Robotica Virtuale

Progetto 1

Introduzione

- costruzione di un’infrastruttura distribuita per la simulazione virtuale del robot Turtlebot 3 in ambiente ROS 2
- utilizzo delle tecnologie di virtualizzazione e composizione Docker e Docker Compose
- utilizzo di Gazebo Server (gzserver, server) e Gazebo Web (gzweb, client) per la costruzione dell’ambiente distribuito di simulazione virtuale 3D
- sviluppo vincolato al kernel Linux per requisiti di virtualizzazione (cgroups, namespaces, …) (vale per tutti i Progetti che vedremo successivamente)

Finalità 1/2

- costruzione di un ambiente distribuito di simulazione virtuale 3D eseguibile su coreografie fisiche di macchine remote in rete locale o virtuali in macchina locale
- analisi e sperimentazione in ambiente ROS 2
  - studio architetturale del nuovo contesto
  - analisi interfacce di messaggistica relative al nuovo modulo di message brokering
  - nuovo meccanismo di build e deploy di applicazioni pacchettizzate (Python)
  - analisi strutturale della nuova versione dell'API nativa (Python 3.9, rclpy)
  - definizione dell'architettura del progetto
- sperimentazione legata al nuovo meccanismo di simulazione 3D distruita in rete locale: utilizzo del nuovo client webgl-based in-browser Gazebo Web

Finalità 2/2

- porting della rappresentazione virtuale del robot Turtlebot 3 all’interno del contesto ROS 2 (modello tridimensionale, canali attuatori e sensori, ecc...)
- creazione di una libreria wrapper di funzionalità per il Turtlebot 3 (sensoristica e attuazione) e connessione al contesto esposto dall’ambiente ROS 2
- sviluppo di un modulo di planning per la navigazione autonoma del Turtlebot 3 in ambiente virtuale 3D simulato
    - percezione geometrica dell’ambiente virtuale mediante l’utilizzo di un sensore laser circolare
    - reazione mediante attuazione alla presenza di ostacoli statici e dinamici (anche inseriti a runtime)

Architettura 1/4

- approfondita analisi architetturale del meta sistema operativo ROS 2 (confronto con la precedente versione ROS 1) e studio delle interfacce di programmazione per il linguaggio Python 3.9
- generazione di un'immagine completa dell'ambiente ROS versione 2 release 'Dashing' su layer Gnu/Linux Ubuntu LTS 18.04 (architettura linux/amd64) e deployment su piattaforma di cloud hosting Docker Hub
  - strutturazione gerarchica del deployment in 3 tag (full, turtlebot3, gzweb_m)
- scrittura di un file di configurazione (docker-compose.yml) per l'inizializzazione e la gestione automatica della coreografia di containers mediante l'utilizzo di Docker Compose
- scrittura di un meccanismo di building automatico, debugging, auto-documentato e configurabile (Makefile gerarchici, config.mk) mediante l'utilizzo del tool Gnu Make

Architettura 2/4 (macchina subscriber)

- si compone di un ambiente ROS 2 completo, del server di simulazione 3D distribuita ‘gzserver’
- sviluppo e costruzione del package ‘subscriber’, al cui interno vi è la logica per la generazione del modello strutturale del Turtlebot 3 e per l’inizializzazione del mondo virtuale 3D (world, model, launch)
- inizializzazione canale ROS 2 nativo ‘/cmd_vel’ con annesso topic di tipo ‘geometry_msgs/Twist’ (messaggistica per l’attuazione del robot)
- inizializzazione canale ROS 2 nativo ‘/turtlebot3_laserscan/out’ con annesso topic di tipo ‘sensor_msgs/LaserScan’ (messaggistica per la sensoristica laser del robot)

Architettura 3/4 (macchina publisher)

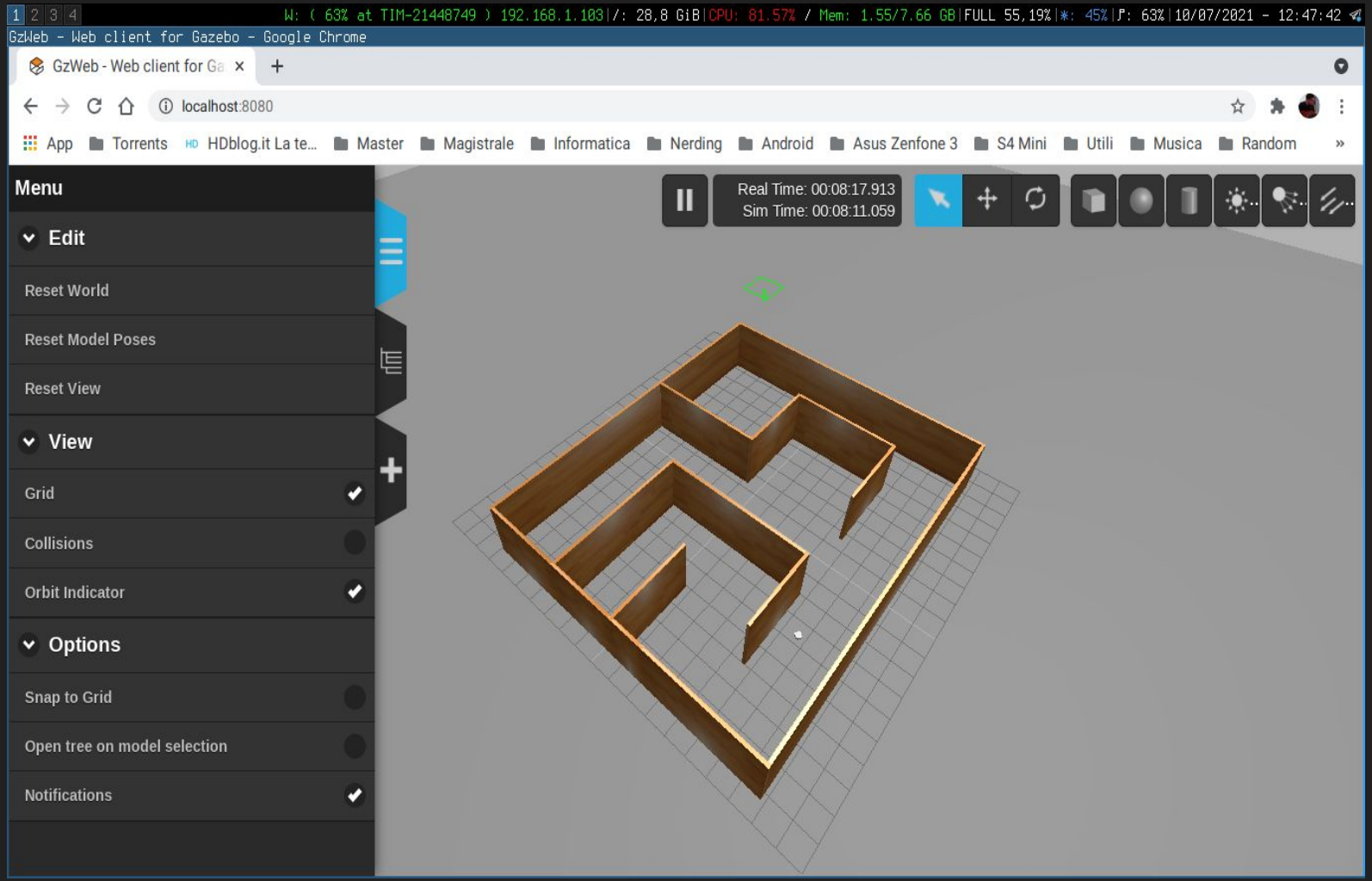
- si compone di un ambiente ROS 2 completo
- sviluppo e costruzione del package ‘publisher’ che contiene la libreria wrapper per il Turtlebot 3 e il planner di navigazione autonoma
- libreria turtlebot3
  - racchiude la logica di controllo relativa all’attuazione e alla sensoristica laser del robot
  - si collega alle primitive dell’ambiente ROS 2 mediante l’utilizzo della libreria nativa ‘rclpy’
  - espone delle interfacce per l’interazione con il Turtlebot 3 relative ad attuazione e alla sensoristica laser
- libreria planner
  - racchiude la logica di planning navigazionale che effettua decisioni sui comandi di attuazione, basate sulle percezioni del sensore laser
  - utilizza la libreria turlebot3 per la comunicazione bidirezionale con il robot
  - permette la navigazione autonoma del robot in ambiente virtuale 3D simulato e la reazione dinamica alla presenza di ostacoli (anche inseriti a runtime)
  - esecuzione basata su una serie di callback asincrone in loop a differenti frequenze di funzionamento (clock multipli), sinergia sincrona tra percezione e attuazione

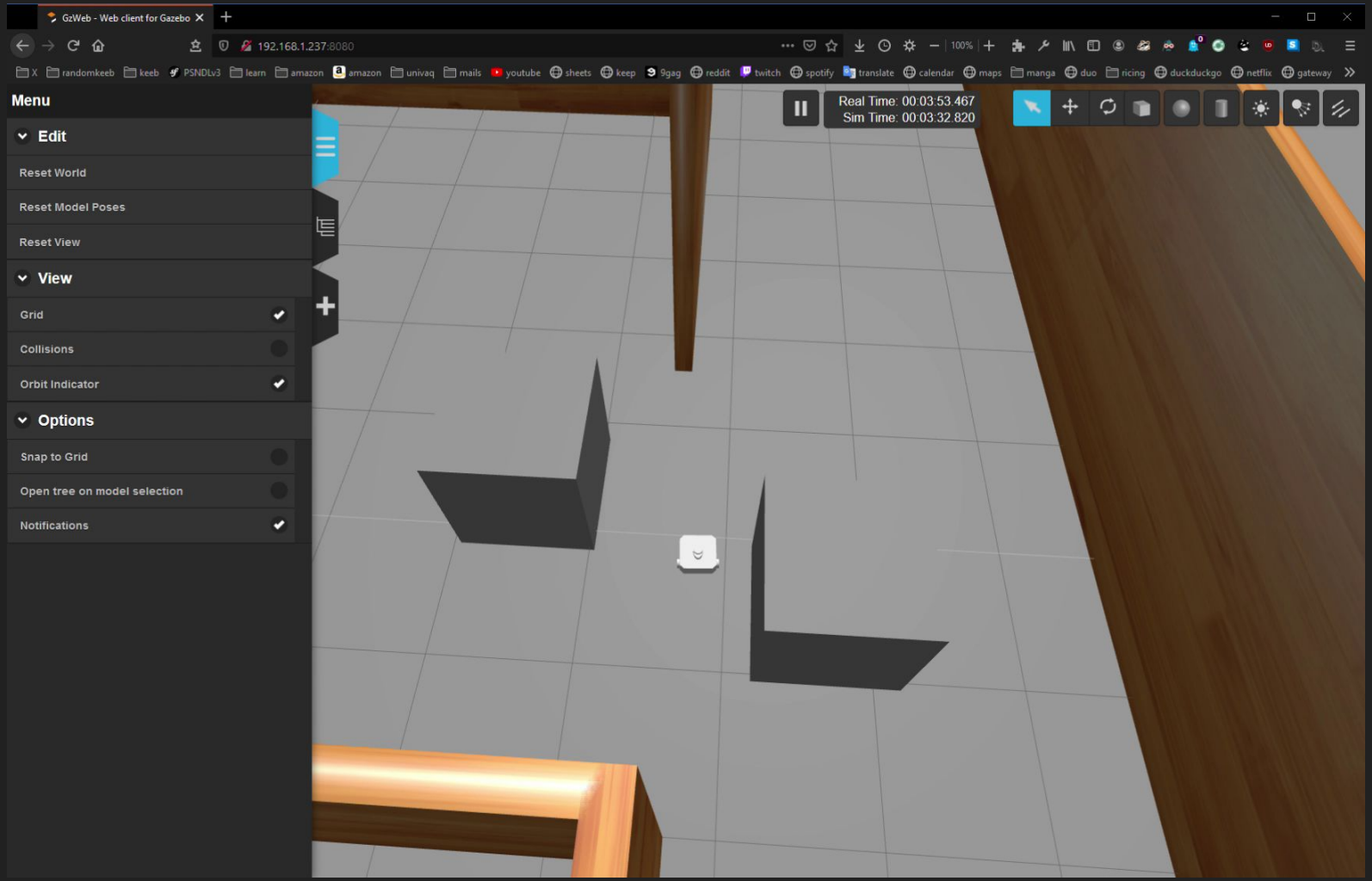
Architettura 4/4 (macchina simulator)

- si compone di un ambiente ROS 2 completo, del client web di simulazione 3D distribuita ‘gzweb’ e relativi modelli e ambiente 3D
- meccanismo di accesso dipendente da un’istanza server di Gazebo (gzserver, in remoto o in locale)
- fornisce un ambiente client virtuale in-browser webgl-powered di simulazione 3D che garantisce interattività utente (posizionamento/eliminazioni oggetti 3D, start/stop simulazione real-time, ecc…)

Validazione

- sviluppo e testing su architettura linux/amd64, su sistemi operativi Gnu/Linux (Void, Arch e derivate) e Windows 10 (WSL versione 2)
- la validazione conferma il soddisfacimento delle finalità e il raggiungimento dei seguenti risultati:
  - l'architettura, completamente costruibile, configurabile ed inizializzabile in maniera automatica, si dimostra estremamente flessibile sia in ambiente remoto che locale distribuito in rete locale
  - la simulazione virtuale in ambiente 3D si dimostra stabile, permettendoci di eseguire la parte di Gazebo Server e di Gazebo Web in modalità cross-os su macchine separate
  - la libreria wrapper per il Turtlebot 3 (attuazione e sensoristica) è stata sviluppata in maniera modulare: funzionale per l'utilizzo all'interno del progetto corrente ed espandibile per eventuali futuri utilizzi
  - l'attenta fase di testing sulla libreria di planning navigazionale evidenzia la notevole robustezza di quest'ultima:
    - il robot è in grado di muoversi autonomamente all'interno dell'ambiente simulato
    - è inoltre in grado di evitare accuratamente gli ostacoli statici e dinamici, anche inseriti manualmente a runtime dall'utente
    - non sono mai stati da noi riscontrati casi di urto del robot con oggetti dell'ambiente virtuale


1 2 3 4
W: (63% at TIM-21448749) 192.168.1.103 | /: 28,8 GiB | CPU: 81.57% / Mem: 1.55/7.66 GB | FULL 55,19% | *: 45% | P: 63% | 10/07/2021 - 12:47:42
GzWeb - Web client for Gazebo - Google Chrome
GzWeb - Web client for Ga
localhost:8080
App
Torrents
HDblog.it La te...
Master
Magistrale
Informatica
Nerding
Android
Asus Zenfone 3
S4 Mini
Utili
Musica
Random
Menu
Edit
Reset World
Reset Model Poses
Reset View
View
Grid
Collisions
Orbit Indicator
Options
Snap to Grid
Open tree on model selection
Notifications
Real Time: 00:08:17.913
Sim Time: 00:08:11.059


GzWeb - Web client for Gazebo
192.168.1.237:8080
Menu
Edit
Reset World
Reset Model Poses
Reset View
View
Grid
Collisions
Orbit Indicator
Options
Snap to Grid
Open tree on model selection
Notifications
Real Time: 00:03:53.467
Sim Time: 00:03:32.820

Robotica Fisica

Progetto 1

Introduzione

- costruzione di un’infrastruttura distribuita per la comunicazione basata su payload JSON tra un ambiente con contesto ROS 2 e uno senza contesto ROS 2
- utilizzo delle tecnologie di virtualizzazione e composizione Docker e Docker Compose
- utilizzo della tecnologia di datastore Redis con finalità di message brokering
- sviluppo e testing su architettura linux/arm/v7 mediante l’utilizzo di una macchina Raspberry PI versione 3 modello B revisione 1.2
- utilizzo di JSON come interfaccia di messaggistica tra il contesto ROS 2 e il contesto non ROS 2
- Il progetto corrente è propedeutico alla definizione e allo sviluppo del Progetto 2 di Robotica Fisica (lavoro preliminare introduttivo finalizzato alla creazione del suddetto meccanismo di comunicazione bidirezionale)

Finalità

- costruzione di un ambiente distribuito che permette la comunicazione tra un ambiente con contesto ROS 2 e un ambiente senza contesto ROS 2, mediante l'utilizzo di messaggistica con interfaccia JSON e del message broker Redis
- configurazione di un ambiente virtualizzato ROS 2 versione 'Dashing' su architettura linux/arm/v7 (Raspberry PI versione 3 modello B revisione 1.2)
- configurazione di un server virtualizzato Redis su architettura linux/arm/v7 (Raspberry PI versione 3 modello B revisione 1.2)
- garantire la comunicazione bidirezionale tra un semplice planner in contesto ROS 2 e un programma Python all'esterno del contesto ROS 2
- costruzione di un meccanismo di traduzione bidirezionale tra le interfacce custom di ROS 2 e l'interfaccia JSON
- definizione di un meccanismo di sinergia tra il message broker Redis e il custom message broker di ROS 2
- utilizzo della libreria di conversione messaggi nativa ROS 2 'rosidl_runtime_py' per il linguaggio Python 3.7.3

Architettura 1/4

- generazione di un’immagine completa dell’ambiente ROS versione 2 release ‘Dashing’ su layer Gnu/Linux Ubuntu LTS 18.04 (architettura linux/arm/v7) e deployment su piattaforma di cloud hosting Docker Hub
  - strutturazione gerarchica del deployment in 2 tag (full, redis)
- scrittura di un file di configurazione (docker-compose.yml) per l’inizializzazione e la gestione automatica della coreografia di containers mediante l’utilizzo di Docker Compose
- preparazione from-scratch della macchina Raspberry PI al funzionamento remoto o locale in rete locale (ssh, vnc)
- scrittura di un meccanismo di building automatico, debugging, auto-documentato e configurabile (Makefile gerarchici, config.mk) mediante l’utilizzo del tool Gnu Make

Architettura 2/4 (macchina publisher)

- composto da 2 packages ROS 2
  - package publisher:
    - contiene un semplice planner che si occupa dell'invio randomico di comandi di attuazione 'geometry_msgs/Twist' utilizzando la libreria turtlebot3 (sviluppata per il Progetto 1 di Robotica Virtuale)
  - package virt_redis_mirror:
    - contiene un wrapper per la libreria 'redis' (Python 3.7.3) che inizializza il contesto client di Redis come message broker, gestisce la connessione con il server Redis (locale o remoto) ed espone delle primitive di invio e ricezione messaggi su canali Redis
    - contiene un translater che si occupa:
      - della traduzione di messaggi con interfaccia custom 'geometry_msgs/Twist' pubblicati su topic ROS 2 '/cmd_vel' e del forwarding dei suddetti messaggi su topic Redis 'to_redis' utilizzando l'interfaccia JSON
      - della traduzione di messaggi con interfaccia JSON pubblicati su topic Redis 'from_redis' e del forwarding dei suddetti messaggi su topic ROS 2 'to_ros2' utilizzando l'interfaccia custom 'std_msgs/String'

Architettura 3/4 (macchina redis)

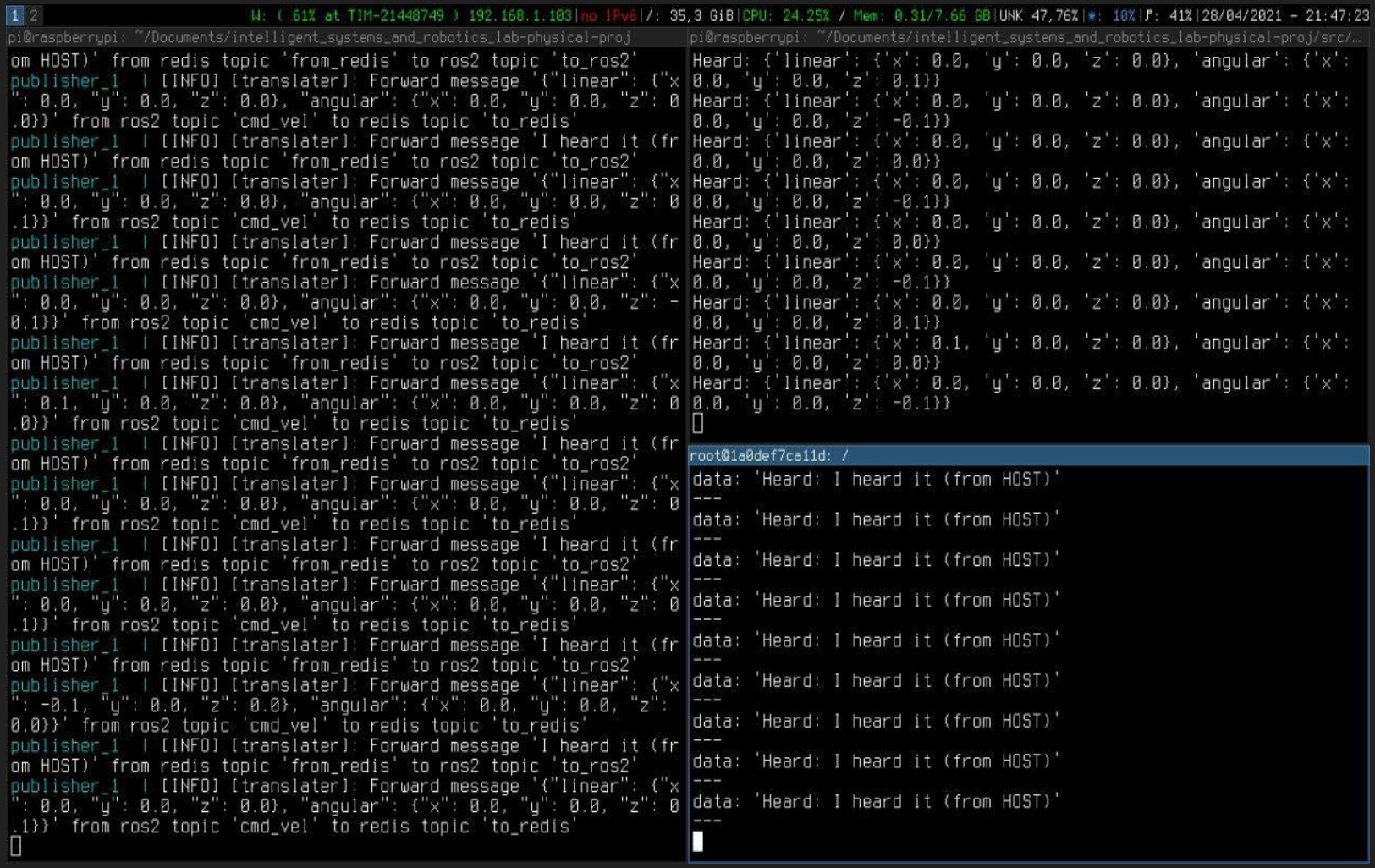
- container Docker costruito dall’immagine ufficiale di Redis (redis:6.2.2-buster) che espone il contesto server di Redis (utilizzato come message broker)
- consente la comunicazione bidirezionale in struttura distribuita tra il modulo con contesto ROS 2 e quello senza contesto ROS 2
- ci permette di disaccoppiare su architettura distribuita le componenti server e client di Redis (utilizzato come message broker)

Architettura 4/4 (parte host_redis_mirror)

- virtual environment per Python 3.7.3 composto da un programma che si occupa di ricevere i messaggi con interfaccia JSON provenienti dal container con contesto ROS 2 e di inviare delle stringhe di acknowledgment con interfaccia JSON
- fase di ricezione:
  - ascolto dal topic Redis ‘to_redis’
  - conversione dei messaggi con interfaccia JSON in dizionari Python
  - stampa a video
- fase di invio:
  - generazione di stringhe di acknowledgment
  - inserimento in payloads JSON
  - pubblicazione su topic Redis ‘from_redis’
  - conseguente invio dei payloads verso il modulo con contesto ROS 2

Validazione

- sviluppo e testing su architettura linux/arm/v7 (Raspberry PI), su sistema operativo Raspberry OS
- la validazione conferma il soddisfacimento delle finalità e il raggiungimento dei seguenti risultati:
  - l'architettura, completamente costruibile, configurabile ed inizializzabile in maniera automatica, si dimostra estremamente stabile all'interno dell'ambiente remoto virtuale generato all'interno della macchina Raspberry PI
  - la fase di testing sul meccanismo di comunicazione bidirezionale da noi implementato tra il contesto ROS 2 e un contesto neutro, evidenzia l'estrema flessibiltà ed utilità di quest'ultimo
  - non sono stati riscontrati problemi di comunicazione, errori di traduzione tra interfacce o perdite di messaggi
  - l'utilizzo dell'interfaccia JSON come lingua franca all'interno del meccanismo di comunicazione bidirezionale implementato si dimostra decisamente utile nel rendere opache le interfacce custom proprie del contesto ROS 2
- l'evoluzione del suddetto meccanismo di comunicazione bidirezionale (che vedremo nel Progetto 2 di Robotica Fisica) consiste nell'eliminare la sinergia esistente tra Redis ed il ROS 2 custom message broker ed utilizzare il solo Redis come unico message broker


1 2
W: (61% at TIM-21448749) 192.168.1.103|no IPv6|/: 35,3 GiB|CPU: 24.25% / Mem: 0.31/7.66 GB|UNK 47,76%|*: 10%|/: 41%|28/04/2021 - 21:47:23
pi@raspberrypi: ~/Documents/intelligent_systems_and_robotics_lab-physical-proj
om HOST)' from redis topic 'from_redis' to ros2 topic 'to_ros2'
publisher_1 | [INFO] [translater]: Forward message '{"linear": {"x
": 0.0, "y": 0.0, "z": 0.0}, "angular": {"x": 0.0, "y": 0.0, "z": 0
.0}}' from ros2 topic 'cmd_vel' to redis topic 'to_redis'
publisher_1 | [INFO] [translater]: Forward message 'I heard it (fr
om HOST)' from redis topic 'from_redis' to ros2 topic 'to_ros2'
publisher_1 | [INFO] [translater]: Forward message '{"linear": {"x
": 0.0, "y": 0.0, "z": 0.0}, "angular": {"x": 0.0, "y": 0.0, "z": 0
.1}}' from ros2 topic 'cmd_vel' to redis topic 'to_redis'
publisher_1 | [INFO] [translater]: Forward message 'I heard it (fr
om HOST)' from redis topic 'from_redis' to ros2 topic 'to_ros2'
publisher_1 | [INFO] [translater]: Forward message '{"linear": {"x
": 0.0, "y": 0.0, "z": 0.0}, "angular": {"x": 0.0, "y": 0.0, "z": -
0.1}}' from ros2 topic 'cmd_vel' to redis topic 'to_redis'
publisher_1 | [INFO] [translater]: Forward message 'I heard it (fr
om HOST)' from redis topic 'from_redis' to ros2 topic 'to_ros2'
publisher_1 | [INFO] [translater]: Forward message '{"linear": {"x
": 0.1, "y": 0.0, "z": 0.0}, "angular": {"x": 0.0, "y": 0.0, "z": 0
.0}}' from ros2 topic 'cmd_vel' to redis topic 'to_redis'
publisher_1 | [INFO] [translater]: Forward message 'I heard it (fr
om HOST)' from redis topic 'from_redis' to ros2 topic 'to_ros2'
publisher_1 | [INFO] [translater]: Forward message '{"linear": {"x
": 0.0, "y": 0.0, "z": 0.0}, "angular": {"x": 0.0, "y": 0.0, "z": 0
.1}}' from ros2 topic 'cmd_vel' to redis topic 'to_redis'
publisher_1 | [INFO] [translater]: Forward message 'I heard it (fr
om HOST)' from redis topic 'from_redis' to ros2 topic 'to_ros2'
publisher_1 | [INFO] [translater]: Forward message '{"linear": {"x
": 0.0, "y": 0.0, "z": 0.0}, "angular": {"x": 0.0, "y": 0.0, "z": 0
.1}}' from ros2 topic 'cmd_vel' to redis topic 'to_redis'
publisher_1 | [INFO] [translater]: Forward message 'I heard it (fr
om HOST)' from redis topic 'from_redis' to ros2 topic 'to_ros2'
publisher_1 | [INFO] [translater]: Forward message '{"linear": {"x
": -0.1, "y": 0.0, "z": 0.0}, "angular": {"x": 0.0, "y": 0.0, "z":
0.0}}' from ros2 topic 'cmd_vel' to redis topic 'to_redis'
publisher_1 | [INFO] [translater]: Forward message 'I heard it (fr
om HOST)' from redis topic 'from_redis' to ros2 topic 'to_ros2'
publisher_1 | [INFO] [translater]: Forward message '{"linear": {"x
": 0.0, "y": 0.0, "z": 0.0}, "angular": {"x": 0.0, "y": 0.0, "z": 0
.1}}' from ros2 topic 'cmd_vel' to redis topic 'to_redis'
pi@raspberrypi: ~/Documents/intelligent_systems_and_robotics_lab-physical-proj/src/...
Heard: {'linear': {'x': 0.0, 'y': 0.0, 'z': 0.0}, 'angular': {'x':
0.0, 'y': 0.0, 'z': 0.1}}
Heard: {'linear': {'x': 0.0, 'y': 0.0, 'z': 0.0}, 'angular': {'x':
0.0, 'y': 0.0, 'z': -0.1}}
Heard: {'linear': {'x': 0.0, 'y': 0.0, 'z': 0.0}, 'angular': {'x':
0.0, 'y': 0.0, 'z': 0.0}}
Heard: {'linear': {'x': 0.0, 'y': 0.0, 'z': 0.0}, 'angular': {'x':
0.0, 'y': 0.0, 'z': -0.1}}
Heard: {'linear': {'x': 0.0, 'y': 0.0, 'z': 0.0}, 'angular': {'x':
0.0, 'y': 0.0, 'z': 0.0}}
Heard: {'linear': {'x': 0.0, 'y': 0.0, 'z': 0.0}, 'angular': {'x':
0.0, 'y': 0.0, 'z': -0.1}}
Heard: {'linear': {'x': 0.0, 'y': 0.0, 'z': 0.0}, 'angular': {'x':
0.0, 'y': 0.0, 'z': 0.1}}
Heard: {'linear': {'x': 0.1, 'y': 0.0, 'z': 0.0}, 'angular': {'x':
0.0, 'y': 0.0, 'z': 0.0}}
Heard: {'linear': {'x': 0.0, 'y': 0.0, 'z': 0.0}, 'angular': {'x':
0.0, 'y': 0.0, 'z': -0.1}}
root@1a0def7ca11d: /
data: 'Heard: I heard it (from HOST)'
---
data: 'Heard: I heard it (from HOST)'
---
data: 'Heard: I heard it (from HOST)'
---
data: 'Heard: I heard it (from HOST)'
---
data: 'Heard: I heard it (from HOST)'
---
data: 'Heard: I heard it (from HOST)'
---
data: 'Heard: I heard it (from HOST)'
---
data: 'Heard: I heard it (from HOST)'

Robotica Fisica

Progetto 2

Introduzione

- costruzione di un'infrastruttura distribuita per la generazione automatica di librerie derivante dalla descrizione astratta di un robot target su file di configurazione in formato yaml
- utilizzo delle tecnologie di virtualizzazione e composizione Docker e Docker Compose
- utilizzo della tecnologia di datastore Redis con finalità di message brokering
- utilizzo di JSON come interfaccia di messaggistica tra il contesto ROS 2 e il contesto non ROS 2
- sviluppo e testing cross-architetturale: linux/arm/v7 mediante l'utilizzo di una macchina Raspberry PI versione 3 modello B revisione 1.2 e linux/amd64
- il modulo di analisi, validazione, interpretazione e generazione di codice sviluppato si chiama rcp (robot configuration parser)
- viene fornita una collezione di esempi (costruibili in maniera automatica):
  - viene generata in maniera autonoma una collezione di librerie a partire da un file di configurazione che descrive in maniera semplificata ed astratta il robot target Freenove 4WD Smart Car Kit for Raspberry PI
  - viene mostrata la comunicazione bidirezionale su architettura distribuita tra una simulazione di planner pacchettizzato in contesto ROS 2 ed un modulo semplificato per la gestione del robot Freenove 4WD Smart Car Kit for Raspberry PI mediante l'utilizzo della suddetta collezione di librerie

Finalità

- costruzione di un ambiente distribuito che permette la comunicazione tra un ambiente con contesto ROS 2 e un ambiente senza contesto ROS 2, mediante l'utilizzo di messaggistica con interfaccia JSON e del message broker Redis
- sviluppo di un modulo di analisi, validazione ed interpretazione di un file di configurazione in formato yaml che esponga un meccanismo descrittivo ad alto livello di un robot target (elenco sensori, elenco attuatori, elenco canali di comunicazione bidirezionali, elenco comandi, …)
- sviluppo di un meccanismo di generazione automatica di una collezione di librerie che gestisca la parte di comunicazione bidirezionale in modo autonomo (comunicazione distribuita tra contesti, message brokering, conversione tra interfacce, ecc…) e che possa essere utilizzata con immediatezza da un programmatore
- consentire la programmazione in contesto neutro di un robot target (anche assente o nuovo all'interno dell'ambiente ROS 2) mediante l'utilizzo delle librerie generate automaticamente dal suddetto file di configurazione
- rendere opaco:
  - il meccanismo di comunicazione bidirezionale distribuito tra contesti (ROS 2 e neutro)
  - l'utilizzo di JSON come lingua franca della comunicazione
  - l'utilizzo del precedente meccanismo di traduzione per la conversione tra interfacce di messaggistica (JSON e tipi custom propri dell'ambiente ROS 2)
- eliminare il meccanismo di sinergia tra il message broker Redis e il custom message broker di ROS 2 (sviluppato nel Progetto 1 di Robotica Fisica) ed utilizzare il solo Redis come unico message broker
- il risultato dovrà eseguire su architettura locale virtualizzata o remota in rete locale e in maniera cross-architetturale (solo su macchina linux/amd64, solo su macchina linux/arm/v7, comunicazione tra macchine linux/amd64 e linux/arm/v7)
- generazione di un immagine ROS 2 versione 'Dashing' multi-architetturale (linux/amd64 e linux/arm/v7)

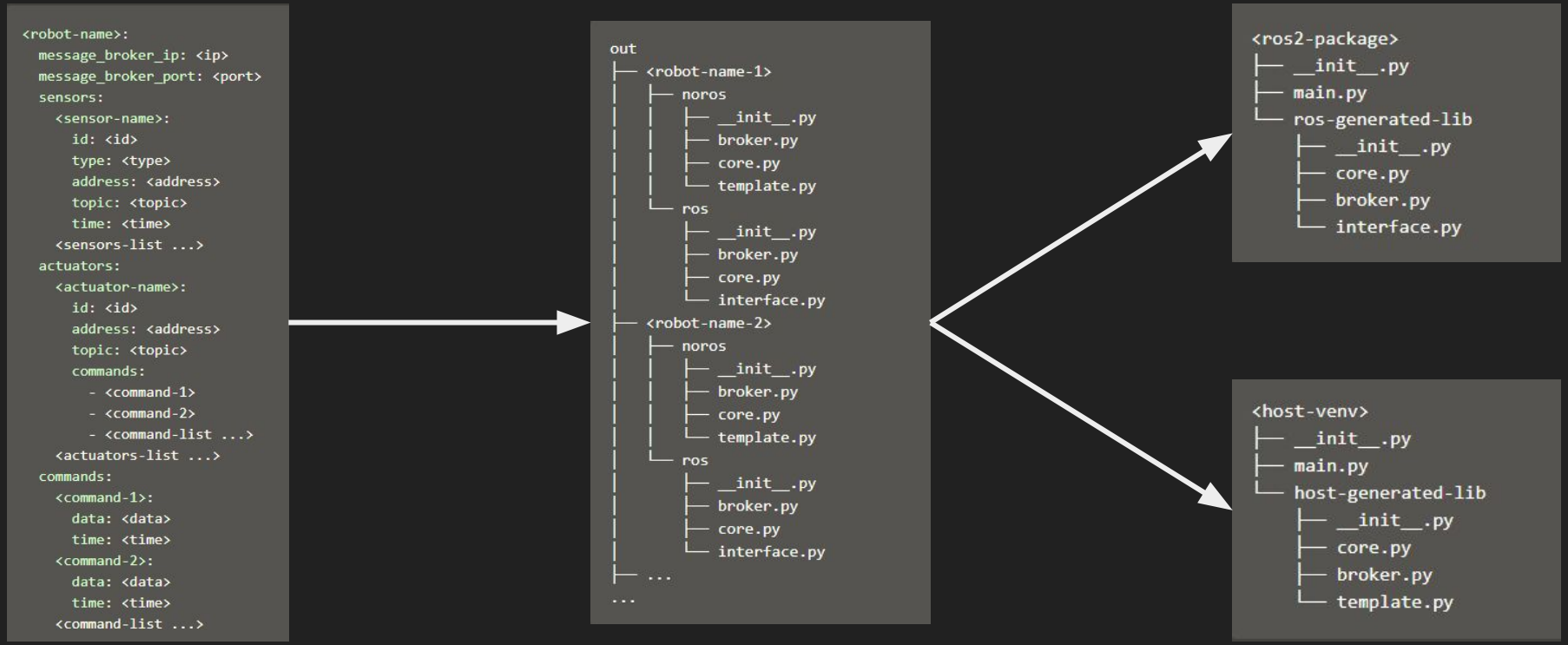
Architettura 1/6

- generazione di un'immagine base dell'ambiente ROS versione 2 release 'Dashing' su layer Gnu/Linux Ubuntu LTS 18.04 (multi-architettura linux/amd64 e linux/arm/v7) e deployment su piattaforma di cloud hosting Docker Hub
  - singolo tag (stable) corrispondente all'ambiente base ROS 2 + libreria client Redis per il linguaggio Python
- scrittura di un file di configurazione (docker-compose.yml) per l'inizializzazione e la gestione automatica della coreografia di containers mediante l'utilizzo di Docker Compose
- scrittura di un meccanismo di building automatico, debugging, auto-documentato e configurabile (Makefile gerarchici, config.mk) mediante l'utilizzo del tool Gnu Make
- sviluppo di una collezione di esempi (costruibili in maniera automatica):
  - viene generata in maniera autonoma una collezione di librerie a partire da un file di configurazione che descrive in maniera semplificata ed astratta il robot target Freenove 4WD Smart Car Kit for Raspberry PI
  - viene mostrata la comunicazione bidirezionale su architettura distribuita tra una simulazione di planner pacchettizzato in contesto ROS 2 ed un modulo semplificato per la gestione del robot Freenove 4WD Smart Car Kit for Raspberry PI mediante l'utilizzo della suddetta collezione di librerie

Architettura 2/6 (rcp)

- generazione del file di configurazione che descrive in maniera semplificata ed astratta il robot target Freenove 4WD Smart Car Kit for Raspberry PI in formato yaml
- attraverso le fasi che descriveremo in seguito, il software rcp si occupa della generazione di 2 collezioni di librerie distinte utilizzabili nei rispettivi contesti dal programmatore (contesto ROS 2 e contesto neutro)
- le collezioni di librerie si occupano, in entrambi i contesti, in maniera astratta e opaca al programmatore:
    - della gestione della comunicazione bidirezionale su architettura distribuita mediante l'utilizzo del solo Redis come message broker (gestione wrapper-based del client Redis come message broker ed esposizione delle primitive di invio e ricezione)
    - del binding tra contesti delle parti astratte (ad alto livello) del robot descritte nel file di configurazione (elenco sensori, elenco attuatori, operazioni di attuazione e sensoristica, ecc…)
    - della traduzione bidirezionale dei messaggi tra l'interfaccia JSON e le interfacce custom dell'ambiente ROS 2 implementate
    - del meccanismo di chiamata automatica delle callback definite dal programmatore (effettiva implementazione della logica delle operazioni e connessione alle librerie specifiche del robot target) secondo la policy di binding descritta precedentemente

Architettura 3/6 (schema generazione)



Architettura 4/6 (rcp: parsing)

- passaggio in input di almeno 1 file di configurazione in cui è presente la descrizione di un robot target
- parsing:
  - analisi e validazione sintattica e strutturale del file di configurazione
  - interpretazione semantica e strutturale
  - successiva costruzione delle primitive dati funzionali alla generazione di codice
- il software si compone di un meccanismo di runtime checking basato su assertions che restituisce eccezioni, hints per l'utente e stacktraces come conseguenza di eventuali errori durante la fase di parsing (analisi, validazione ed interpretazione)
- la restituzione di eventuali errori durante la fase di parsing è bloccante (non si procederà con la successiva fase di generazione di codice relativa al file di configurazione passato come argomento che contiene l'errore corrente)

Architettura 5/6 (rcp: generazione (contesto ROS 2))

- utilizza le primitive dati generate dopo la corretta conclusione della fase di parsing sul file di configurazione corrente per generare le 2 collezioni di librerie per i seguenti contesti (rispettivamente):
- contesto ROS 2:
  - viene generato un file broker.py che espone le primitive di invio e ricezione messaggi astraendo e rendendo opaca:
    - la gestione della comunicazione bidirezionale su architettura distribuita mediante l'utilizzo del solo Redis come message broker
    - il meccanismo di traduzione bidirezionale dei messaggi tra l'interfaccia JSON e le interfacce custom dell'ambiente ROS 2 implementate
  - viene generato un file interface.py che espone un meccanismo di costruzione dei messaggi standard:
    - header del messaggio indipendente dalle interfacce custom dell'ambiente ROS 2 (nome topic, nome comando, tipo del messaggio (tipizzazione forte))
    - body (data) costruito come un wrapper per ciascuna interfaccia custom ROS 2 implementata (ad esempio: geometry_msgs/Twist)
  - viene generato un file core.py che definisce la classe generale che astrae la descrizione formale del robot target nel file di configurazione in oggetto e che:
    - effettua il binding in contesto ROS 2 delle parti astratte (ad alto livello) del robot target descritte nel file di configurazione (elenco sensori, elenco attuatori, operazioni di attuazione e sensoristica, ecc…)
    - implementa il meccanismo di chiamata automatica delle callback definite dal programmatore (effettiva implementazione della logica delle operazioni in ambiente ROS 2) secondo la policy di binding descritta precedentemente
    - le callbacks possono essere definite dal programmatore e registrate a runtime

Architettura 6/6 (rcp: generazione (contesto neutro))

- contesto neutro (non ROS 2):
  - viene generato un file broker.py che espone le primitive di invio e ricezione messaggi astraendo e rendendo opaca:
    - la gestione della comunicazione bidirezionale su architettura distribuita mediante l'utilizzo del solo Redis come message broker
    - il meccanismo di traduzione bidirezionale dei messaggi tra l'interfaccia JSON e le interfacce custom implementate dal programmatore derivanti dalla connessione con le librerie fisiche del robot target (seguendo la metodologia standard di definizione dei messaggi (topic + payload))
  - viene generato un file template.py che espone una collezione di callbacks di default già connesse alle rispettive parti del robot target definite all'interno del file di configurazione
    - il programmatore ha il compito di implementare il corpo di tali metodi oppure può definirne di nuovi, registrandoli a runtime
  - viene generato un file core.py che definisce la classe generale che astrae la descrizione formale del robot target nel file di configurazione in oggetto e che:
    - effettua il binding in contesto neutro delle parti astratte (ad alto livello) del robot target descritte nel file di configurazione (elenco sensori, elenco attuatori, operazioni di attuazione e sensoristica, ecc…)
    - implementa il meccanismo di chiamata automatica delle callback definite dal programmatore (effettiva implementazione della logica delle operazioni in ambiente neutro) secondo la policy di binding descritta precedentemente

Validazione 1/2

- sviluppo e testing multi-architetturale: linux/amd64 e linux/arm/v7 (Raspberry PI), su sistema operativo Raspberry OS e Gnu/Linux (void, arch e derivate)
- la validazione conferma il soddisfacimento delle finalità e il raggiungimento dei seguenti risultati:
  - l'architettura, completamente costruibile, configurabile ed inizializzabile in maniera automatica, si dimostra estremamente flessibile e stabile sia in ambiente remoto sia locale virtuale, distribuito in rete locale
  - l'attenta fase di testing garantisce il risultato atteso in ogni tipo di coreografia in rete locale e su ogni tipo di interazione multi-architetturale (solo linux/amd64, solo linux/arm/v7, combinazione delle precedenti)
  - la fase di testing sul meccanismo di comunicazione bidirezionale da noi implementato tra il contesto ROS 2 e un contesto neutro, evidenzia l'estrema flessibiltà ed utilità di quest'ultimo nel garantire la possibiltà di programmare la logica relativa a robot esterni al contesto ROS 2 (per qualunque ragione: porting assente, produzione recente, ecc...) in contesto neutro
  - non sono stati riscontrati problemi di comunicazione, errori di traduzione tra interfacce o perdite di messaggi
  - l'utilizzo dell'interfaccia JSON come lingua franca all'interno del meccanismo di comunicazione bidirezionale implementato, si dimostra decisamente utile nel rendere opaco ed efficace il meccanismo di interazione tra le librerie del robot fisico in contesto neutro e le interfacce/contesto custom proprio dell'ambiente ROS 2
  - la flessibilità dell'architettura e il medio requisito in termini di risorse di quest'ultima, ci consente di utilizzare il medesimo risultato:
    - in soluzione distrbuita in locale, mediante virtualizzazione, su chip linux/amd64 o linux/arm/v7 (più comune), in cui l'intero stack esegue sul robot
    - o in soluzione distribuita in remoto con connessione verso una macchina linux/amd64 (più comune) o linux/arm/v7 con maggiori capacità computazionali, qualora vi sia la necessità di potenza di calcolo maggiore
  - il set minimale di primitive esposte consente al programmatore, con un numero esiguo di chiamate, di sviluppare codice per un robot target in contesto neutro e comunicare con il contesto ROS 2:
    - ignorando l'implementazione del meccanismo di comunicazione bidirezionale
    - ignorando la struttura architetturale distribuita che rappresenta la coreografia di esecuzione
    - ignorando la presenza e la traduzione tra interfacce (JSON, interfacce custom di ROS 2)
    - gestendo il binding tra l'implementazione logica (in entrambi i contesti) mediante il solo meccanismo unitario delle callback
    - ignorando il fatto che il robot non sia presente in ambiente ROS 2, per una qualsiasi motivazione

Validazione 2/2

- il software rcp è stato sviluppato seguendo il paradigma della suddivisione modulare semantica e garantisce la totale astrazione dalla logica operazionale in modo da garantirne l’espandibilità futura con sforzo minimo e alta flessibilità
- esempi:
  - implementazione di nuove porzioni del file di configurazione e successiva modifica della fase di parsing e generazione di codice
  - implementazione di ulteriori interfacce custom in ambiente ROS 2 e successivo binding tra contesti
  - ...

riccardo@k56cb:~/Documenti/Universita/Robotics/rcp/rcp/src

```
f and then reboot or run the command 'sysctl vm.overcommit_memory=1
' for this to take effect.
redis     | 1:M 27 May 2021 16:09:23.940 * Ready to accept connectio
ns
ros       | Starting >>> ros_usage_example
ros       | Finished <<< ros_usage_example [0.76s]
ros       |
ros       | Summary: 1 package finished [0.98s]
ros       | [INFO] [freenove]: received sensor data: [1, 0, 0]
ros       | [INFO] [freenove]: sending command turn_left
ros       |
ros       | [INFO] [freenove]: received sensor data: [0, 0, 1]
ros       | [INFO] [freenove]: sending command turn_right
ros       |
ros       | [INFO] [freenove]: received sensor data: [0, 1, 0]
ros       | [INFO] [freenove]: sending command go_forward
ros       |
ros       | [INFO] [freenove]: received sensor data: [0, 1, 0]
ros       | [INFO] [freenove]: sending command go_forward
ros       |
ros       | [INFO] [freenove]: received sensor data: [0, 1, 0]
ros       | [INFO] [freenove]: sending command go_forward
ros       |
ros       | [INFO] [freenove]: received sensor data: [1, 0, 0]
ros       | [INFO] [freenove]: sending command turn_left
ros       |
ros       | [INFO] [freenove]: received sensor data: [0, 0, 0]
ros       | [INFO] [freenove]: received sensor data: [1, 0, 0]
ros       | [INFO] [freenove]: sending command turn_left
ros       |
ros       | [INFO] [freenove]: received sensor data: [0, 0, 1]
ros       | [INFO] [freenove]: sending command turn_right
ros       |
ros       | [INFO] [freenove]: received sensor data: [1, 0, 0]
ros       | [INFO] [freenove]: sending command turn_left
ros       |
ros       | [INFO] [freenove]: received sensor data: [1, 0, 0]
ros       | [INFO] [freenove]: sending command turn_left
ros       |
ros       | [INFO] [freenove]: received sensor data: [0, 0, 0]
```

pi@raspberrypi: ~/Documents/rcp/rcp/src

```
(rcp) pi@raspberrypi:~/Documents/rcp/rcp/src $
(rcp) pi@raspberrypi:~/Documents/rcp/rcp/src $ make exec_remote_rpi

sending sensor data to ros:  [1, 0, 0] (expected ros command: turn_
left)
ros command: turn_left
ros twist payload linear:
- {'x': 0.0, 'y': 0.0, 'z': 0.0}
ros twist payload angular:
- {'x': 0.0, 'y': 0.0, 'z': 0.18647641246217783}

sending sensor data to ros:  [0, 0, 1] (expected ros command: turn_
right)
ros command: turn_right
ros twist payload linear:
- {'x': 0.0, 'y': 0.0, 'z': 0.0}
ros twist payload angular:
- {'x': 0.0, 'y': 0.0, 'z': -0.16543343020593237}

sending sensor data to ros:  [0, 1, 0] (expected ros command: go_fo
rward)
ros command: go_forward
ros twist payload linear:
- {'x': 0.5952217976646269, 'y': 0.0, 'z': 0.0}
ros twist payload angular:
- {'x': 0.0, 'y': 0.0, 'z': 0.0}

sending sensor data to ros:  [0, 1, 0] (expected ros command: go_fo
rward)
ros command: go_forward
ros twist payload linear:
- {'x': 0.7217794699336291, 'y': 0.0, 'z': 0.0}
ros twist payload angular:
- {'x': 0.0, 'y': 0.0, 'z': 0.0}

sending sensor data to ros:  [0, 1, 0] (expected ros command: go_fo
rward)
ros command: go_forward
ros twist payload linear:
- {'x': 0.5085235790042122, 'y': 0.0, 'z': 0.0}
```

```
1 2 3 4 5 6 7 8 9 []= zsh                                   R:7054 D:0 U:0 A:192.168.1.237 Thu 27-05-21 18:39:53

ros      |
ros      | [INFO] [freenove]: received sensor data: [1, 0, 0]
ros      | [INFO] [freenove]: sending command turn_left
ros      |
ros      | [INFO] [freenove]: received sensor data: [0, 0, 0]
ros      | [INFO] [freenove]: received sensor data: [1, 0, 0]
ros      | [INFO] [freenove]: sending command turn_left
ros      |
ros      | [INFO] [freenove]: received sensor data: [1, 0, 0]
ros      | [INFO] [freenove]: sending command turn_left
ros      |
ros      | [INFO] [freenove]: received sensor data: [0, 0, 1]
ros      | [INFO] [freenove]: sending command turn_right
ros      |
ros      | [INFO] [freenove]: received sensor data: [1, 0, 0]
ros      | [INFO] [freenove]: sending command turn_left
ros      |
ros      | [INFO] [freenove]: received sensor data: [0, 0, 0]
```

```
sending sensor data to ros:  [1, 0, 0] (expected ros command: turn_left)
ros command: turn_left
ros twist payload linear:
- {'x': 0.0, 'y': 0.0, 'z': 0.0}
ros twist payload angular:
- {'x': 0.0, 'y': 0.0, 'z': 0.43936875710804213}

sending sensor data to ros:  [0, 0, 0] (expected ros command: idle (no command))
sending sensor data to ros:  [0, 0, 0] (expected ros command: idle (no command))
sending sensor data to ros:  [0, 0, 0] (expected ros command: idle (no command))
sending sensor data to ros:  [0, 0, 1] (expected ros command: turn_right)
ros command: turn_right
ros twist payload linear:
- {'x': 0.0, 'y': 0.0, 'z': 0.0}
ros twist payload angular:
- {'x': 0.0, 'y': 0.0, 'z': -0.5609665043280284}
```