Introduzione

Autori

- Valentino Di Giosaffatte
- Riccardo Armando Di Prinzio

Corso

Intelligent Systems And Robotics Laboratory
Master Degree in Computer Science
Curriculum: Network and Data Science

University of L'Aquila
Department of Information Engineering, Computer Science and Mathematics

Supervisore

- Prof. Giovanni De Gasperis

Virtual Robotics - Progetto 1

Introduzione

Descrizione

Il team ha realizzato, come progetto di robotica virtuale del corso Intelligent Systems And Robotics Laboratory, un'infrastruttura distribuita per la simulazione virtuale del robot Turtlebot3 in ambiente ROS versione 2.

Per quanto riguarda la tecnologia di virtualizzazione, il team ha deciso di utilizzare Docker.
Per la costruzione dell'ambiente distribuito, il team ha deciso di utilizzare Docker Compose.

Come ambiente server-client distribuito di simulazione virtuale 3d, il team ha deciso di utilizzare Gazebo Server (gzserver) e Gazebo Web (gzweb).

Link

- Repository: https://github.com/mdegree-nedas/intelligent_systems_and_robotics_lab-virtual-proj
- La documentazione aggiuntiva associata al progetto corrente si trova al seguente link: https://github.com/mdegree-nedas/intelligent_systems_and_robotics_lab-virtual-proj/tree/master/docs
- La documentazione relativa al meta sistema operativo ROS2 versione 'dashing' (utilizzata nel progetto) si trova al seguente link: https://docs.ros.org/en/dashing/index.html.
- La documentazione relativa al server di simulazione virtuale Gazebo Server versione '9' si trova al seguente link: https://osrf-distributions.s3.amazonaws.com/gazebo/api/9.0.0/index.html.
- La documentazione relativa al client web di simulazione virtuale Gazebo Web (gzweb) versione '1.4' si trova al seguente link: http://gazebosim.org/gzweb

- La documentazione relativa al robot Turtlebot3 si trova al seguente link: https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/.

Scopo del progetto

- Costruzione di un ambiente distribuito di simulazione virtuale e planning di navigazione per il robot Turtlebot3 all'interno di un contesto ROS2

- Costruzione di una soluzione versatile, eseguibile in modalità' distribuita su macchina locale o distribuita su una coreografia di macchine remote in rete locale

- Sperimentazione, analisi e programmazione in ambiente ROS2 (nuova versione sperimentale del meta sistema operativo robotico)
    - meccanismo di build e deploy di applicazioni pacchettizzate
    - interfacce di messaggistica e custom message broker
    - studio architetturale del nuovo contesto
    - studio della nuova versione dell'API per il linguaggio di programmazione Python versione 3.9
    - definizione dell'architettura ROS2 distribuita in rete locale

- Costruzione dell'ambiente distribuito client-server per la simulazione robotica virtuale
    - sperimentazione legata al nuovo client Gazebo Web (accelerazione 3d webgl in-browser)

- Sviluppo legato alla rappresentazione virtuale del robot Turtlebot3 nel contesto ROS2
    - creazione di una libreria wrapper di funzionalità (sensori e attuatori) per il suddetto robot, connessa al contesto ROS2 esposto dall'ambiente
    - creazione di un modulo di planning per la navigazione autonoma del robot all'interno dell'ambiente virtuale simulato
        - percezione geometrica dell'ambiente virtuale attraverso l'utilizzo di un sensore laser
        - reazione mediante attuazione alla presenza di ostacoli statici e dinamici

Architettura del software

- lo sviluppo ha avuto origine con una approfondita analisi architetturale del meta sistema operativo robotico ROS versione 2 (e doveroso confronto con ROS versione 1)
- all'analisi e' seguito uno studio della documentazione e delle interfacce di programmazione relative al linguaggio Python version 3.9

- utilizzando la tecnologia di virtualizzazione Docker, il team ha generato un'immagine completa dell'ambiente ROS versione 2 release Dashing
    - il deploy e' stato effettuato sulla piattaforma di cloud hosting Docker Hub, all'interno dell'organizzazione 'isrlab'
        - https://hub.docker.com/r/isrlab/ros2_dashing
    - il team ha generato le seguenti 3 versioni dell'immagine
        - tag: full
            - ambiente ROS 2 release Dashing completo
            - documentazione:
                - https://github.com/mdegree-nedas/intelligent_systems_and_robotics_lab-virtual-proj/blob/master/docs/ros2_dashing_full.md
        - tag: turtlebot3 (costruito sul tag full)
            - inserimento del contesto e del modello virtuale del robot Turtlebot3

                - documentazione:
                    - https://github.com/mdegree-nedas/intelligent_systems_and_robotics_lab-virtual-proj/blob/master/docs/ros2_dashing_turtlebot3.md
            - tag: gzweb_m (costruito sul tag turtlebot3)
                - costruzione dell'ambiente client web per la simulazione virtuale e relativi modelli 3D
                - documentazione:
                    - https://github.com/mdegree-nedas/intelligent_systems_and_robotics_lab-virtual-proj/blob/master/docs/ros2_dashing_gzweb_m.md
        - e' raccomandato l'utilizzo di una versione recente del kernel Linux (Docker utilizza features come cgroups, namespace, ecc)

- utilizzando la tecnologia di container management Docker Compose, il team ha generato il file di configurazione docker-compose.yml per l'inizializzazione e la gestione automatica della coreografia di containers
    - i Dockerfile sono disponibili al seguente link:
        - https://github.com/mdegree-nedas/intelligent_systems_and_robotics_lab-virtual-proj/tree/master/init

- il team ha costruito un meccanismo di build automatico e debugging (Makefile), auto-documentato e configurabile (config.mk) basato sul tool GNU Make

- architettura
    - macchina subscriber ROS 2 Dashing, package subscriber, gzserver
    - macchina simulator, client simulatore 3d in-browser, gzweb, modelli 3d
    - macchina publisher ROS 2 Dashing, package publisher, planner, libreria turtlebot3
    - link filesystem pacchetti ROS 2:
        - https://github.com/mdegree-nedas/intelligent_systems_and_robotics_lab-virtual-proj/blob/master/docs/packages.md

- subscriber:
    - macchina che contiene il simulatore virtuale Gazebo (parte server, gzserver) e il package subscriber, che ha il compito di effettuare lo startup della struttura del robot e del mondo simulato.

    - world: sezione contenente il file in sintassi SDF che descrive il mondo (l'ambiente) in cui il robot effettua i suoi movimenti e le sue percezioni

    - model: sezione contenente il modello "maze" da noi creato che descrive, in sintassi SDF, un labirinto costituito da mura con texture 'legno'

    - launch: lo startup dell'architettura viene effettuato mediante un launchfile ROS 2; il file ha lo scopo di eseguire un'istanza di gzserver con uno specifico file world ed effettuare lo startup della struttura del robot (creazione dei topics, impostazione delle configurazioni)

    - topics turtlebot3:
        - /cmd_vel: topic geometry_msgs/Twist che contiene i messaggi corrispondenti ai movimenti del robot (attuatore di movimento)

        - /turtlebot3_laserscan/out: topic sensor_msgs/LaserScan che contiene i messaggi corrispondenti alle percezioni del robot utilizzando il sensore laser

- publisher:

- macchina che contiene il package publisher. Esso contiene la libreria turtlebot3 e il planner. La libreria astrae la logica per controllare il robot e il planner utilizza la libreria ed effettua decisioni guidate dalla percezione dei sensori sui comandi di attuazione da impartire al robot

- libreria turtlebot3 (attuazione e sensoristica):
  - Libreria che racchiude la logica necessaria al corretto utilizzo del robot

  - attuazione: classe Python che racchiude la logica necessaria all'utilizzo degli attuatori per il movimento del robot. La classe Movement rende disponibile al programmatore i movimenti go_forward, go_back, turn_right, turn_left, stop.

  - sensoristica: classe Python che racchiude la logica necessaria all'utilizzo del sensore laser. La classe Sensor rende disponibile al programmatore una serie di metodi che restituiscono i dati percepiti dal robot. La percezione a 360° con LaserScan restituisce un array che contiene 360 elementi (ogni elemento corrisponde ad 1 grado della percezione). I metodi della classe ci permettono di effettuare le percezioni solo su aree a cui siamo interessati (back, left, right, front_left, front_right, back_left, back_right).

- libreria planner
  - libreria che racchiude la logica del planner che effettua decisioni basate sulle percezioni dei sensori sui comandi di attuazione da inviare al robot

  - utilizza la libreria turtlebot3 precedentemente costruita che racchiude la logica delle chiamate ai sensori e agli attuatori del robot Turtlebot3

  - esegue una serie di azioni in loop:
    - effettua la percezione dell'ambiente circostante (percezione visiva divisa in spicchi). La frequenza delle percezioni viene limitata da un clock per evitare il sovraccarico di richieste ai sensori del robot. Il clock delle percezioni mentre il robot si sta muovendo in avanti è più veloce del clock delle percezioni mentre il robot si sta girando poiché il rischio di collisione con un eventuale ostacolo è presente soltanto quando il robot si sta muovendo in avanti

    - analizza la percezione effettuata: trova per ogni spicchio la distanza minima dall'ostacolo rilevato

    - seleziona il movimento di attuazione da effettuare in base ai risultati dell'analisi della percezione (confrontandola con alcuni valori soglia che indicano la minima distanza di sicurezza dagli ostacoli consentita) e in base all'ultimo movimento effettuato dal robot

    - effettua il movimento di attuazione deciso nello step precedente: effettua una chiamata alla libreria turtlebot3 con il movimento specifico di attuazione da effettuare

- simulator:
  - macchina che contiene il simulatore virtuale per Gazebo gzweb (parte client).

  - il compito di questa macchina è di fornire all'utente un punto di accesso per assistere alla simulazione del robot nell'ambiente creato.

- l’utente può’ interagire in tempo reale con la simulazione 3d, ad esempio: fermando/avviando la simulazione real-time, eliminando/posizionando elementi nell’ambiente simulato, ecc. …)

- funzionamento:
  - la macchina imposta la variabile d’ambiente $GAZEBO_MASTER_URI in modo che punti alla macchina subscriber e successivamente esegue il comando “npm start” dalla cartella contenente la logica di gzweb.

  - dopo aver eseguito il comando “npm start” la macchina si connetterà al Gazebo server attivo sulla macchina subscriber e da quel momento l’utente sarà in grado di assistere alla simulazione visitando la pagina “http://localhost:8080” dal proprio browser web.

Validazione

- Abbiamo sviluppato il progetto su hardware con architettura linux/amd64 utilizzando sistemi operativi basati su Linux (void, arch e derivate), per Windows (wsl2)

- Abbiamo utilizzato Docker per virtualizzare l’applicazione distribuita

- lo startup e’ completamente automatizzato attraverso il tool Gnu Make (Makefile, config.mk)

- come risultati abbiamo ottenuto:
  - un’applicazione distribuita avviabile mediante l’utilizzo di make commands

  - un’ambiente distribuito costituito da containers Docker (avviati mediante Docker Compose) che dialogano tra loro (subscriber, publisher e simulator)

  - una libreria che astrae e rende disponibili al programmatore le funzionalità del robot Turtlebot3 (la percezione mediante i sensori e i comandi di attuazione)

  - un planner che utilizza la libreria turtlebot3 ed effettua decisioni sui comandi di attuazione in base alle percezioni sull’ambiente del robot. Lo scopo principale di esso è evitare che il robot urti gli ostacoli presenti sul suo cammino

  - un ambiente costruito da zero dove il robot può muoversi attraverso un labirinto costituito da mura di legno

- In conclusione:
  - Nella simulazione progettata, il robot target Turtlebot3 riesce a muoversi liberamente attraverso il labirinto creato senza urtare alcun ostacolo. Riesce ad evitare sia i muri del labirinto sia gli ostacoli posti dall’utente davanti al robot durante la simulazione in tempo reale

---

Physical Robotics - Progetto 1

Introduzione

Descrizione

Il team ha realizzato, come progetto di robotica fisica del corso Intelligent Systems And Robotics Laboratory, un'infrastruttura distribuita per la comunicazione basata su payload JSON tra un ambiente con contesto ROS2 e uno senza contesto ROS2.
L'infrastruttura è stata testata su una macchina Raspberry PI (architettura linux/arm/v7).

Per quanto riguarda la tecnologia di virtualizzazione, il team ha deciso di utilizzare Docker.
Per la costruzione dell'ambiente distribuito, il team ha deciso di utilizzare Docker Compose.

Come mezzo per lo scambio di messaggi JSON abbiamo deciso di utilizzare il datastore Redis.

Il progetto qui descritto è stato un lavoro preliminare per il Progetto 2 di Physical Robotics.

Link

- Repository: https://github.com/mdegree-nedas/intelligent_systems_and_robotics_lab-physical-proj
- La documentazione aggiuntiva associata al progetto corrente si trova al seguente link: https://github.com/mdegree-nedas/intelligent_systems_and_robotics_lab-physical-proj/tree/master/docs
- La documentazione relativa al datastore Redis si trova a questo link: https://redis.io/documentation
- La documentazione relativa al robot Turtlebot3 si trova a questo link: https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/
- La documentazione relativa alla libreria Python "redis" si trova a questo link: https://github.com/andymccurdy/redis-py
- La documentazione relativa alla libreria di conversione ROS2 rosidl_runtime_py utilizzata per convertire i messaggi ROS2 a JSON si trova a questo link: https://github.com/ros2/rosidl_runtime_py
- Le specifiche ufficiali ed altre informazioni relative al Raspberry utilizzato si trovano a questo link: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/
- Il sistema operativo Raspberry PI OS utilizzato sul Raspberry a nostra disposizione si trova a questo link: https://downloads.raspberrypi.org/raspios_full_armhf/images/raspios_full_armhf-2021-05-28/2021-05-07-raspios-buster-armhf-full.zip
- L'immagine Docker di Redis utilizzata si trova a questo link: https://hub.docker.com/_/redis

Scopo del progetto

- Realizzazione di un'infrastruttura distribuita che permette di far comunicare un ambiente con contesto ROS2 con un ambiente senza contesto ROS2 attraverso payloads JSON inviati sfruttando il datastore Redis.

- Programmazione in ambiente con contesto ROS2 di un package che contiene un planner che invia comandi semplici ad un ricevitore senza contesto ROS2.
    - Per la generazione dei comandi abbiamo deciso di appoggiarci alla libreria Turtlebot3 precedentemente costruita per il progetto di Virtual Robotics.

- Programmazione in ambiente con contesto ROS2 di un package che intercetta i comandi inviati dal planner su un topic ROS2, traduce i comandi intercettati in messaggi JSON ed effettua il forwarding dei messaggi su un topic Redis.

- Programmazione in ambiente con contesto ROS2 di un package che intercetta i messaggi ricevuti da un topic Redis, traduce i messaggi intercettati in messaggi ROS2 String ed effettua il forwarding dei messaggi su un topic ROS2.

- Programmazione Python in ambiente senza contesto ROS2 di un ricevitore per comandi inviati dal planner ROS2 che invia in risposta una stringa di saluto.

- Programmazione Python di un publisher ed un subscriber Redis sfruttando la libreria Python redis (versione 3.5.3).

- Utilizzo della libreria di conversione ROS2 rosidl_runtime_py (versione 0.7.11) per la conversione dei messaggi dei topic ROS2 in JSON.

- Preparazione di una macchina Raspberry PI (version 3 model B rev 1.2):
    - Flashing del sistema operativo Raspbian gnu/linux 10 Buster su Raspberry PI
    - Settaggio utente pi con password "admin"
    - Configurazione della macchina Raspberry
    - Installazione e configurazione di Docker (versione 20.10.6)
    - Installazione di Docker Compose (versione 1.29.1)

- Utilizzo di un container Docker di appoggio contenente Redis (immagine Docker redis:6.2.2-buster) in modo da permettere il passaggio dei messaggi JSON dagli ambienti descritti senza l'obbligo di installazione dell'ambiente Redis nella macchina.

Architettura del software

- utilizzando la tecnologia di virtualizzazione Docker, il team ha generato un'immagine completa per Raspberry PI dell'ambiente ROS versione 2 release Dashing:
    - il deploy e' stato effettuato sulla piattaforma di cloud hosting Docker Hub, all'interno dell'organizzazione 'isrlab'
        - https://hub.docker.com/r/isrlab/ros2_dashing_rpi
    - il team ha generato le seguenti 2 versioni dell'immagine
        - tag: full
            - ambiente ROS 2 release Dashing completo
            - documentazione https://github.com/mdegree-nedas/intelligent_systems_and_robotics_lab-virtual-proj/blob/master/docs/ros2_dashing_full.md

        - tag: redis (costruito sul tag full)
            - inserimento della libreria Python3 redis versione 3.5.3
            - L'immagine aggiunge all'immagine con il tag full il layer creato dall'azione del comando:
                - pip3 install redis==3.5.3

- utilizzando la tecnologia di container management Docker Compose, il team ha generato il file di configurazione docker-compose.yml per l'inizializzazione e gestione automatica della coreografia di containers
    - il Dockerfile del servizio "publisher" è disponibile al seguente link: https://github.com/mdegree-nedas/intelligent_systems_and_robotics_lab-physical-proj/tree/master/init/publisher

- utilizzando il Raspberry PI 3 a disposizione del team, i membri hanno approfondito la loro conoscenza riguardo il montaggio, l'installazione, la configurazione, e l'utilizzo di una macchina Raspberry:
    - Flash del sistema operativo sulla SDcard
    - Primo avvio e configurazione del sistema operativo Raspberry OS
    - Utilizzo in remoto del Raspberry
        - ssh
        - vnc

- il team ha costruito un meccanismo di build automatico e debugging (Makefile), auto-documentato e configurabile (config.mk) basato sul tool GNU Make

- architettura
    - servizio publisher: derivato dall'immagine Docker ros2_dashing_rpi:redis che contiene:
        - package publisher: costituito da un planner e dalla libreria turtlebot3
        - package virt_redis_mirror: costituito dal programma translater
    - servizio redis: derivato dall'immagine redis:6.2.2-buster
    - host_redis_mirror: Python virtual environment senza contesto ROS 2

- package publisher:
    - pacchetto ROS2 che contiene un planner che invia comandi utilizzando la libreria turtlebot3.

    - libplanner:
        - un semplice planner che ad ogni secondo sceglie randomicamente un comando di movimento dalla libreria turtlebot3 ignorando la percezione dei sensori

    - libturtlebot3:
        - Libreria che racchiude la logica necessaria al corretto utilizzo del robot

        - movimenti: classe Python che racchiude la logica necessaria all'utilizzo degli attuatori per il movimento del robot. La classe Movement rende disponibile al programmatore i movimenti go_forward, go_back, turn_right, turn_left, stop.

        - sensori: classe Python che racchiude la logica necessaria all'utilizzo del sensore laser. La classe Sensor rende disponibile al programmatore una serie di metodi che restituiscono i dati percepiti dal robot. La percezione a 360° con LaserScan restituisce un'array che contiene 360 elementi (ogni elemento corrisponde ad 1° della percezione). I metodi della classe ci permettono di effettuare le percezioni solo su aree a cui siamo interessati (back, left, right, front_left, front_right, back_left, back_right).

- package virt_redis_mirror:
    - pacchetto ROS2 che contiene un programma che effettua la traduzione di messaggi pubblicati su topic ROS2 ed il forwarding dei suddetti messaggi su topic Redis e viceversa.

    - classe Translater: utilizza la classe RedisWrapper. Ha il compito di monitorare 4 topics: cmd_vel, to_ros2 (ROS2 topics) e to_redis, from_redis (Redis topics). I messaggi ROS2 geometry_msgs/Twist intercettati nel topic cmd_vel vengono convertiti in JSON ed inviati sul topic Redis to_redis.
      I messaggi ricevuti dal topic from_redis vengono wrappati in un oggetto std_msgs/String e viene inviato sul topic ROS2 to_ros2

    - classe RedisWrapper: classe wrapper per la libreria Python3 redis, contiene metodi che astraggono le chiamate alla libreria redis

- servizio redis:
    - Container derivato dall'immagine Docker redis:6.2.2-buster che ha lo scopo di fare da tramite per la comunicazione tra il container con contesto ROS2 e l'host senza contesto ROS2

- host_redis_mirror:
    - environment Python che contiene un programma che ha lo scopo di ricevere i messaggi JSON provenienti dal contesto ROS2 e restituire una stringa di saluto al mittente.

    - script main: script Python3: utilizza la classe RedisWrapper. Ha il compito di mettersi in ascolto sul topic Redis to_redis e, una volta ricevuto un messaggio JSON, lo converte in dizionario, lo stampa a video ed invia una stringa in risposta sul topic Redis from_redis

    - classe RedisWrapper: classe wrapper per la libreria Python3 redis, contiene metodi che astraggono le chiamate alla libreria redis

Validazione

- Abbiamo sviluppato il progetto su hardware con architettura linux/arm/v7 (Raspberry PI) utilizzando il sistema operativo Raspberry OS

- Abbiamo sfruttato la virtualizzazione offerta da Docker per containerizzare l'applicazione distribuita

- come risultati abbiamo ottenuto:
    - un'applicazione distribuita avviabile mediante l'utilizzo di make commands

    - un'ambiente distribuito costituito da containers Docker (avviati mediante Docker Compose) che dialogano tra loro (redis, publisher)

    - un planner che invia comandi di attuazione sfruttando la libreria turtlebot3

    - una classe wrapper Python che astrae le funzionalità utilizzate per inviare e ricevere messaggi attraverso Redis

    - un traduttore che ha il compito di convertire i messaggi passanti su un topic ROS2 in JSON e successivamente inviare le stringhe JSON su un topic Redis e viceversa

    - un ricevitore che ascolta su un topic Redis i messaggi JSON inviati dal traduttore e, alla ricezioni di essi, li converte in dizionario Python e li stampa a video. Successivamente esso pubblica una stringa di saluto su un topic Redis (su cui il traduttore ha fatto subscribe in precedenza)

- In conclusione:
    - Siamo riusciti ad implementare una comunicazione bidirezionale tra un publisher in un container con contesto ROS2 ed un ricevitore situato nell'host senza contesto ROS2 attraverso l'utilizzo di Redis:
        - il planner (situato nel container con contesto ROS2) pubblica messaggi di attuazione sul topic ROS2 /cmd_vel (sfruttando la libreria turtlebot3)
        - i messaggi di attuazione sul topic ROS2 vengono intercettati dal traduttore
        - il traduttore converte i messaggi di attuazione (di tipo Twist) in messaggi JSON
        - il traduttore invia i messaggi JSON su Redis
        - grazie al container Redis, il messaggio viene ricevuto dal ricevitore (situato nella macchina host senza contesto ROS2)
        - il ricevitore riceve il messaggio JSON dal topic Redis
        - il messaggio JSON viene convertito in dizionario Python e viene stampato a video
        - il ricevitore crea una stringa di saluto e la pubblica su uno specifico topic Redis
        - il traduttore intercetta il messaggio di saluto inviato dal ricevitore sul topic Redis
        - il messaggio viene convertito nel tipo String ROS2
        - il traduttore pubblica il messaggio String su uno specifico topic ROS2

Physical Robotics - Progetto 2

Introduzione

Descrizione

Il team ha realizzato, come secondo progetto di robotica fisica del corso Intelligent Systems And Robotics Laboratory, un'infrastruttura distribuita che ha lo scopo di semplificare il lavoro del programmatore generando automaticamente codice Python a partire da un file di configurazione YAML che descrive il robot target.
L'infrastruttura è stata testata su una macchina Raspberry PI (con architettura linux/arm/v7) e su macchine basate su sistema operativo Linux (con architettura linux/amd64).

Per quanto riguarda la tecnologia di virtualizzazione, il team ha deciso di utilizzare Docker.
Per la costruzione dell'ambiente distribuito, il team ha deciso di utilizzare Docker Compose.

Come mezzo per lo scambio di messaggi JSON abbiamo deciso di utilizzare il datastore Redis.

Questo progetto riprende il lavoro preliminare sviluppato nel Progetto 1 di Physical Robotics e lo amplia secondo gli scopi previsti del progetto.

Link

- Repository: https://github.com/mdegree-nedas/rcp
- La documentazione aggiuntiva associata al progetto corrente si trova al seguente link: https://github.com/mdegree-nedas/rcp/tree/master/docs
- La documentazione relativa al robot Freenove 4WD Smart Car Kit for Raspberry PI si trova al seguente link: https://github.com/Freenove/Freenove_4WD_Smart_Car_Kit_for_Raspberry_Pi
- La documentazione relativa al datastore Redis si trova a questo link: https://redis.io/documentation
- La documentazione relativa alla libreria Python “redis”: https://github.com/andymccurdy/redis-py
- La documentazione relativa alla libreria Python “pyYAML”: https://pyyaml.org/wiki/PyYAMLDocumentation
- L’immagine Docker di Redis utilizzata si trova a questo link: https://hub.docker.com/_/redis
- Per la creazione di immagini Docker multi-architettura abbiamo preso spunto da una guida utile che spiega in maniera dettagliata il processo di creazione dell’immagine, la guida menzionata si trova a questo link: https://medium.com/@artur.klauser/building-multi-architecture-docker-images-with-buildx-27d80f7e2408
- Le specifiche ufficiali ed altre informazioni relative al Raspberry utilizzato si trovano a questo link: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/
- Il sistema operativo Raspberry PI OS utilizzato sul Raspberry a nostra disposizione si trova a questo link: https://downloads.raspberrypi.org/raspios_full_armhf/images/raspios_full_armhf-2021-05-28/2021-05-07-raspios-buster-armhf-full.zip

Scopo del progetto

- Realizzazione di un’infrastruttura distribuita che permette di far comunicare un ambiente con contesto ROS2 con un ambiente senza contesto ROS2 attraverso payloads JSON inviati sfruttando il datastore Redis.
- Realizzazione di un generatore di codice Python che genera la libreria Python specifica del robot scelto partendo da un file di configurazione scritto in sintassi YAML
- Utilizzo della libreria Python pyYAML (versione 5.4.1) per il parsing del file di configurazione YAML
- Utilizzo della libreria Python redis (versione 3.5.3) per la comunicazione tra gli ambienti attraverso messaggi JSON

- Generazione automatica di un publisher-subscriber che permetta la comunicazione attraverso Redis

- Utilizzo di un container Docker di appoggio contenente Redis (immagine Docker redis:6.2.2-buster) in modo da permettere il passaggio dei messaggi JSON dagli ambienti descritti senza l'obbligo di installazione dell'ambiente Redis nella macchina.

- Programmazione di esempi automatizzati che mostrano l'utilizzo della libreria autogenerata.

- L'infrastruttura distribuita deve essere funzionante sia in locale su un sistema basato su Linux con architettura linux/amd64, sia in locale su una macchina Raspberry PI con architettura linux/arm/v7 sia in remoto. In quest'ultimo caso l'ambiente ROS2 viene eseguito su sistema basato su Linux e l'ambiente senza ROS2 viene eseguito su macchina Raspberry.

- Creazione di immagini Docker multi-architettura allo scopo di poter essere utilizzate da tutte le macchine specificate senza bisogno di creare immagini diverse

- Preparazione di una macchina Raspberry PI (version 3 model B rev 1.2):
    - Flashing del sistema operativo Raspbian gnu/linux 10 Buster su Raspberry PI
    - Settaggio utente pi con password "admin"
    - Configurazione della macchina Raspberry
    - Installazione e configurazione di Docker (versione 20.10.6)
    - Installazione di Docker Compose (versione 1.29.1)

Architettura del software

- utilizzando la tecnologia di virtualizzazione Docker, il team ha generato un'immagine completa multi-architettura (linux/amd64, linux/arm/v7) dell'ambiente base ROS versione 2 release Dashing contenente la libreria Python3 redis:
    - il deploy e' stato effettuato sulla piattaforma di cloud hosting Docker Hub, all'interno dell'organizzazione 'isrlab'
        - https://hub.docker.com/r/isrlab/rcp
    - il team ha generato la seguente versione dell'immagine
        - tag: stable
            - ambiente ROS 2 release Dashing base + libreria Python3 redis versione 3.5.3
            - documentazione https://github.com/mdegree-nedas/rcp/blob/master/docs/rcp_docker_image/README.md
            - Dockerfile https://github.com/mdegree-nedas/rcp/blob/master/docs/rcp_docker_image/Dockerfile

- utilizzando la tecnologia di container management Docker Compose, il team ha generato il file di configurazione docker-compose.yml per l'inizializzazione e gestione automatica della coreografia di containers
    - il Dockerfile del servizio "ros" è disponibile al seguente link: https://github.com/mdegree-nedas/rcp/blob/master/rcp/src/test/init/Dockerfile

- utilizzando il Raspberry PI 3 a disposizione del team, i membri hanno approfondito la loro conoscenza riguardo il montaggio, l'installazione, la configurazione, e l'utilizzo di una macchina Raspberry:

    - Flash del sistema operativo sulla SDcard
    - Primo avvio e configurazione del sistema operativo Raspberry OS
    - Utilizzo in remoto del Raspberry
        - ssh
        - vnc

- il team ha costruito un meccanismo di build automatico e debugging (Makefile), auto-documentato e configurabile (config.mk) basato sul tool GNU Make

- architettura
    - directory test: servizio ros derivato dall'immagine Docker rcp:stable, workspace ROS2 contenente il package ros_usage_example, directory host, directory examples, directory init
    - servizio redis: derivato dall'immagine Docker redis:6.2.2-buster
    - directory out
    - directory config
    - libreria Python librcp
    - generatore di codice rcp

- test:
    - directory che esegue gli esempi che utilizzano la libreria autogenerata. Utilizza un file docker-compose.yml che avvia i servizi utilizzati (ros e redis)

    - examples: directory che contiene i files di esempio di utilizzo della libreria autogenerata
        - actuators:
            - esempio che mostra la parte con contesto ROS che invia comandi di attuazione alla parte con contesto non ROS

            - nonros_usage_example: esempio della parte non ROS che mostra la ricezione di comandi di attuazione

            - ros_usage_example: esempio della parte ROS che mostra l'invio di un comando di attuazione alla parte non ROS

        - sensors:
            - esempio che mostra la parte con contesto non ROS che invia dati dei sensori alla parte con contesto ROS

            - nonros_usage_example: esempio della parte non ROS che mostra l'invio di dati di sensori (generati in una callback registrata) alla parte ROS

            - ros_usage_example: esempio della parte ROS che mostra la ricezione di dati di sensori ricevuti dalla parte non ROS in un'apposita callback registrata

        - loop:
            - esempio che mostra la parte con contesto non ROS che invia dati dei sensori alla parte con contesto ROS, quest'ultima parte utilizza un controller che interpreta i dati dei sensori ricevuti e successivamente invia comandi di attuazioni alla parte con contesto non ROS.

            - nonros_usage_example: esempio della parte non ROS che mostra l'invio di dati di sensori (generati in una callback registrata) alla parte ROS e la

conseguente ricezione di comandi di attuazione provenienti dalla parte ROS.

- ros_usage_example: esempio della parte ROS che mostra inizialmente la ricezione di dati di sensori ricevuti dalla parte non ROS in un'apposita callback registrata. I dati ricevuti vengono inviati ad un controller che effettua una decisione sul comando di attuazione da eseguire. Tale comando viene inviato alla parte non ROS.

- controller_example: classe Controller che in base ai dati del sensore, effettua una decisione sul comando di attuazione da eseguire (left, forward, right, idle).

  - workspace: directory che corrisponde ad un workspace ROS2 che ha al suo interno il pacchetto ROS2 ros_usage_example. Tale pacchetto contiene la parte ros della libreria autogenerata e il main file della parte ros dello specifico esempio che si è deciso di provare

  - host: directory che contiene la parte noros della libreria autogenerata e il main file della parte nonros dello specifico esempio che si è deciso di provare

  - init: contiene il Dockerfile per la generazione dell'immagine docker del servizio ros. Il servizio monterà a runtime il workspace ROS2 ed eseguirà la parte ros

- redis:
  - Container derivato dall'immagine Docker redis:6.2.2-buster che ha lo scopo di fare da tramite per la comunicazione tra il container con contesto ROS2 e l'environment Python senza contesto ROS2

- out:
  - directory che contiene l'output della generazione della libreria specifica del robot target

  - noros:
    - broker.py:
      - classe RedisWrapper: classe che astrae le funzioni di subscribe e publish della libreria Python3 redis

      - classe RedisMiddleware: classe che offre all'esterno la funzione send (che si occupa di effettuare una publish di un messaggio convertito in JSON su uno specifico topic) e la funzione receive (che si occupa di effettuare le subscribe sui topic specifici per ogni attuatore)

    - core.py: classe che corrisponde al robot target. Contiene al suo interno l'astrazione di tutte le componenti del robot specificate nel file di configurazione YAML. Questa classe viene utilizzata dal programmatore nel contesto non ROS per registrare metodi per la ricezione dei dati dei comandi per gli attuatori, per registrare metodi per l'invio dei dati dei sensori e per prepararsi alla ricezione dei dati dei comandi per gli attuatori. Inoltre contiene informazioni su tutti i topic relativi agli attuatori e ai sensori, ed informazioni sui comandi.

    - template.py: file che contiene la classe Template. Essa offre al programmatore della callbacks implementabili in cui è possibile scrivere la logica che date le

informazioni ricevute tramite argomento della callback, effettua il movimento del robot sfruttando le API dello specifico robot utilizzato.
Inoltre offre al programmatore della callbacks implementabili in cui è possibile scrivere la logica per poter leggere uno specifico sensore del robot fisico ed inviare il risultato al contesto ROS.

- ros:
  - broker.py: file che contiene classi che astraggono la comunicazione attraverso la libreria Python3 redis.
    - classe RedisWrapper: classe che astrae le funzioni di subscribe e publish della libreria Python3 redis
    - classe RedisMiddleware: classe che offre all'esterno la funzione send (che si occupa di effettuare una publish di un messaggio che ha uno specifico "msg_type" per un comando "command" su uno specifico topic) e la funzione receive (che si occupa di effettuare le subscribe sui topic specifici per ogni sensore)
    - classe Converter: classe contiene le funzioni che effettuano la conversione in JSON di uno specifico msg_type (per esempio: GeometryMsgsTwist)
  - core.py: classe che corrisponde al robot target. Contiene al suo interno l'astrazione di tutte le componenti del robot specificate nel file di configurazione YAML. Questa classe viene utilizzata dal programmatore nel contesto ROS per registrare metodi per la ricezione dei dati dai sensori, per inviare comandi agli attuatori e per prepararsi alla ricezione di dati dai sensori. Inoltre contiene informazioni su tutti i topic relativi agli attuatori e ai sensori, sui comandi e sui tipi di messaggi usati (per esempio: twist)
  - interface.py: nel file sono definite le classi che corrispondono ai "msg_type", cioè ai tipi di messaggi su cui è possibile effettuare le send (per esempio: GeometryMsgsTwist)

- config:
  - directory che contiene i file YAML che descrivono la struttura del robot target

- librcp:
  - libreria Python3 che permette di effettuare il parsing del file YAML di configurazione di un robot e permette di generare da esso una libreria che ha il compito di astrarre le operazioni di comunicazione tra il contesto ROS e il contesto non ROS e quindi fornire all'utente una semplificazione per poter utilizzare un robot attraverso ROS.
  - core:
    - classe Rcp: classe che espone gli oggetti Parser e Generator all'esterno
  - generator:
    - classe Generator: classe che espone gli oggetti noros_core, noros_template, noros_broker, ros_core, ros_interface, ros_broker, ovvero tutti gli oggetti che saranno utilizzati per generare le rispettive parti non ros e ros
    - nonros:
      - generator_broker: classe che si occupa della generazione del file Python broker.py della parte nonros

                - generator_core: classe che si occupa della generazione del file Python core.py della parte nonros

                - generator_template: classe che si occupa della generazione del file Python template.py della parte nonros

            - ros:
                - generator_broker: classe che si occupa della generazione del file Python broker.py della parte ros

                - generator_core: classe che si occupa della generazione del file Python core.py della parte ros

                - generator_interface: classe che si occupa della generazione del file Python interface.py della parte ros

        - reader:
            - classe Parser: classe che espone gli oggetti Syntax e Structure

            - classe Syntax: classe che si occupa della lettura e del parsing del file YAML. Le informazioni contenute nel file YAML vengono convertite in un dizionario Python dict

            - classe Structure: classe che si occupa di validare la struttura importata dal file YAML. La struttura importata viene controllata tramite asserzioni e, se essa è una configurazione “ben formata” di un robot, viene ritornata, altrimenti viene sollevata un’eccezione

- rcp.py:
    - main script di rcp. Istanzia la classe Rcp e, per ogni file YAML passato come argomento nella chiamata da console, utilizza la libreria librcp per parsare, validare e successivamente generare il codice di libreria specifica per il robot target definito nello specifico file YAML

Validazione

- Abbiamo sviluppato il progetto su hardware con architettura linux/amd64 utilizzando sistemi operativi basati su Linux (void, artix, arch, wsl2) e su hardware Raspberry (architettura linux/arm/v7) utilizzando il sistema operativo Raspberry OS

- Abbiamo sfruttato la virtualizzazione offerta da Docker per containerizzare l’applicazione distribuita

- come risultati abbiamo ottenuto:
    - un’applicazione distribuita avviabile mediante l’utilizzo di make commands

    - un’ambiente distribuito costituito da containers Docker generati da immagini multi-architettura (e avviati mediante Docker Compose) che dialogano tra loro (redis, ros)

    - un’ambiente distribuito che riesce a funzionare sia in locale (sulla stessa macchina) sia in remoto (utilizzando più macchine)

    - un’ambiente che permette la comunicazione bidirezionale attraverso Redis tra un container con contesto ROS e dei ricevitori situati sull’environment Python senza contesto ROS

    - un esempio di file di configurazione YAML che descrive il robot Freenove 4WD Smart Car

    - un generatore di codice Python che genera una libreria specifica del robot target contenente le parti ros e non ros a partire dal file di configurazione del robot scritto in sintassi YAML

    - una libreria autogenerata riutilizzabile che astrae e descrive le funzionalità del robot

    - la sottolibreria ros che permette di astrarre funzionalità e logica per inviare comandi di attuazione alla parte non ros e per ricevere percezioni dalla parte non ros

    - la sottolibreria non ros che permette di astrarre funzionalità e logica per inviare percezioni alla parte ros e ricevere comandi di attuazione dalla parte ros

    - degli esempi riutilizzabili che mostrano l’utilizzo della libreria del robot. Gli esempi presentati forniscono delle linee guida per la scrittura di programmi che sfruttano la libreria autogenerata

- In conclusione:
    - è stato progettato con successo un programma che, da un file di descrizione del robot in sintassi YAML, genera con successo una libreria specifica del robot che ha il compito di facilitare la programmazione e l’interfacciamento del robot fisico con la logica situata in un container ROS2

    - sono stati progettati degli esempi che mostrano l’utilizzo della libreria autogenerata e che forniscono delle utili linee guida per il programmatore che utilizzerà la libreria con il proprio robot