per generare la parte iniziale del frame CAN, nel rettangolo rosso, e l'id, da noi passato, verrà inserito nel campo CAN-ID cerchiato in verde; se non vengono passati i campi RTR e Control, sarà la funzione stessa a completarli per noi.
- ***Data***: il capo data rappresenti di fatto l'informazione che vogliamo inviare sul CAN-bus, ovvero il dato generato dal sensore, ogni frame può contenere 8 *bytes*, e nel nostro caso andremo a passare *byte* per *byte* il nostro dato mediante il comando ***CAN.write.***
- ***Len:*** ultimo ma non meno importate dato che passiamo è ***len*** che indica la lunghezza del dato che vogliamo inserire nel *frame* e che sfruttiamo, come possiamo vendere nella fig , nel *for.*

Infine una volta completati i due precedenti comandi *CAN.beginPacket* e *CAN.write* possiamo eseguire *CAN.endPacket* per generare la parte finale (rettangolo viola della fig.) del CAN frame e liberare così la rete CAN.

3.5.7 Callback

```
void receiverCallback(int packetSize){
  Serial.print("\n\n\n");
  Serial.print("ID: ");
  Serial.print(CAN.packetId());
  Serial.print("  LENGHT: ");
  Serial.print(packetSize);
  Serial.print("  VALUES: ");
  //get raw bytes
  for(int i=0; i<packetSize; i++){
    int l = CAN.read();
    Serial.print(l, DEC);
    Serial.print(" | ");
    if(l==0) digitalWrite(LED_BUILTIN, LOW);
    if(l==1) digitalWrite(LED_BUILTIN, HIGH);
  }
}
```

Figura 3.5.7.1

Poiché l'obiettivo del nostro progetto è una comunicazione bidirezionale tra ROS e CAN, è stata implementata una funzione nel Sender, il `receiverCallback()` (3.5.7.1) che permette allo stesso di ricevere eventuali frame da parte di ROS. Analizzeremo meglio questa funzione dopo aver introdotto il lavoro del mio collega su ROS.

3.5.8 Modalità di simulazione

Avendo introdotto i sensori virtuali, possiamo ora parlare della modalità di simulazione, ovvero come sfruttarli per simulare una rete CAN.
Come avevamo accennato in precedenza, riprendiamo le 3 modalità di simulazione entrandone nel dettaglio.

```
//--SIMULATION TYPE-------------------------+
int simulation_type=0;//                    |
//  0= continuous sending of packages       |
//  1= send packets if sensor values change |
//  2= random selected delay and sensor     |
//-----------------------------------------+
```

Figura 3.5.8.1

```
//Simulation type--------------------
Serial.println("CAN Sender");
Serial.println("Selected simulation type: ");
Serial.print(simulation_type);
if(simulation_type==0)
  Serial.println(" = Continuous sending of packages.");
if(simulation_type==1)
  Serial.println(" = Send packets if sensor values change.");
if(simulation_type==2)
  Serial.println("= Radarandom selected delay and sensor.");
//----------------------------------------
```

Figura 3.5.8.2

Nella parte iniziale del codice troviamo *simulation_type* (3.5.8.1) dove dobbiamo assegnare un numero che va da 0 a 2, in base alla tipologia di simulazione che desideriamo avere.

- simulation_type==0: (3.5.8.2) in questa tipologia di simulazione i frame CAN vengono inviati continuamente con un delay di mezzo secondo tra un frame e l'altro, è possibile aggiungere ulteriori sensori andando semplicemente a richiamare la funzione del sensore (3.5.8.3) seguito da un ulteriore delay(500)

```
if(simulation_type==0)
{
  temperature_sensor();
  delay(500);
  humidity_sensor();
  delay(500);
  extended_humidity_sensor();
  delay(500);
  anti_collision_radar();
  delay(500);
}
```

Figura 3.5.8.3

- simulation_type==1 : (3.5.8.4, 3.5.8.5) con questa tipologia di simulazione vengono inviati sul CAN soltanto i frame dei sensori che cambiano il proprio dato: se ad esempio viene inviato un frame dal nodo virtuale `temperature_sensor()` con certo valore, questo nodo invierà ulteriori frame solo se il dato relativo alla temperatura dovesse cambiare.

```
if(simulation_type==1)
{
    send_if_new_data_are_changed();

}
```

Figura 3.5.8.4

```
void send_if_new_data_are_changed()
{
  sensors_event_t event;
  dht.temperature().getEvent(&event);
  dht.humidity().getEvent(&event);

  if(temperature_value!=event.temperature)
    {
      temperature_sensor();
      temperature_value=event.temperature;
    }

  if(humidity_value!=event.relative_humidity)
  {
    humidity_sensor();
    humidity_value=event.relative_humidity;
  }
  if(radar_value!=distanceSensor.measureDistanceCm())
  {
    anti_collision_radar();
    radar_value=distanceSensor.measureDistanceCm();
  }
}
```

Figura 3.5.8.5

- simulation_type==2 : (3.5.8.6) con questa tipologia di simulazione i frame vengono inviati sul CAN con un ordine ed un delay del tutto casuale, sfruttando la funzione rand() (rettangolo rosso, 3.5.8.7), da notare che il case 5: e il case 6: (rettangolo verde, 3.5.8.8) sono volutamente lasciati senza richiamare sensori, per simulare momenti di durata casuale in cui il CAN è libero.

```
if(simulation_type==2)
  random_();
```

Figura 3.5.8.6

```
int random_()
{
  int  v1 = rand() % 3;
  int  v2 = rand() % 6;
  int delay_time[3]={1000,500,20};
  int delay_=delay_time[v1];
  switch(v2)
  {
    case 1:
  {
    extended_temperature_sensor();
    delay(delay_);
  }
  case 2:
   {
    humidity_sensor();
    delay(delay_);
   }
```

Figura 3.5.8.7

```
  case 3:
  {
      anti_collision_radar();
      delay(delay_);
    }
  case 4:
    {
      extended_humidity_sensor();
      delay(delay_);
    }
  case 5:
    {
      delay(delay_);
    }
  case 6:
  {
    delay(delay_);
  }
}
}
```

Figura 3.5.8.8

3.5.9 Funzione loop

Per finire, come ultima funzione caratteristica di Arduino, troviamo il `void loop()` (3.5.9.1) che esegue, come dice il nome stesso della funzione, in loop le varie operazioni per la simulazione sulla base della tipologia di simulazione scelta

```
void loop() {

      if(simulation_type>2)
      {
        Serial.println(" =Selected simulation type wrong");
        delay(10000);
      }
  if(simulation_type==0)
  {
    temperature_sensor();
    delay(500);
    humidity_sensor();
    delay(500);
    extended_humidity_sensor();
    delay(500);
    anti_collision_radar();
    delay(500);
  }
  if(simulation_type==1)
  {
      send_if_new_data_are_changed();

  }
  if(simulation_type==2)
    random_();

  Serial.println("All done!");
  delay(1000);
}
```

Figura 3.5.9.1

3.6 Sviluppo CanReceiver e ROS

Come specificato al termine del paragrafo 3.5, lo sviluppo dello sketch per il ***Receiver*** e ***ROS*** è spettato al mio collega quindi di seguito verranno descritte brevemente le sue parti principali, omettendo volontariamente i dettagli e rimandando il lettore al suo elaborato per un eventuale approfondimento.

3.6.1 Sketch CanReceiver.ino

```
#define USE_USBCON
//libraries for ROS communication
#include <ros.h>
#include <agrosbuspkg/RosInfo.h>

#include <mcp_can.h>
#include <mcp_can_dfs.h>
#include <CAN.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

//for sync with actual system time
#include <Time.h>
```

Figura 3.6.1.1

Nello sketch del *Receiver*, oltre alle librerie CAN, presenti anche nel Sender, troviamo librerie che permettono ad Arduino di interfacciarsi con ROS, **`<ros.h>`** e la libreria **`<Time.h>`** invece ci servirà per ricavare il *timestamp* di sistema; generalmente questo, viene ricavato utilizzando un *Real Time Clock (RTC)* su chip integrato, poiché Arduino ne è sprovvisto e non avendo noi a disposizione nessun RTC aggiuntivo (3.6.1.2) da collegare, che di fatto risolverebbe questa mancanza, è stato deciso di ricorrere ad un messaggio di sync (3.6.1.3) costituito dall'attuale *timestamp* richiesto dal monitor seriale all'avvio.

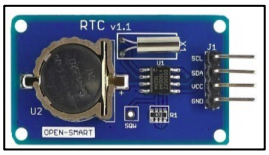


Figura 3.6.1.2 vista dall'alto del Real Time Clock DS1307
Fonte: Arduino Store

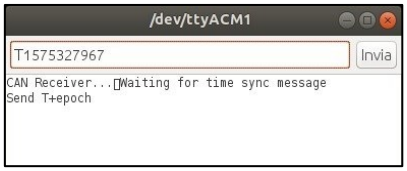


Figura 3.6.1.3 Input Epoch da seriale

Figura 3.6.1.4 sito dal quale è possibile reperire l'Epoch time
https://www.epochconverter.com/

```
void processSyncMessage(){
  unsigned long pctime;
  const unsigned long DEFAULT_TIME = 1357041600; // Jan 1 2013

  if(Serial.find(TIME_HEADER)) {
     pctime = Serial.parseInt();
     if(pctime >= DEFAULT_TIME) { // check the integer is a valid time (greater than Jan 1 2013)
       setTime(pctime); // Sync Arduino clock to the time received on the serial port
       epoch_at_start = pctime;
     }
  }
}

time_t requestSync()
{
  Serial.write(TIME_REQUEST);
  return 0; // the time will be sent later in response to serial mesg
}
```

Figura *3.6.1.5* funzioni che realizzano il sync con il Linux epoch

Se non viene inserito nessun epoch[1] da monitor seriale, oppure viene inserito un epoch non valido viene considerata di default la data del 1° Gennaio 2013. Queste due procedure verranno definite durante la fase di *setup* dello *sketch*.

3.7.1.1 Struttura dati

La struttura dati sarà una coda circolare quindi per realizzarla utilizzeremo due *array circolari*, ognuno dei quali utilizzerà due puntatori: uno all'elemento di testa e l'altro all'elemento di coda. Per definire la struttura di un elemento nella coda in Arduino, è possibile utilizzare l'operatore *struct:*

```
typedef struct raw_data{
   unsigned char * value;
   unsigned int id;
   unsigned char len;
   unsigned long long timestamp;
} raw_data;

unsigned char headR = 0;
unsigned char tailR = 0;
unsigned int countR = 0;
raw_data buff_toROS[BUFFER_LENGHT];

unsigned char headC = 0;
unsigned char tailC = 0;
unsigned int countC = 0;
raw_data buff_toCAN[BUFFER_LENGHT];
```

Figura 3.7.1.1.1 strutture che realizzano le code circolari del Receiver

le due code ***buff_toROS*** e ***buff_toCAN*** agiranno come buffer: permetteranno quindi di compensare diverse velocità di ricezione e trasferimento tra la rete CAN e la parte ROS.

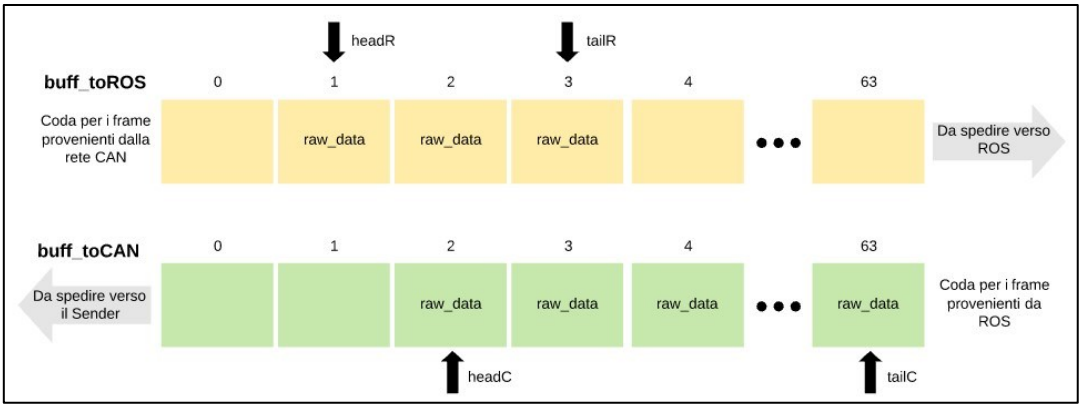


Figura 3.7.1.1.2 con riferimento alla figura *3.2.2* dettagliamo ulteriormente la struttura delle due code circolari

[1] Il tempo Unix (noto anche come tempo di epoca (epoch), tempo di POSIX, secondi dall'epoca o tempo di epoca di UNIX) è un sistema per descrivere un punto nel tempo. È il numero di secondi trascorsi dall'epoca di Unix, ovvero il tempo 00:00:00 UTC del 1 ° gennaio 1970, meno i secondi bisestili.

Mediante il modulo *rosserial,*il *Receiver* diventa un'istanza di un nodo ROS.

```
ros::NodeHandle nh;
agrosbuspkg::RosInfo ros_pkt;

//define where to publish
ros::Publisher fromarduino("fromarduino", &ros_pkt);
//setting subscription to fromros topic
ros::Subscriber<agrosbuspkg::RosInfo> fromros("fromros", &AddOneToCANtail);
```

Figura 3.7.7 definizione del ROS Handler e dichiarazione dei topic per la comunicazione con ROS

I *topic* di comunicazione con il ***Listener*** saranno dunque due: 'fromarduino' per strutture *raw_data* spedite dal *Receiver* verso il *Listener* e 'fromros' per strutture *raw_data* provenienti dal *Listener*.

3.7.1.2 Setup e main loop

```
void setup() {

  //start the serial communication
  Serial.begin(115200);
  while (!Serial); //wait until ready

  //Setting the CAN connection
  Serial.print("CAN Receiver...");
  CAN.begin(500E3);
  // start the CAN bus at 500 kbps
  if (!CAN.begin(500E3)){
      Serial.println("Starting CAN 500Kbps failed!");
      Serial.println("Trying other CAN speed:");
      set_can_speed(); //if 500 Kbps not work
  }
  setSyncProvider(requestSync);  //set function to callback when sync required
  Serial.println("Waiting for time sync message");
  Serial.println("Send T+epoch");

  //wait until user not send the time info
  while(!Serial.available());
  //set the Arduino time based on message received
  //wait for info about epoch
  processSyncMessage();

  //inizializzazioneNodeHandle
  nh.initNode();
  nh.advertise(fromarduino);
  //nh.advertise(uint8test); //testiamo se perdiamo pacchetti
  nh.subscribe(fromros);

  epoch_at_start = epoch_at_start * pow(10,3);

  //Set call back function for each CAN packet received
  //Each time a CAN packet comes add_one() is called
  CAN.onReceive(AddOneToROStail);
}
```

*Figura 3.7.*1.2.1 *definizione del ROS Handler e dichiarazione dei topic per la comunicazione con ROS*

Nel *main loop* invece non faremo altro che andare a controllare ad ogni ciclo se ci sono differenze tra i puntatori di testa e coda dei due buffer definiti in precedenza. Se esiste una differenza tra i due puntatori vuol dire che sono sopraggiunti dei *Frame* CAN al *Receiver* che, in attesa di essere inviati, vengono memorizzati nei buffer in strutture di tipo *raw_data* (*buff_toROS* se proveniente dal *Sender, buff_toCAN* se proveniente da ROS). I buffer vengono ripuliti dai vari *raw_data* fin quando i due puntatori non si trovano nuovamente sullo stesso elemento (condizione di *buffer* vuoto). Le funzioni che prelevano dai *buffer* e inviano i *raw_data* verso ROS o verso il *Sender* sono rispettivamente `PublishOneToROS` e `PublishOneToCAN`:

```
void PublishOneToCAN(){

  CAN.beginPacket(buff_toCAN[tailC].id);
  int l = buff_toCAN[tailC].len;
  int i=0;

  for(i=0; i<l; i++){
    CAN.write(buff_toCAN[tailC].value[i]);
  }
  CAN.endPacket();

  free(buff_toCAN[tailC].value);

  tailC = tailC + 1;
  tailC = tailC%BUFFER_LENGHT;
  countC = countC-1;

}
```

```
void loop() {
   nh.spinOnce();

   //cleaning the toROS buffer
   while(headR != tailR){
     PublishOneToROS();
   }
   //cleaning the toCAN buffer

   while(headC != tailC){
     PublishOneToCAN();
   }
}
```

Figura 3.7.1.2.2 Receiver main loop

```
//TO ROS TOPIC /fromarduino
void PublishOneToROS(){

    ros_pkt.id = buff_toROS[headR].id;
    ros_pkt.timestamp = buff_toROS[headR].timestamp;
    l = buff_toROS[headR].len;
    ros_pkt.lenght = l;

    //define lenght and assign pointer
    ros_pkt.value_length = l;
    ros_pkt.value = buff_toROS[headR].value;

    tailR = tailR + 1;
    tailR = tailR%BUFFER_LENGHT;
    countR = countR - 1;

    fromarduino.publish(&ros_pkt);

    free(buff_toROS[headR].value);
    free(ros_pkt.value);

}
```

Figura 3.7.1.2.3 funzioni che inviano raw_data verso ROS o verso il Sender prelevando dai buffer

3.7.1.3 Callbacks

In ultimo si vogliono discutere in breve le due *callback* definite per la ricezione dei Frame CAN da entrambe le direzioni:

```
//ARDUINO -> ROS
void AddOneToROStail(int packetSize){

  int i = 0;

  if(countR == BUFFER_LENGHT){
    Serial.print("Overrun from CAN..losing packets");
    return;
  }

  buff_toROS[headR].id = CAN.packetId();
  buff_toROS[headR].len = packetSize;
  buff_toROS[headR].timestamp = epoch_at_start + millis();

  //get the raw bytes
  buff_toROS[headR].value = new unsigned char[packetSize];
  i=0;
  while (CAN.available()) {
        buff_toROS[headR].value[i] = CAN.read();
        i++;
  }

  headR = headR + 1;
  headR = headR%BUFFER_LENGHT; //make sure head [0 % BUFFER_LENGHT-1]
  countR = countR + 1; //new alement in the queue
}
```

```
//ROS -> ARDUINO
void AddOneToCANtail(const agrosbuspkg::RosInfo& can_pkt){

  int i = 0;

  //before adding check if is full
  if(countC == BUFFER_LENGHT){
    Serial.print("Overrun from ROS..losing packets");
    return;
  }

  buff_toCAN[headC].id = can_pkt.id;
  buff_toCAN[headC].len = can_pkt.lenght;
  buff_toCAN[headC].timestamp = can_pkt.timestamp;

  //getting the raw bytes
  buff_toCAN[headC].value = new unsigned char[can_pkt.lenght];
  for(i=0; i<can_pkt.lenght; i++){
    buff_toCAN[headC].value[i] = can_pkt.value[i];
  }

  headC = headC + 1;
  headC = headC%BUFFER_LENGHT; //make sure head [0 % BUFFER_LENGHT-1]
  countC = countC + 1; //new alement in the queue

}
```

Figura 3.7.1.3.1 callback per la ricezione di raw_data implementate sul Receiver

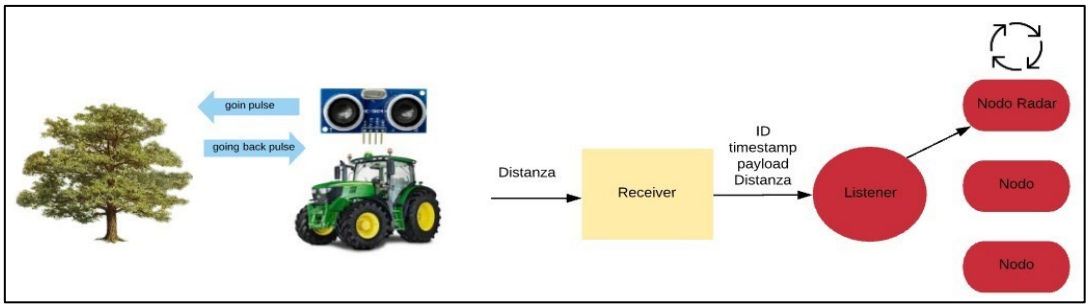
In particolare, si vuole far notare che nella costruzione di un *raw_data* da aggiungere a *buff_toROS*, la generazione del *timestamp* viene realizzata andando ad aggiungere al `epoch_start` (inizializzato nel `setup`) il numero di millisecondi trascorsi dall'avvio dello sketch tramite la funzione **`millis()`** della libreria `<Time.h>`.

3.8 Descrizione dello script Listener.py

Lo script *Listener.py* ha il compito di andare a smistare i pacchetti in arrivo, ad esempio un `raw_data` appartenente ad un certo ID, dal *Receiver* verso i nodi competenti. Un *raw_data* contenente informazioni sul valore rilevato dal radar ad ultrasuoni HCSR04 verrà inviato dal *Listener.py* verso un secondo script *radar.py* che consumerà quelle informazioni.

Allo stesso modo il *Listener.py* dovrà permettere la comunicazione nella direzione opposta permettendo quindi agli script dei relativi sensori di rispondere con dei messaggi verso il *Recever*. In sostanza quindi *Listener.py* agirà come vero e proprio ponte tra il *Receiver* e gli script che realizzano la logica dei dispositivi sulla macchina agricola.

Ricezione informazioni



Risposta

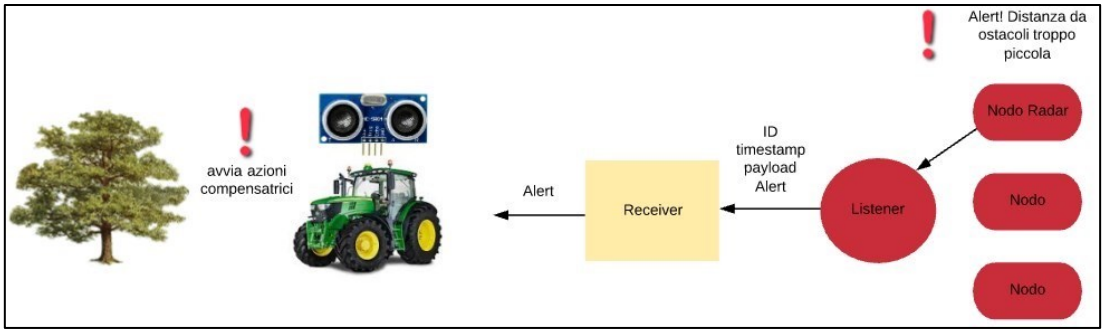


Figura 3.8.1 e 3.8.2: percorso di una comunicazione tra un dispositivo e il nodo contenente la sua logica (lato ROS).

3.8.1 Analisi del codice

Dopo aver importato le librerie è necessario importare le definizioni dei vari dispositivi. Ciascuno di essi è descritto nelle librerie tramite una classe;

```
15  #library import + instance creation
16  #RADAR HCSR04
17  from radar import Radar
18  radarHCSR04 = Radar("hcsr04_topic",2,10,pub)
19
20  #Temp/HUM DHT11
21  from tempsensor import Temperature
22  DHT11 = Temperature("dht11_topic",1,10)
23
24  from humsensor import Humidity
25  DHT11_hum = Temperature("dht11_hum_topic",1,10)
```

Figura 3.8.1.3 inclusione delle classi dei diversi dispositivi nello script Listener.py

Questo approccio permette di mantenere una discreta modularità della parte ROS, difatti per andare ad integrare nuovi sensori o dispositivi sarà sufficiente includerne la classe in *Listener.py* e crearne una istanza. Di seguito la definizione delle varie callback da richiamare quando un *raw_data* sopraggiunge al *Listener*:

```
#Callback for incoming packets
def callback(data):
    #printing info about packet that has just arrived
    logfunction(data.id, data.timestamp, data.lenght, data.value,0)

    #call appropriate instace depending on IDs
    if data.id == 20:
        Radar.sendpacket(radarHCSR04, data.value)

    elif data.id == 21:
        Humidity.sendpacket(DHT11_hum, data.value)

    elif data.id == 18:
        Temperature.sendpacket(DHT11, data.value)
```

Figura 3.8.1.4 callback per la ricezione: a seconda dell'id viene richiamata la callback del sensore ad esso associata

Nel richiamare le funzioni **`sendpacket`** per ciascuna classe si passano come parametri le istanze dei vari sensori e la parte di *raw_data* che contenga solo informazioni sui valori rilevati dai sensori quindi `data.value` (*array di byte*). Le funzioni **`sendpacket`** si occupano di convertire gli *array di byte* nei corrispondenti valori reali e di inizializzare un nuovo *topic* a cui effettueranno la *Subscription* i nodi che compongono la logica del dispositivo. Di seguito, a titolo di esempio, la definizione della classe ***Radar***:

```
def radar_callback(data):
    #when the radar logic send back something than
    #this callback decide how to deal with it and
    #eventually it can create a RosInfo() to send
    #a packet back to the Ros Network
    packet_backToCan = RosInfo()
    pub.publish(pacchetto_backtoCan)
    pass

class Radar:
    #defining attributes for each sensor, a RosInfo to send things back
    #to arduino and two pointers (topic) to communicate with sensor logic
    #script
    def __init__(self, topic_name, pow, q_size, fromros_pointer):
        self.topic_name = topic_name
        self.pow_value = pow
        self.queue_size = q_size
        self.radar_ToCANpkt = RosInfo() #packet to fill and sand back to arduino from top

        self.pub_radar = rospy.Publisher(topic_name, Float32,queue_size=self.queue_size)
        #depending on witch data type should turn back from sensor logic
        rospy.Subscriber(topic_name + "_back", Float32, radar_callback) #for outgoing pac

    #when a packet for the sensor arrive from the topic "fromarduino"
    #this method stars and send immediatly to the sensor logic
    def sendpacket(self, sensor_value_tuple):
        value_to_send = self.convert(sensor_value_tuple)
        self.pub_radar.publish(value_to_send)

    #each sensor have to provide a proper conversion function
    def convert(self,tup):
        bytes = array('b', tup)
        value = struct.unpack('<h', bytes)
        #return the right value with the right pow conversion
        return (value[0] / pow(10, self.pow_value))
```

Figura 3.8.1.5 classe Radar

4. Risultati

In questa sezione conclusiva dell'elaborato andiamo ad esaminare i risultati dell'interfaccia progettata analizzando il comportamento dei vari elementi che la compongono. Per ricostruire lo schema circuitale si rimanda il lettore ai paragrafi 3.1, 3.2, 3.3 e 3.4 per una descrizione più dettagliata. È possibile dunque lanciare l'IDE di Arduino e caricare sulla scheda *MKR-Zero* lo *sketch* CanSender.ino. In questo modo il *Sender* andrà in esecuzione e inizierà ad inviare *Frame* CAN verso la scheda *MKR-WAN 1300* (*Receiver*). Di seguito un'immagine che mostra il monitor seriale della scheda *MKR-Zero* che in questo caso è collegata alla porta seriale **'dev/ttyACM0'**.

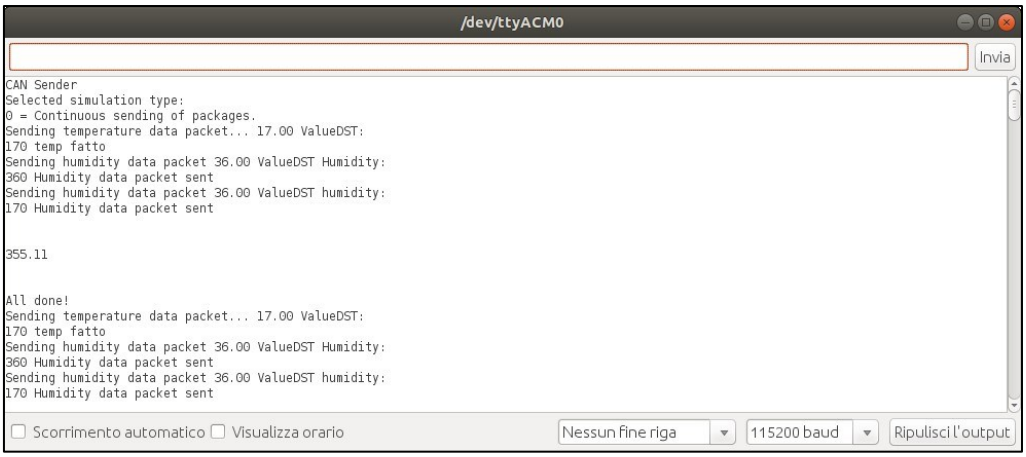


Figura 4.1 Monitor seriale del Sender in esecuzione

Per inizializzare il *Receiver* andremo a caricare lo *sketch* CanRecevier.ino sulla scheda *MKR-WAN 1300* che nel nostro caso è collegata alla porta seriale 'dev'. Una volta completato il caricamento dello sketch sarà necessario aprire il monitor seriale per inserire il messaggio di sync contente l'attuale *epoch*. Il *Receiver* rimane comunque in attesa del messaggio di sync e, nel caso venga inserito un messaggio scorretto si utilizzerà come riferimento la data del 1° Gennaio 2013. Per conoscere l'epoch attuale è possibile lanciare da linea di comando la seguente istruzione:

```
> date +%s
```

Figura 4.2 Monitor seriale del Receiver in attesa del messaggio di sync

In questo caso il messaggio di sync sarà **'T1575327967'**

Non appena ricevuto il messaggio di sync il ***Receiver*** inizierà a riempire il *buffer* che conserva i *raw_data* da inviare verso ROS e proverà ad inviarli, tuttavia senza eseguire ulteriori operazioni il *Receiver* non è in grado di comunicare con ROS e quindi di apparire come nodo al *Ros-Master*. Con il *Receiver* in esecuzione avviamo il ROS-Master da terminale utilizzando il comando:

```
> roscore
```

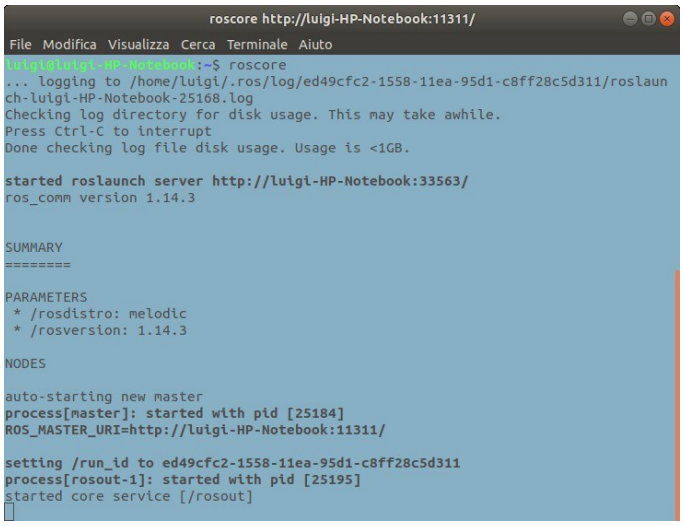


Figura 4.3

per permettere al *Receiver* di essere riconosciuto come nodo dal Ros-Master eseguiamo la seguente istruzione da terminale:

```
> rosrun rosserial_python serial_node.py port:=<posta_seriale> baud:=<velocità_di_comunicazione_seriale>
```

In questo caso quindi:

```
> rosrun rosserial_python serial_node.py _port:=/dev/ttyACM1_baud:=115200
```

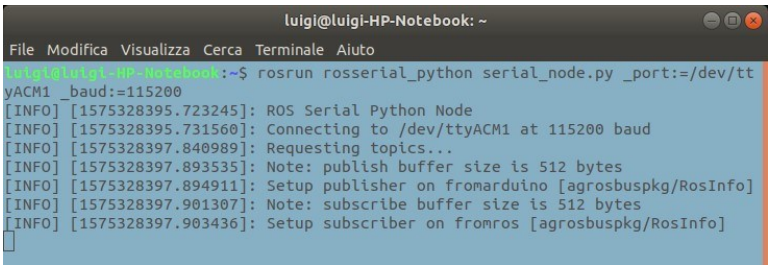


Figura 4.4

Per verificare che stiano effettivamente arrivando dati da Arduino possiamo utilizzare le funzioni messe a disposizione da ROS per lavorare con i *topic*. Ad esempio, utilizzando **`rostopic echo`** è possibile andare a stampare su terminale i dati in transito su un determinato *topic*, nel nostro caso effettuiamo una *echo* sul *topic* 'fromarduino':

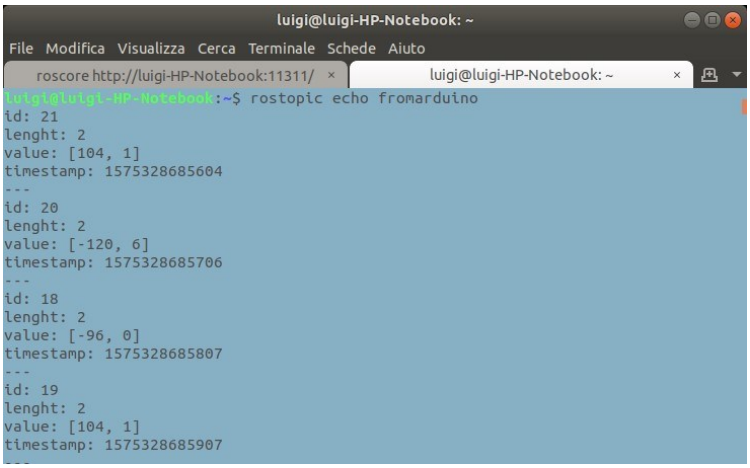


Figura 4.5

Per lanciare lo script *Listener.py* utilizziamo il comando **`rosrun`**. Di seguito la sintassi:

```
> rosrun <package> <executable>

> rosrun agrosbuspkg listener.py
```

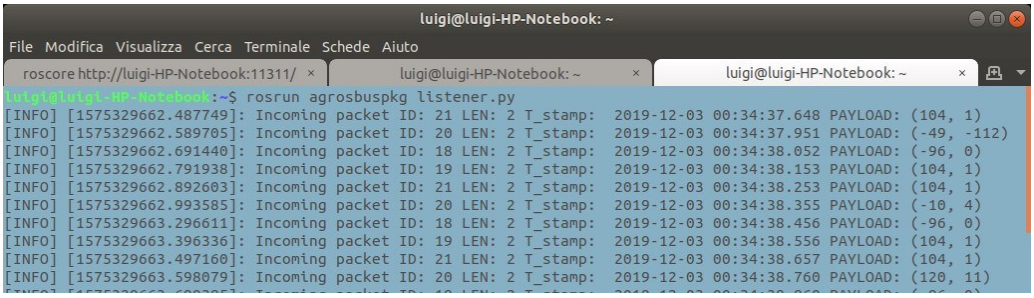


Figura 4.6

A questo punto ciascuna classe relativa ai sensori presente nello script *Listener.py* avrà creato un *topic* che servirà per inviare i *byte array* riconvertiti in valori reali verso i nodi che realizzino la logica di ciascun sensore. Per visualizzare i nuovi topic creati ricorriamo nuovamente al comando **`rostopic`**:

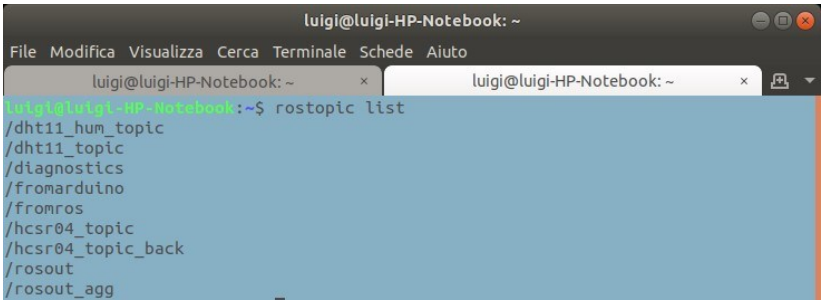


Figura 4.7

Per visualizzare i valori reali è necessario eseguire una echo per ciascun topic creato dalle classi dei dispositivi. Per brevità si riporta l'*echo* del solo *topic* riguardante il radar ad ultrasuoni HC-SR04:

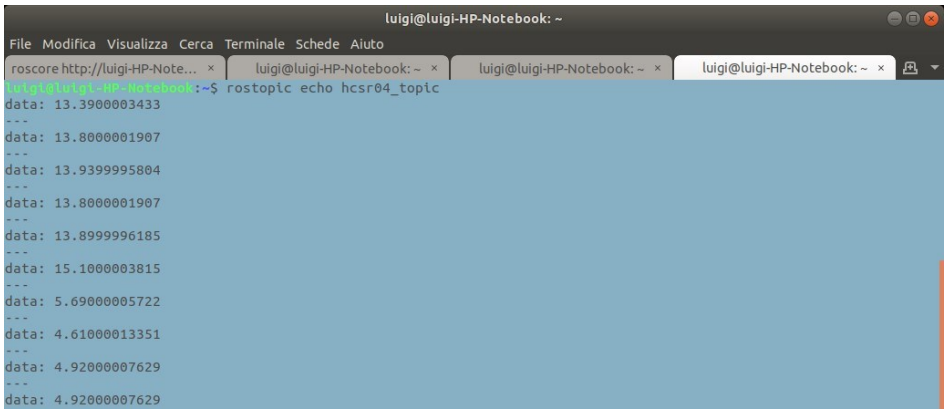


Figura 4.8

Quest'ultima fase conclude la comunicazione tra la rete su *CAN-bus* e ROS nella direzione verso ROS. Per testare anche il funzionamento nella direzione opposta aggiungiamo una funzione allo *script Listener.py* per accendere e spegnere il led integrato sulla scheda *MKR-Zero*. Di seguito il codice:

```
85  #where ros keep sensing packets in both directions
86  def main_loop():
87      l = 0
88      #start to listen for coming packets
89      while not rospy.is_shutdown():
90          if(l==0):
91              l=1
92          else:
93              l=0
94          sendBlinkInfo(l)
95          time.sleep(1)
```

Figura 4.9 modifica al main loop dello script Listener.py per inviare dati nella direzione verso la rete CAN. La variabile l il led dovrà essere HIGH o LOW (fromros)

```
57  def sendBlinkInfo(blinkYesNo):
58      ros_pkt.id = 1
59      ros_pkt.lenght = 1
60      ros_pkt.timestamp = time.time()
61      ros_pkt.value = [blinkYesNo]
62
63      pub.publish(ros_pkt)
```

Figura 4.10 funzione che costruisce il pacchetto ed effettua il publish sul topic di comunicazione verso la rete CAN stabilirà se

In questo modo l'informazione per accendere e spegnere giungerà dapprima al *Receiver* e poco dopo al *Sender* che richiamerà una *callback,* che avevamo introdotto precedentemente nel Sender (3.5.4), definita appositamente per questa tipologia di messaggio:

```
void SenderCallback(int packetSize){
  Serial.print("\n\n\n");
  Serial.print("ID: ");
  Serial.print(CAN.packetId());
  Serial.print(" LENGHT: ");
  Serial.print(packetSize);
  Serial.print(" VALUES: ");
  //get the raw bytes
  for(int i=0; i<packetSize; i++){
    int l = CAN.read();
    Serial.print(l,DEC);
    Serial.print(" | ");
    if(l==0) digitalWrite(LED_BUILTIN, LOW);
    if(l==1) digitalWrite(LED_BUILTIN, HIGH);
  }
}
```

Figura 4.11 SenderCallback

Di seguito un'immagine del monitor seriale del Sender durante la ricezione dei *Frame* Can per l'attivazione del led:

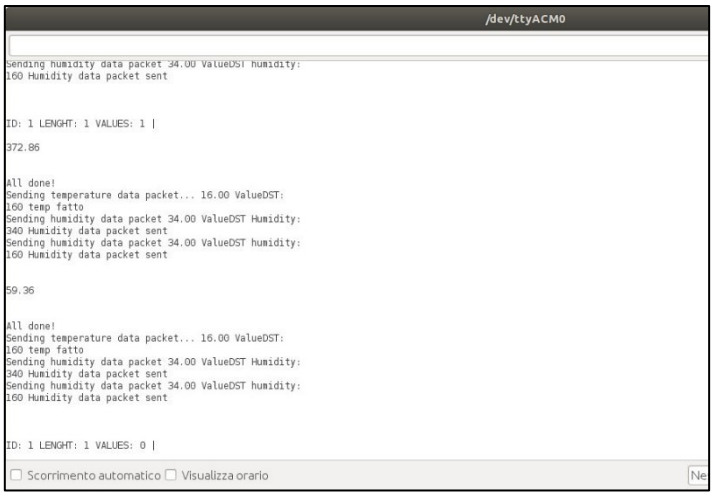


Figura 4.12 monitor seriale del CanSender durante la ricezione

Il *Sender* riceve il pacchetto contenente le informazioni sulle attivazioni del led e reagisce opportunamente richiamando la **`SenderCallback()`** che effettua una print sul monitor seriale. Successivamente il led viene impostato su **LOW (0)** o **HIGH (1)** dipendentemente dal valore del campo *values*. Questo ultimo step conclude lo sviluppo dell'interfaccia; si è ottenuta infatti la comunicazione bidirezionale tra la rete CAN e ROS, obiettivo dell'intera progettazione.

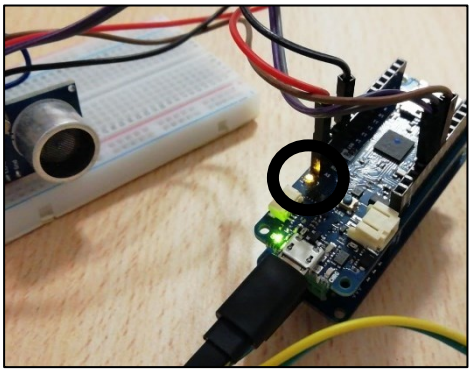


Figura 4.13 Led Built in HIGH

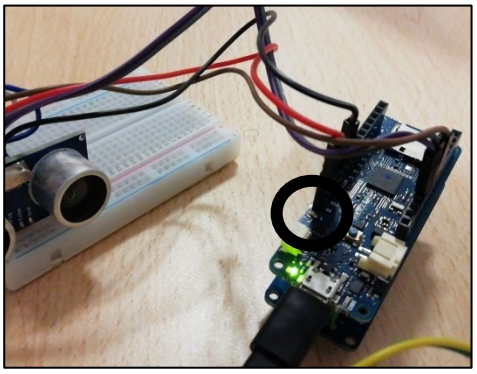


Figura 4.14 Led Built in LOW

5. Conclusioni

Come si è evidenziato nei capitoli precedenti, la progettazione dell'interfaccia ha prodotto i risultati sperati e ha evidenziato come l'utilizzo di ROS o Arduino siano fondamentali per chi necessita di rapidi e potenti strumenti per prototipazione e sviluppo. È stata ottenuta un'interfaccia stabile che non presenta criticità e che permette una comunicazione bidirezionale tra i dispositivi sulla rete CAN e ROS. In particolare, ROS si è dimostrato uno strumento dalle grandi potenzialità e l'interfaccia descritta in questo elaborato si aggiunge ai numerosi progetti che ne hanno fatto utilizzo con risultati positivi. Il progetto AGROSBUS, ora che è stato completato il primo punto, proseguirà con lo sviluppo degli ulteriori due descritti nella sezione 1.2 continuando a perseguire l'obiettivo primario di indirizzare lo sviluppo di applicazioni per l'AdP verso l'utilizzo di ROS. Nel paragrafo 2.1.4 abbiamo

evidenziato come siano previste ulteriori integrazioni per ROS, quindi ci si aspetta che la sua integrazione nell'ambito della robotica in generale ed in particolare dell'Adp continui a crescere.

Come sviluppi futuri un possibile obiettivo sarà di eliminare il *Sender* e di far comunicare il *Receiver* direttamente con un AdP vero e proprio, quindi renderlo in grado di comunicare con un CAN-bus reale, cosa che allo stato attuale, pur collegando direttamente i PIN CAN-H e CAN-L del CAN Shield ad una porta OBD2 o utilizzando un adattatore, non sarebbe in grado di fare.

Tutto il codice prodotto e descritto nell'elaborato può essere consultato accedendo al repository 'agrosbus' al seguente link: http://bitbucket.org/dimarcaluigi/agrosbus/

Bibliografia e Sitografia

Benzi, F. (s.d.). *Can Bus Controller Area Network.* Pavia: Dipartimento Ingegneria Elettrica Università di Pavia.

stvmac11. (s.d.). *Car to Arduino Communication: CAN Bus Sniffing and Brodcasting With Arduino.* Tratto da www.instructables.com: www.instructables.com/id/CAN-Bus-Sniffing-and-Broadcasting-withArduino/

Tellez, R. (2019, Luglio 22). *Top 10 ROS-based robotics companies in 2019.* Tratto da The RobotReport: www.therobotreport.com/top-10-ros-based-robotics-companies-2019/

Ackerman, E. (2013, marzo 26). *Clearpath Robotics Announces Grizzly Robotic Utility Vehicle*. Tratto da IEEE Spectrum: https://spectrum.ieee.org/automaton/robotics/industrial-robots/clearpath-roboticsannounces-grizzly-robotic-utility-vehicle

Bisaglia, C. (2018, Giugno). *Agricoltura di precisione in Italia: un'opportunità di aggiornamento delle agrotecniche, di sviluppo professionale e di efficienza del settore*. Tratto da Agrieuropa: https://agriregionieuropa.univpm.it/it/content/article/31/53/agricoltura-di-precisione-italiaunopportunita-di-aggiornamento-delle

Morgan Quigley, B. G. (s.d.). *ROS: an open-source Robot Operating System.* Stanford, CA: Computer Science Department, Stanford University.

Programmare un robot con ROS – Robot Operating System. (s.d.). Tratto da Robotiko: https://www.robotiko.it/programmare-un-robot-ros/

Susnea, I. &. (s.d.). *The bubble rebound obstacle avoidance algorithm for mobile robots*. 540 - 545. 10.1109/ICCA.2010.5524302.

Harman, Thomas & Fairchild, Carol. (2016). *ROS ROBOTICS BY EXAMPLE*.