Università degli Studi di Palermo

DOCUMENTAZIONE TECNICA

INTELLIGENZA ARTIFICIALE 2 (6 CFU) & ROBOTICA (6 CFU)

A cura di:

Gabriele Bova

Paolo Manuele Gulotta

Andrea Spinelli

Vincenzo Zizzo

CdL: Ingegneria Informatica LM

Anno Accademico: 2025/2026

Sommario

Capitolo 1: Introduzione

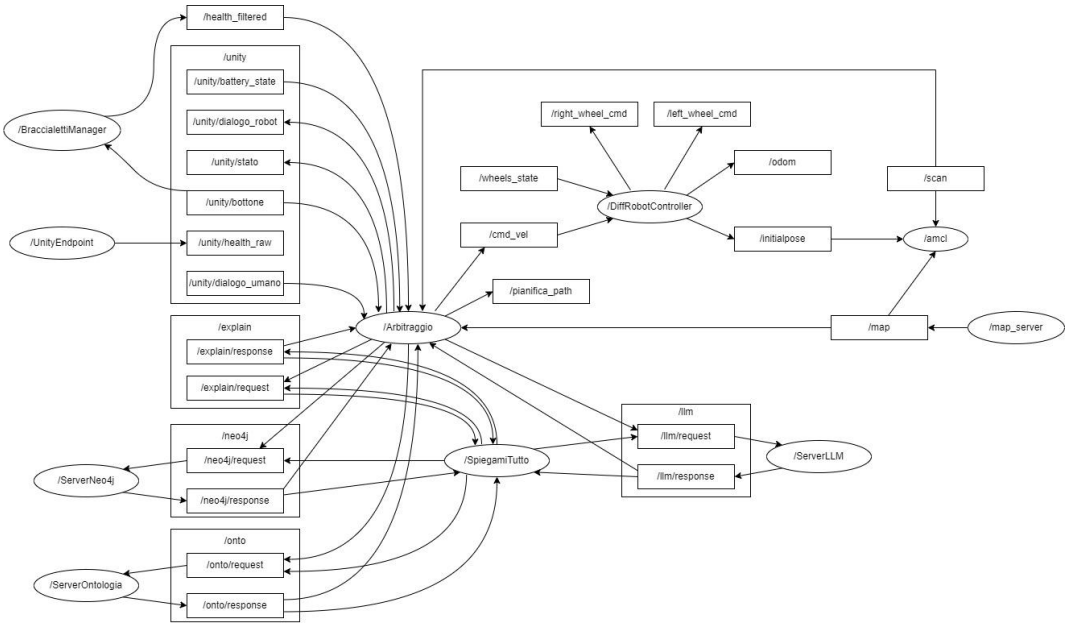
1.1 Obiettivo

Lo scopo del nostro progetto è stato quello di implementare in un ambiente simulato un Robot sociale autonomo chiamato **Pippor**, che affianca il personale umano della reception, progettato per operare all'interno di strutture ricettive nel contesto dell'ospitalità. Il compito di Pippor è quello di muoversi autonomamente nella struttura per assistere i clienti a 360 gradi: dall'accoglienza personalizzata alla gestione dei guasti e delle emergenze mediche. Durante il suo servizio, Pippor sarà costantemente connesso a dei braccialetti smart forniti agli ospiti. Questi ultimi sono strumenti essenziali, capaci non solo di identificare l'utente, ma di monitorarne in tempo reale i parametri vitali, allertando il robot in caso di malori o situazioni di pericolo. Durante l'interazione, Pippor raccoglierà dati sul contesto ambientale e sanitario, sfruttando un'intelligenza ibrida per dialogare empaticamente e ragionare logicamente, con l'obiettivo finale di garantire un soggiorno sicuro e di agire tempestivamente, chiamando soccorsi o supporto tecnico solo quando strettamente necessario.

1.2 Architettura

L'architettura complessiva del progetto si fonda su un paradigma distribuito che disaccoppia nettamente il livello di simulazione fisico-visiva dal livello di controllo logico-cognitivo. Per realizzare questo ambiente, abbiamo integrato due ecosistemi distinti: **Unity**, che funge da motore di simulazione per la fisica e i sensori, e **ROS2**, che ospita l'intero stack di intelligenza del robot. Il ponte tra questi due mondi è realizzato tramite **Unity Robotics Hub**, sfruttando il protocollo TCP/IP per la serializzazione bidirezionale dei messaggi (tramite ROS-TCP-Connector lato Unity e ROS-TCP-Endpoint lato ROS2).

A supporto delle capacità decisionali e semantiche del robot, l'ambiente ROS2 è stato potenziato integrando un ecosistema di librerie Python e servizi esterni dedicati. La persistenza della conoscenza e la gestione delle relazioni topologiche e sociali sono affidate al graph database **Neo4j**, che funge da memoria a lungo termine del sistema. Per colmare il divario tra i dati grezzi e il ragionamento logico, vengono impiegate le librerie **rdflib** e **owlready2**, essenziali per la manipolazione di ontologie, la gestione di triple RDF e la creazione dinamica di file OWL per il reasoner. Sul fronte dell'interazione uomo-macchina, il modulo di dialogo sfrutta **google-genai** per interfacciarsi con i Large Language Models (LLM), garantendo una comprensione naturale del linguaggio, mentre la sicurezza e la configurazione delle variabili d'ambiente (come API Key e percorsi di sistema) sono gestite in modo robusto tramite **python-dotenv**.



1.3 Ambiente (Hotel)

L'ambiente di simulazione è stato progettato per rappresentare uno scenario realistico di servizio in un hotel. La scelta di questo setting non è puramente estetica, ma funzionale: offre una varietà di spazi con caratteristiche geometriche e di navigazione distinte (corridoi, aree aperte, ostacoli dinamici) ideali per testare algoritmi di localizzazione e pianificazione del movimento.

1.3.1 Luoghi importanti e topologia

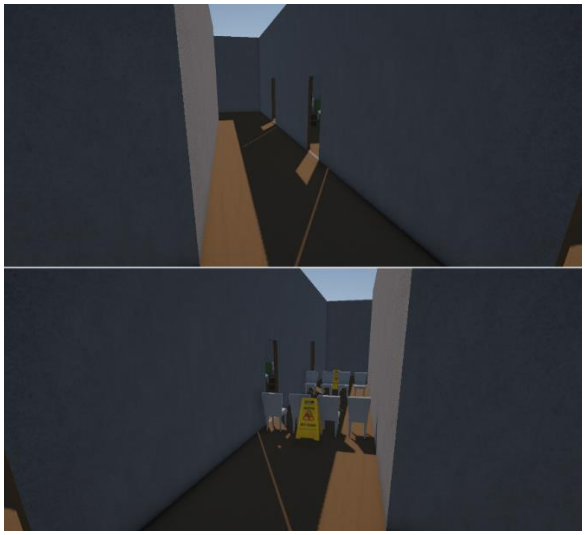
L'architettura della scena Unity è stata suddivisa in tre zone macroscopiche, ognuna delle quali presenta sfide specifiche per il sistema di navigazione autonoma del robot Pippor. La continuità spaziale è garantita da un sistema di coordinate coerente (Unity World Space) che viene poi mappato nel sistema di riferimento di ROS2 per la navigazione.

Reception (punto di ingresso)

È un'area caratterizzata da un bancone statico che funge da barriera fisica e visiva. L'area è ampia per permettere manovre di rotazione sul posto (zero-turn) tipiche della guida differenziale. Qui è collocato il robot. È il luogo dove idealmente inizia l'interazione uomo-macchina. La geometria del bancone è stata impostata con collider statici precisi per evitare compenetrazioni durante le fasi di docking o avvicinamento.

Corridoi

Rappresentano la sfida principale di path planning, collegando tutti i luoghi di interesse e costringono il robot a muoversi in uno spazio confinato. La larghezza dei corridoi riduce il margine di errore per l'algoritmo di localizzazione. Qualsiasi deviazione significativa dalla traiettoria ottimale rischia di portare il robot in collisione con le pareti. Qui sono stati posizionati dei waypoints invisibili utilizzati dagli NPC per il loro pattugliamento. Questo trasforma i corridoi in aree ad alto traffico, dove il robot deve costantemente adattare la traiettoria per evitare collisioni dinamiche.



Coffee room

È un ambiente non strutturato e denso. A differenza dei corridoi, qui gli ostacoli non sono solo perimetrali. La presenza di tavoli rotondi e sedie crea una mappa di occupazione complessa. Le gambe dei tavoli e delle sedie sono ostacoli sottili che possono essere difficili da rilevare per sensori a bassa risoluzione, richiedendo un raycasting ad alta frequenza. Quest'area funge da punto di aggregazione naturale per gli NPC, permettendo di testare la capacità del robot di navigare in spazi ristretti e potenzialmente affollati. Tuttavia, la simulazione prevede che gli eventi critici (ad esempio un malore) siano distribuiti logicamente in tutto l'hotel in quanto sono eventi stocastici che possono verificarsi casualmente in un qualsiasi punto della mappa, inclusa questa sala, obbligando il robot ad una risposta adattiva ovunque si trovi.

Stazione di ricarica

È il punto dove staziona il robot nei momenti di riposo. Non è solo un oggetto passivo, ma un volume logico. È stato implementato un box collider con proprietà IsTrigger. Il sistema è progettato affinché il robot debba fisicamente entrare in questo volume per attivare la logica di ricarica. Questo ha richiesto un tuning preciso dell'altezza del collider della stazione per garantire l'intersezione con il collider del robot (che si trova sollevato da terra a causa delle ruote), risolvendo problemi di mancata rilevazione dovuti a collider troppo bassi o piatti (plane).

1.3.2 Ostacoli

La navigazione nell'hotel si basa sul sistema NavMesh (Navigation Mesh) di Unity, che definisce le aree camminabili.

Distinguiamo 2 tipi di ostacoli: statici e dinamici.

Ostacoli statici

Tutti gli elementi architettonici (muri, colonne, bancone, tavoli) sono stati marcati come navigation static. Durante il processo di "baking" della NavMesh, Unity calcola le superfici percorribili sottraendo il volume di questi ostacoli più un raggio di sicurezza (agent radius).

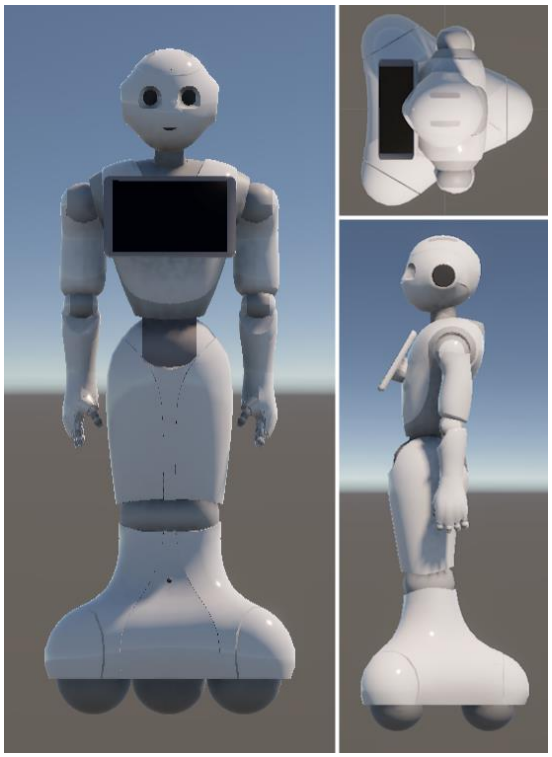
Ostacoli dinamici

La vera complessità è introdotta dagli NPC. Ogni persona è governata da uno script NPCController personalizzato. Essi non sono semplici ostacoli mobili, ma agenti intelligenti che utilizzano la NavMesh. Gli NPC hanno stati logici (pattugliamento, attesa, scenari di emergenza). Quando si attiva uno scenario, l'NPC diventa un ostacolo statico improvviso in una posizione non prevista dalla mappa statica.

Gli NPC non sono entità anonime; le loro identità (nome, cognome, ID) e parametri vitali sono connessi dinamicamente a **Neo4j**. Questo aggiunge un livello di semantica all'ambiente: il robot non vede solo un "cilindro" da evitare, ma un'entità con cui può comunicare via ROS2 (topic /unity/health_raw e /unity/dialogo). L'implementazione delle animazioni è servita per garantire una maggiore realisticità alla simulazione. I personaggi non si limitano a traslare nello spazio, ma eseguono movimenti complessi (camminata, attesa, caduta) che mimano il comportamento umano. Parallelamente, è stata posta grande attenzione alla configurazione dei collider (i volumi di collisione fisica). Questi sono essenziali per evitare problemi di navigazione: impediscono ai personaggi di attraversare muri o oggetti e assicurano che il robot rilevi correttamente la presenza fisica delle persone, permettendo al sistema di guida di calcolare percorsi sicuri senza compenetrazioni.

1.4 Robot (Pippor)

Il protagonista della simulazione è una replica digitale del robot Pepper di SoftBank Robotics. La sua implementazione in Unity non è una semplice animazione, ma una simulazione fisica completa basata su giunti, forze e comunicazione bidirezionale con ROS2. Il robot è stato modellato come un sistema multi-corpo (Multi-Body System).



1.4.1 Struttura gerarchica

La gerarchia della Prefab di Unity riflette la cinematica del robot reale.

- **Chassis (base):** Il corpo principale, che ospita il computer di bordo simulato e le batterie. Ha una massa definita di circa 28 kg, un fattore determinante per la simulazione fisica che ha richiesto un tuning avanzato dei motori.

- **Batteria (power management):** È stato implementato un sottosistema energetico tramite lo script RobotBattery. Questo componente non si limita a decrementare un contatore, ma simula cicli di carica/scarica basati sul tempo e sull'interazione con la stazione di ricarica. I dati vengono serializzati in formato **JSON** ({"level": 85, "is_charging": true}) e inviati a ROS2, permettendo al sistema di prendere decisioni autonome.
- **Ruote (wheels):** Il sistema di locomozione, nonché il cuore ingegneristico della simulazione. Usa una configurazione a guida differenziale (2 ruote motrici e 1 omnidirezionale passiva per l'equilibrio).

1.4.2 Sensori

La percezione ambientale del robot è affidata a un **LiDAR 2D** simulato, che costituisce il sensore cardine per le operazioni di mapping e navigazione autonoma. All'interno dell'ambiente Unity, il funzionamento fisico del dispositivo è riprodotto fedelmente attraverso la tecnica del **raycasting**: uno script dedicato emette ciclicamente una scansione radiale di raggi invisibili su un piano orizzontale, coprendo il campo visivo del robot. Il motore fisico calcola in tempo reale le intersezioni tra questi vettori e i collider degli elementi scenici (siano essi ostacoli statici come muri o dinamici come gli NPC), determinando la distanza esatta di ogni punto di impatto. Questi dati geometrici vengono infine serializzati nel formato standard sensor_msgs/LaserScan e trasmessi al nodo ROS2, permettendo allo stack di navigazione di ricostruire la geometria degli ostacoli circostanti con la stessa logica di un sensore hardware reale.

1.4.3 Ruote e guida differenziale

Il sistema di locomozione è il cuore ingegneristico della simulazione. Pippor utilizza una configurazione a **guida differenziale**.

Per la simulazione fisica delle ruote motrici, abbiamo abbandonato i classici HingeJoint in favore dei componenti **ArticulationBody** di Unity. Essi offrono una simulazione molto più stabile e precisa per le catene cinematiche robotiche, utilizzando un solver fisico dedicato (Featherstone) che riduce il "jittering" e gli errori di calcolo nelle giunzioni. Ogni ruota è controllata in velocità (TargetVelocity) attraverso il drive interno del componente.

1.4.4 La sfida della massa

Una delle sfide tecniche maggiori riscontrate durante lo sviluppo è stata la gestione della massa.

Inizialmente, con una massa realistica di 28 kg per lo chassis, il robot risultava "pigro", lento ad accelerare e incapace di frenare tempestivamente, scivolando come su ghiaccio o muovendosi a scatti. Questo avveniva perché i parametri di default di Unity sono tarati per oggetti leggeri (~1 kg).

La soluzione è stata quella di ricalibrare manualmente i parametri dell'azionamento (xDrive) nello script DiffDriveRobot:

- **Force Limit (torque):** Elevato a valori molto alti (2000). Questo non aumenta la velocità massima, ma fornisce al motore la "coppia" (N·m) necessaria per vincere l'inerzia di un corpo da 28 kg.
- **Damping (smorzamento):** Impostato a valori elevati (500). Nel controllo in velocità, il damping agisce come un guadagno proporzionale: definisce quanto "aggressivamente" il motore cerca di annullare la differenza tra velocità attuale e velocità target. Senza un damping elevato, il motore non applicava abbastanza forza frenante per arrestare la massa del robot.

- **Massa delle ruote:** Abbiamo aumentato la massa delle singole ruote a circa 1.5 kg. Un rapporto di massa troppo elevato tra corpo (28kg) e ruote (pochi grammi) destabilizza il solver fisico. Bilanciando le masse, la simulazione è diventata fluida e reattiva.

1.4.5 Watchdog e sicurezza

Nello script di controllo (DiffDriveRobot.cs), è stato implementato un sistema di Safety Watchdog. Se il robot non riceve comandi di velocità dai topic ROS (/cmd_vel o specifici per ruota) per più di un secondo (a causa di latenza di rete o crash del nodo di navigazione), il robot non continua a muoversi all'infinito con l'ultimo comando ricevuto. Invece di un blocco istantaneo (che causerebbe ribaltamenti o slittamenti irrealistici), il codice applica una decelerazione progressiva, riducendo la targetVelocity delle ruote verso zero in modo controllato, simulando l'attrito e la frenata rigenerativa dei motori reali.

Capitolo 2: Robotica

In questo capitolo tratteremo più nello specifico gli argomenti del corso che ci sono stati utili per la realizzazione del progetto, di come sono stati implementati e di come la loro coesione permetta di costruire un sistema autonomo ed efficiente.

2.1 Generazione della mappa (MapExporter)

Uno dei problemi fondamentali nella simulazione robotica è garantire che il robot "veda" il mondo esattamente come lo vede il simulatore. In un ambiente reale, un robot costruisce la mappa esplorando (SLAM). In un ambiente simulato, abbiamo già la verità assoluta (la scena Unity), ma ROS2 non può leggerla direttamente.

Il modulo **MapExporter** è stato sviluppato per colmare questo divario, permettendo di convertire istantaneamente l'ambiente 3D di Unity in una mappa 2D standard (formato .png + .yaml) comprensibile dallo stack di navigazione di ROS2.

È essenziale perché:

- Evita di dover guidare manualmente il robot per ore per mappare l'hotel tramite SLAM.
- La mappa generata è matematicamente perfetta, priva di errori di deriva sensoriale o loop-closure tipici del mapping manuale.
- Garantisce che la mappa statica di ROS2 corrisponda perfettamente ai collider di Unity.

2.1.1 Funzionamento tecnico

Il MapExporter non è un semplice screenshot dall'alto, ma un processo di scansione fisica basato sulle collisioni.

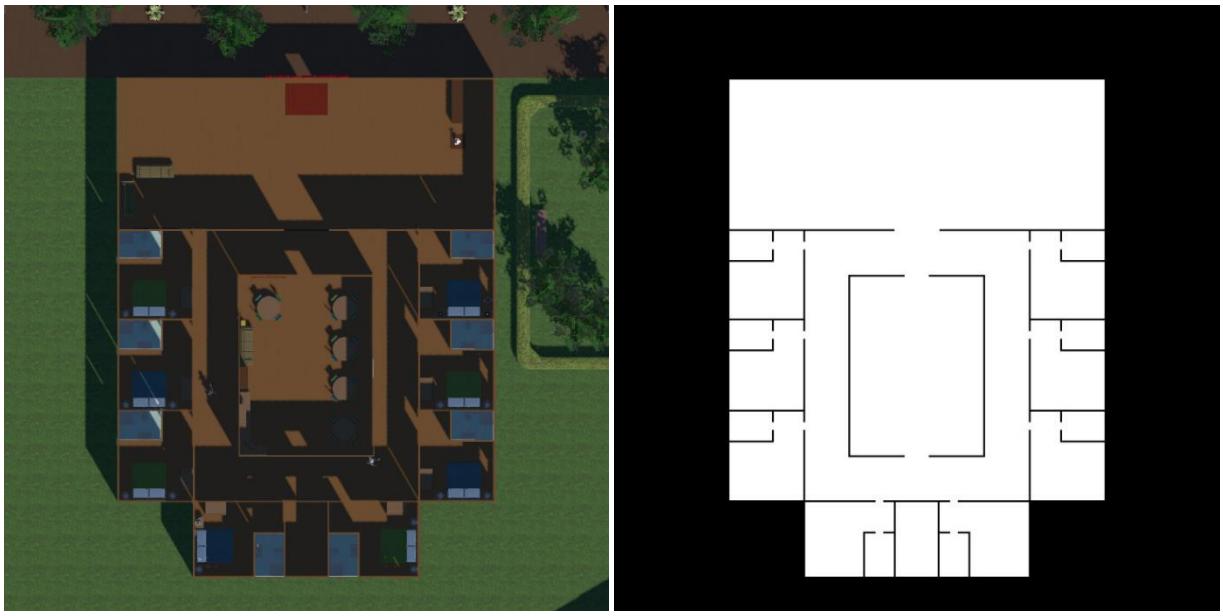
Il primo processo è quello di scansione. Lo script esegue un "taglio" orizzontale dell'ambiente a un'altezza specifica (configurabile, tipicamente all'altezza del LiDAR del robot).

1. Viene definita una griglia virtuale che copre l'intera area dell'hotel con una risoluzione specifica (circa 3.9 cm per pixel).
2. Per ogni cella della griglia, l'algoritmo verifica se c'è un oggetto fisico in quel punto. Questo avviene controllando la presenza di collider nel layer "Obstacle" o "Default".
3. Classificazione:
    - **Occupato (nero):** Se viene rilevato un muro.
    - **Libero (bianco):** Se lo spazio è vuoto e navigabile.
    - **Ignoto (nero):** Aree fuori dai confini o non scansionate.

Al termine della scansione, il sistema esporta due file essenziali per ROS2:

- **Mappa immagine (map_ros.png):** Un file immagine in scala di grigi dove i pixel neri rappresentano ostacoli e i bianchi spazio libero.
- **Metadati (map_ros.yaml):** Un file di configurazione che descrive come interpretare l'immagine. Contiene parametri critici come:
    - resolution: La dimensione di un pixel in metri.
    - origin: Le coordinate [x, y, θ] del pixel in basso a sinistra rispetto all'origine del mondo Unity/ROS2, essenziale per allineare la mappa visualizzata su RViz con la posizione reale del robot in Unity.

L'utilizzo del MapExporter ha permesso di iterare rapidamente sul design dell'hotel. Ogni volta che veniva spostato un muro in Unity, bastava un click per rigenerare la mappa di navigazione, permettendo di testare immediatamente i nuovi percorsi con l'algoritmo di pianificazione (Global Planner) senza configurazioni aggiuntive.



2.2 Sistema di arbitraggio

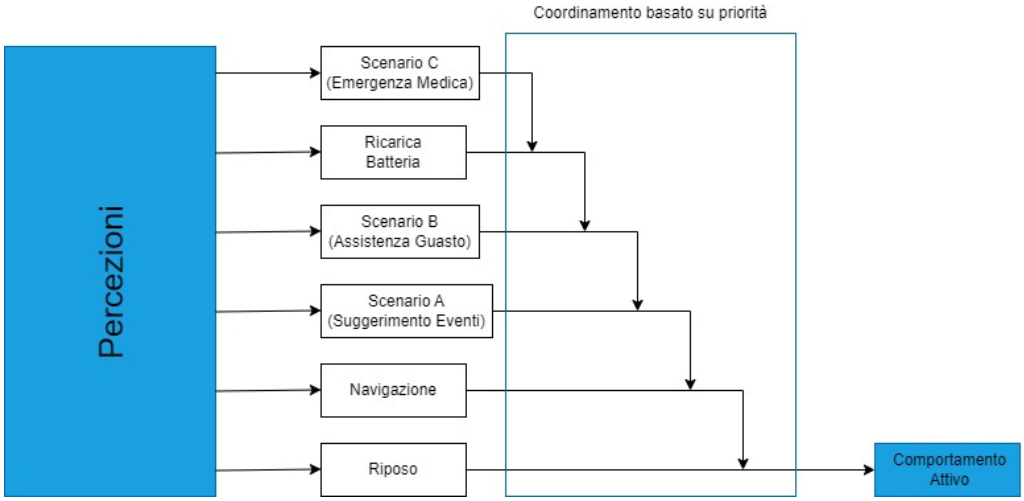
Il nodo Arbitraggio costituisce il nucleo decisionale dell'architettura di controllo, orchestrando le attività del robot attraverso un ciclo di valutazione ad alta frequenza (10 Hz) basato su un paradigma di priorità fisse. A differenza delle macchine a stati finiti tradizionali, questo sistema adotta una struttura a soppressione (subsumption) definita da una lista ordinata di comportamenti: ad ogni iterazione, il sistema scansiona i trigger partendo dall'alto e attiva esclusivamente il primo comportamento che soddisfa le condizioni logiche, inibendo l'esecuzione di tutti i livelli sottostanti.

La gerarchia è stata progettata con un preciso criterio di sicurezza: al vertice assoluto (indice 0) è stato collocato lo **Scenario C** (gestione emergenze). Questa scelta riflette un vincolo etico e operativo imprescindibile: la salvaguardia della salute dell'ospite deve prevalere su qualsiasi altra funzione, inclusa la sopravvivenza del robot stesso. L'attivazione di questo scenario, governata sia dalla pressione manuale del pulsante di soccorso (reattiva) che dal rilevamento automatico di anomalie biometriche critiche (proattiva), provoca l'immediata sospensione di ogni altra attività.

Immediatamente subordinata all'emergenza troviamo la **Ricarica Batteria** (indice 1). Il comportamento di ricarica scatta quando l'autonomia scende sotto la soglia di sicurezza, ma la sua posizione secondaria rispetto allo Scenario C implica che il robot sacrificherà la propria autonomia (rischiando lo spegnimento) pur di non abbandonare un ospite in pericolo di vita.

A livello intermedio (indici 2 e 3) si collocano gli scenari di interazione sociale: lo **Scenario B** (assistenza guasti) e lo **Scenario A** (rilevamento interessi e suggerimento eventi). Per questi comportamenti, il codice implementa un meccanismo di esclusione reciproca per evitare incoerenze durante il dialogo, pur mantenendo sempre la possibilità di essere interrotti da un'emergenza medica improvvisa e/o dallo scaricamento della batteria. Qualora ciò si verificasse l'interazione verrebbe delegata al personale umano della reception. Infine, alla base della piramide (indici 4 e 5), troviamo i comportamenti di routine come la **Navigazione** e il **Riposo**, attivati solo in assenza di richieste prioritarie. I comportamenti

superiori possono se necessario “iniettare” informazioni ai livelli inferiori, per esempio possono fornire alla navigazione la destinazione da raggiungere.



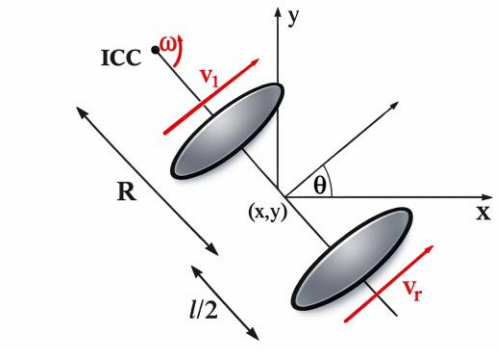
2.3 Sistema di guida differenziale

Il controllo di basso livello della base del robot è gestito dal nodo DiffRobotController, il quale funge da interfaccia bidirezionale tra la logica di navigazione (che ragiona in termini di velocità lineari e angolari complessive) e gli attuatori fisici (che richiedono comandi specifici per singola ruota). L'architettura del nodo si fonda sul modello cinematico del **pilota differenziale** (Differential Drive), parametrizzato sulle specifiche fisiche del robot: il raggio delle ruote (R) e la distanza interasse (L).

Il funzionamento del controller si divide in due pipeline principali: la **cinematica inversa per l'attuazione** e la **cinematica diretta per la stima della posa**.

Per quanto riguarda l'attuazione (metodo command_turn), il sistema sottoscrive i comandi di velocità (/cmd_vel) e li traduce in velocità angolari per i motori destro e sinistro; la logica applica le equazioni del modello unicycle, distribuendo la velocità lineare e quella angolare sulle due ruote in funzione della carreggiata, permettendo al robot di curvare variando la velocità relativa delle due ruote motrici.

Parallelamente, il sistema esegue la stima odometrica (metodo compute_odometry) leggendo i feedback degli encoder (/wheels_state). L'implementazione adottata è robusta rispetto alle singolarità matematiche, infatti l'algoritmo valuta dinamicamente la differenza di velocità tra le ruote per distinguere tra moto rettilineo e curvilineo. Nel caso di moto curvilineo, viene calcolato il **Centro Istantaneo di Curvatura (ICC)** per aggiornare la posa (x, y, θ) lungo un arco di circonferenza esatto, garantendo una precisione superiore rispetto alla semplice integrazione lineare, mentre nel caso di moto rettilineo si applica una integrazione standard per evitare divisioni per zero.



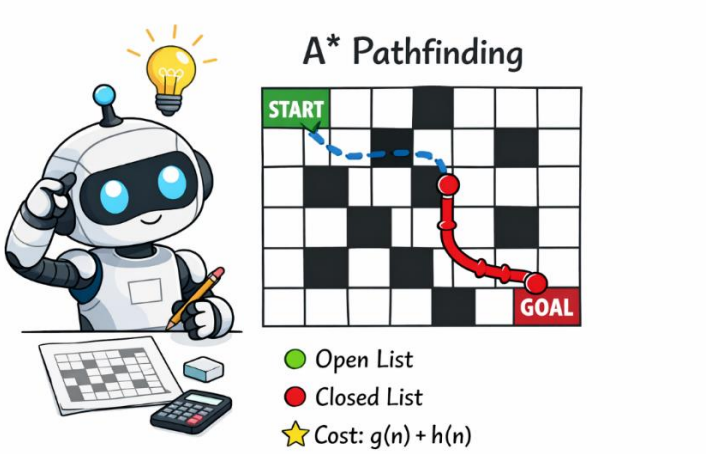
A completamento dell'infrastruttura ROS2, il nodo gestisce la pubblicazione delle trasformate (TF) necessarie: trasmette dinamicamente la trasformazione tra il sistema di riferimento odometrico e la base del robot, pubblica le trasformate statiche per il LiDAR e mantiene uno storico delle pose (Path) per la visualizzazione della traiettoria effettiva in RViz, occupandosi inoltre di inizializzare il sistema di localizzazione AMCL alla posa di start predefinita.

2.4 Pianificazione

Per il modulo di navigazione, è stata sviluppata un'implementazione personalizzata dell'algoritmo **A***, ottimizzata per operare su mappe a griglia di occupazione (Occupancy Grid) con connettività a 8 vicini. A differenza di pianificatori più semplici limitati ai movimenti cardinali, il nostro sistema sfrutta una topologia che permette spostamenti diagonali; per gestire correttamente i costi di attraversamento, l'algoritmo assegna un peso di 1.0 ai passi ortogonali e di $\sqrt{2}$ a quelli diagonali, utilizzando coerentemente una **euristica Octile** per la stima del costo residuo (h), che garantisce l'ammissibilità del percorso minimizzando l'errore di stima rispetto alla distanza Euclidea o Manhattan.

Un aspetto critico della sicurezza di navigazione è gestito attraverso un controllo esplicito di Corner Cutting Prevention: prima di validare un movimento diagonale, l'algoritmo verifica che le due celle cardinali adiacenti all'angolo di svolta siano libere. Questo impedisce al robot di tentare traiettorie che "taglierebbero" lo spigolo di un ostacolo, evitando collisioni fisiche in passaggi stretti.

L'implementazione offre inoltre una flessibilità operativa nella gestione dei dati di mappa: oltre a distinguere tra spazio libero (0) e occupato (100), è stato introdotto il flag allow_unknown per decidere dinamicamente se considerare le celle inesplorate (-1) come attraversabili o meno, adattando il comportamento in base alla fase. Infine, per ottimizzare i tempi di calcolo durante le operazioni di pianificazione, il metodo supporta un vincolo di **Bounding Box** (bbox), che circoscrive l'espansione dei nodi all'interno di una sottofinestra specifica dell'area operativa, riducendo drasticamente il carico computazionale rispetto a una ricerca sull'intera mappa globale.



2.5 Sistema di navigazione

Il modulo Naviga costituisce il nucleo architetturale deputato alla gestione della mobilità autonoma, integrando la percezione ambientale, la pianificazione globale e il controllo di traiettoria in un unico loop di controllo ad alta frequenza (10 Hz). La gestione della mappa, ricevuta sul topic /map, implementa una strategia di sicurezza preventiva denominata **Map Inflation**: per garantire che il robot mantenga una distanza di sicurezza dagli ostacoli fisici, i dati della griglia di occupazione vengono elaborati tramite un algoritmo ottimizzato che "gonfia" gli ostacoli di un raggio di sicurezza (SAFETY_RADIUS). Per

massimizzare le performance in Python ed evitare colli di bottiglia computazionali, l'algoritmo calcola anticipatamente gli offset geometrici del "pennello" di espansione una sola volta e applica le modifiche esclusivamente agli indici delle celle occupate, evitando iterazioni ridondanti sull'intera matrice.

La pianificazione del percorso è demandata al metodo pianifica, che integra l'algoritmo A* precedentemente descritto. Una volta ottenuto il percorso grezzo, viene applicata una routine di semplificazione (simplify_path) che rimuove i waypoint intermedi ridondanti lungo i segmenti rettilinei, generando una traiettoria vettoriale più pulita composta solo dai punti di svolta essenziali.

Il cuore operativo è il metodo control_loop, che implementa una macchina a stati per l'inseguimento dei waypoint. Il sistema aggiorna costantemente la propria posa interrogando il buffer delle trasformate TF2 (map → base_link) e calcola l'errore angolare rispetto al prossimo target.

Il sistema, inoltre, verifica continuamente la presenza di ostacoli frontali che potrebbero ostruire il percorso inizialmente pianificato. In tal caso, viene richiamato il metodo pianifica, escludendo il corridoio bloccato, per calcolare un percorso alternativo (se disponibile).

La logica di controllo adotta un approccio ibrido per gestire la cinematica differenziale:

- **Stato di rotazione (Turn-in-place):** Se l'errore angolare supera una soglia critica (TOLLERANZA_START_ROT), il robot si ferma e ruota sul posto utilizzando un controllore PI (Proporzionale-Integrale) per annullare l'errore di orientamento.
- **Stato di avanzamento (Drive-to-goal):** Quando l'allineamento rientra nella tolleranza, il robot avanza applicando una velocità lineare proporzionale alla distanza, correggendo simultaneamente la rotta con un termine angolare dinamico per mantenere la traiettoria fluida.

Il passaggio tra i waypoint avviene in continuità quando il robot entra nel raggio di tolleranza (dist_threshold) del punto corrente, garantendo un movimento armonico senza stop superflui, a meno che la geometria del percorso non imponga una curva a gomito che riattiva lo stato di rotazione sul posto.

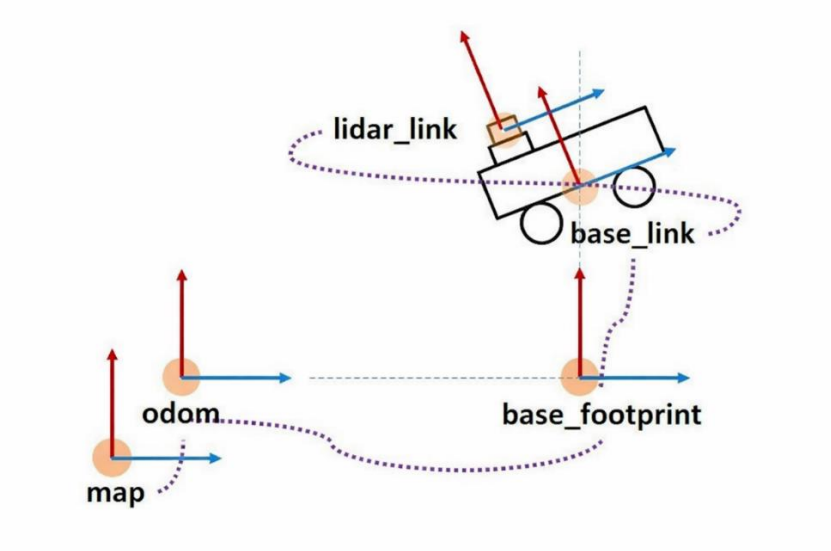
2.6 TF

Per garantire la coerenza spaziale tra i dati sensoriali e la navigazione, il sistema sfrutta il modulo **TF2** per mantenere e aggiornare in tempo reale l'albero delle trasformazioni coordinate (TF Tree). All'interno del nostro progetto, i sistemi di riferimento non sono entità isolate, ma sono legati gerarchicamente per permettere al robot di comprendere la propria posizione rispetto all'ambiente globale.

Possiamo distinguere due tipologie di relazioni spaziali:

- **Trasformate statiche (Intra-Robot):** Sono le relazioni immutabili, definite dalla geometria fisica del robot. Un esempio è la trasformata tra il centro del robot (base_link o base_footprint) e il sensore LiDAR (lidar_link). Poiché il Lidar è imbullonato allo chassis, la sua posizione relativa non cambia mai; questa trasformazione viene pubblicata una sola volta dallo StaticTransformBroadcaster per dire al sistema che i dati del LiDAR provengono da un punto traslato e ruotato rispetto al centro di controllo.
- **Trasformate dinamiche (World-to-Robot):** Qui risiede la complessità maggiore. Il controller differenziale pubblica costantemente la trasformata tra **odom** (il punto di partenza locale) e **base_link** (il robot), basandosi sulle rotazioni delle ruote; questa stima è fluida ma soggetta a drift nel tempo. Per correggere questo errore senza causare "salti" discontinui nella navigazione, il sistema di localizzazione non sposta il robot, ma pubblica una trasformata correttiva tra **map** (il mondo fisso) e **odom**. In pratica, l'origine del sistema odometrico viene "fatta scivolare"

rispetto alla mappa globale per compensare l'errore accumulato, garantendo che le coordinate del robot risultino corrette rispetto all'ambiente circostante.



2.7 Localizzazione

La localizzazione del robot all'interno della mappa globale è gestita dal nodo **AMCL** (Adaptive Monte Carlo Localization), che implementa un filtro particellare probabilistico per stimare la posa (x, y, θ) in un ambiente noto. Il sistema è stato configurato per operare in sincronia con il tempo simulato (use_sim_time: true), fondamentale per l'integrazione con Unity che detta il clock, e si aggancia all'albero delle trasformate standard ROS collegando il frame del robot base_footprint e quello odometrico al sistema di riferimento globale map. Il cuore dell'algoritmo risiede nella gestione della nuvola di particelle, configurata per adattarsi dinamicamente all'incertezza del sistema: il numero di campioni varia tra un minimo di min_particles: 500 (quando la posizione è certa) e un massimo di max_particles: 2000 (durante fasi di incertezza o inizializzazione globale). Per ottimizzare il carico computazionale della CPU, il filtro non viene aggiornato a ogni ciclo di clock, ma segue una logica basata sullo spostamento delta: i parametri update_min_d: 0.2 e update_min_a: 0.2 impongono che il ricampionamento avvenga solo dopo una traslazione di 20 cm o una rotazione di circa 0.2 rad.

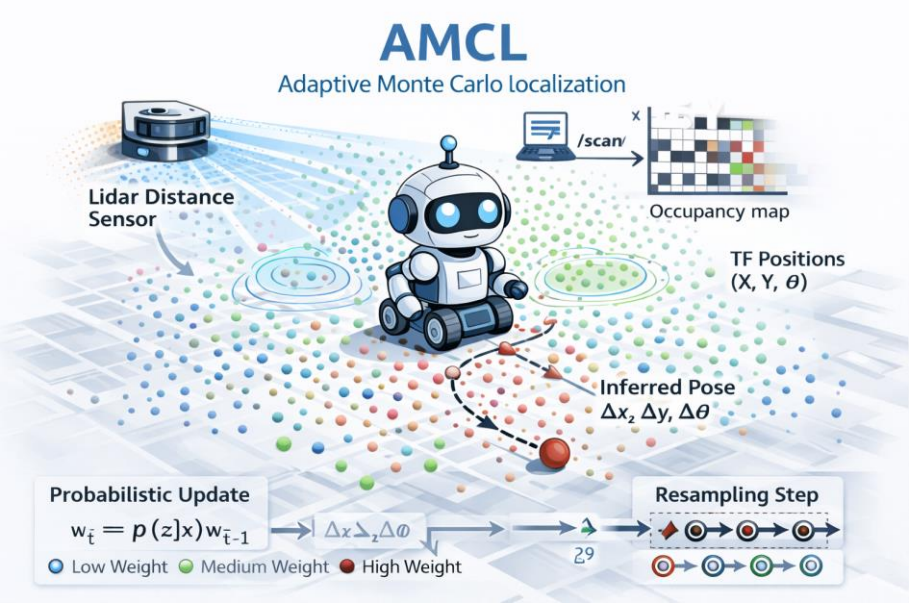
Il modello di movimento (fase di predizione) è definito dal parametro odom_model_type: "diff", specifico per robot a guida differenziale. L'errore intrinseco dell'odometria è modellato attraverso i quattro parametri di rumore alpha1...alpha4, che definiscono la varianza dell'errore rotazionale e traslazionale generato dal movimento. Nel nostro setup, questi valori sono stati impostati molto bassi (0.01 e 0.02), indicando che il sistema ripone un'elevata fiducia nella precisione degli encoder simulati e introduce solo una minima dispersione gaussiana per prevenire la degenerazione del filtro.

Per quanto riguarda il modello sensoriale (fase di correzione), il nodo sottoscrive il topic scan. In questa configurazione, abbiamo impostato il parametro max_beams: 360, optando per un campionamento distribuito uniformemente che offre un bilanciamento ottimale: questo valore garantisce una densità di informazioni sufficiente per una localizzazione precisa anche in ambienti complessi, filtrando al contempo i dati ridondanti per mantenere il carico computazionale sostenibile. La probabilità di misura è calcolata tramite un modello a mistura pesato dai parametri z:

- **z_hit: 0.7**: È il peso predominante, assegnato alla probabilità che il LiDAR colpisca un ostacolo reale presente nella mappa (modellato con una gaussiana di deviazione standard sigma_hit: 0.3).

- **z_short: 0.1**: Gestisce gli ostacoli dinamici non mappati (es. NPC che passano davanti al robot), che accorciano la lettura prevista.
- **z_max: 0.1** e **z_rand: 0.1**: Gestiscono rispettivamente i fallimenti del sensore e il rumore casuale di fondo.

Infine, il risultato della stima viene reso operativo abilitando il tf_broadcast: true, che pubblica la trasformata correttiva map-odom con una transform_tolerance: 3.0 secondi, garantendo robustezza anche in presenza di leggere latenze di rete tra il simulatore e lo stack di navigazione.



2.8 Elaborazione dei segnali biometrici tramite filtro di Kalman

Per garantire la robustezza e l'affidabilità del monitoraggio dei parametri vitali (battito cardiaco e pressione sanguigna), non ci siamo limitati a inoltrare i dati grezzi provenienti dai braccialetti, intrinsecamente soggetti a rumore di misura e artefatti di movimento. Abbiamo invece ingegnerizzato un modulo ROS2 dedicato, BraccialettiManager, che implementa manualmente un **filtro di Kalman lineare** utilizzando esclusivamente la libreria NumPy per massimizzare le performance e ridurre le dipendenze esterne.

Nello specifico, è stata adottata una strategia a filtri disaccoppiati, istanziando tre filtri indipendenti (rispettivamente per battiti, pressione minima e massima). Questa separazione architetturale è cruciale per garantire la Fault Isolation: essa impedisce che un picco di rumore o un artefatto su un singolo sensore contamini matematicamente la stima degli altri parametri attraverso la matrice di covarianza, preservando la coerenza del quadro clinico anche in presenza di disturbi parziali.

L'architettura del filtro è stata modellata su uno spazio di stato bidimensionale composto dalla coppia **[Valore, Trend]**. Questa scelta progettuale, basata su un modello fisico a "velocità costante", permette al sistema non solo di stimare il valore attuale del parametro (es. i battiti al minuto), ma anche la sua derivata temporale (la tendenza di crescita o decrescita). La matrice di transizione di stato (F) proietta lo stato nel futuro basandosi sul tempo trascorso (dt) tra due pacchetti successivi, mentre la componente di trend viene limitata fisicamente per evitare che il filtro reagisca eccessivamente a picchi momentanei non fisiologici. Questo approccio matematico consente di "prevedere" il valore atteso prima ancora di analizzare la misura reale.

Il cuore della logica di sicurezza risiede nella gestione delle anomalie, implementata nel metodo process_signal. Anziché accettare ciecamente ogni dato in ingresso, il sistema confronta la misura

grezza ricevuta con la predizione matematica del filtro. Se la discrepanza (errore residuo) supera una soglia di tolleranza dinamica predefinita (adattata in base al ritardo temporale del pacchetto), il dato viene classificato come **outlier** e scartato temporaneamente, privilegiando la stima del modello interno. Tuttavia, per distinguere un artefatto momentaneo (es. un movimento brusco del braccio) da un reale cambiamento fisiologico repentino (es. un inizio di tachicardia), è stato introdotto un meccanismo di persistenza: solo se l'anomalia persiste per un numero consecutivo di campioni superiore a MAX_OUTLIERS, il filtro interpreta il dato come un effettivo cambio di stato e si "resetta" sul nuovo valore, garantendo così una reattività adeguata alle vere emergenze. Infine, il sistema gestisce autonomamente la disconnessione o la rimozione del dispositivo: se non giungono dati per un intervallo superiore al RESET_TIMEOUT, lo stato interno dell'ospite viene invalidato, prevenendo l'elaborazione di dati obsoleti.

2.9 Infrastruttura di comunicazione: pattern RPC su topic

La gestione della comunicazione tra i nodi del sistema (in particolare tra la logica di controllo, l'ontologia il graph database, il Reasoner e il modulo LLM) ha richiesto l'implementazione di un meccanismo di **chiamata sincrona bloccante**. Sebbene ROS2 offra nativamente i Services per questo scopo, abbiamo optato per un'architettura custom basata su **topic asincroni** (Pattern Request-Response over Topics), implementata nelle classi SincronizzaManager e SincronizzamiTutto.

Questa scelta architetturale, apparentemente controintuitiva, offre vantaggi strategici in termini di disaccoppiamento e scalabilità futura, in particolare per il **caching**. A differenza dei service (che sono punto-punto), i topic operano in modalità publish-subscribe: ciò significa che le risposte del server (es. un output complesso dell'LLM o una query pesante al database) sono visibili sul bus di comunicazione globale. Questo permetterebbe, in sviluppi futuri, di inserire nodi di caching che memorizzano le risposte frequenti senza dover modificare né il client né il server, riducendo la latenza e i costi computazionali. Questo vantaggio risulterebbe particolarmente evidente anche e soprattutto qualora si decidesse di implementare un sistema multi-agente.

A livello implementativo, la classe generica SincronizzamiTutto astrae la complessità di trasformare uno scambio di messaggi asincrono in una chiamata procedurale bloccante. Il funzionamento si basa sull'uso di **correlation IDs**:

1. Ogni richiesta viene marcata con un **UUID** (Universally Unique IDentifier) univoco e inviata sul topic di request.
2. Immediatamente viene istanziato un oggetto concurrent.futures.Future, che blocca il thread chiamante in attesa del risultato (con gestione del timeout).
3. Quando arriva una risposta sul topic di response, la callback on_response verifica l'ID del pacchetto JSON. Se l'ID corrisponde a una richiesta pendente, il risultato viene iniettato nel Future corrispondente, sbloccando istantaneamente il thread che era in attesa.

La classe SincronizzaManager invece incapsula le istanze di trasporto e fornisce metodi semantici di alto livello. In questo modo, la logica di business può invocare operazioni complesse come se fossero semplici funzioni locali, ignorando totalmente la complessità della comunicazione distribuita sottostante e la gestione dei messaggi JSON.

Capitolo 3: Intelligenza Artificiale

Questo capitolo illustra l'architettura cognitiva del sistema, focalizzandosi sull'implementazione di un approccio ibrido che integra l'intelligenza artificiale simbolica con i moderni modelli generativi. Verranno descritte le metodologie adottate per la **rappresentazione della conoscenza** tramite ontologie formali e graph database, e le strategie di **ragionamento automatico** (Reasoning) utilizzate per dedurre nuove informazioni e validare le decisioni del robot.

L'obiettivo è mostrare come il sistema garantisca coerenza logica, sicurezza operativa e trasparenza (**Explainability**) in un ambiente dinamico caratterizzato dall'Assunzione del Mondo Aperto (OWA), superando i limiti di determinismo tipici dei soli modelli statistici.

3.1 Large Language Model

Nell'ambito del sottosistema di interazione uomo-robot, l'adozione dei Large Language Models (LLM) è stata strategica ma rigorosamente confinata al livello di presentazione, escludendo tassativamente qualsiasi coinvolgimento nei processi decisionali o nella pianificazione delle azioni. Nello specifico, è stata integrata l'API di Google Gemini sfruttando il modello gemini-2.5-flash.

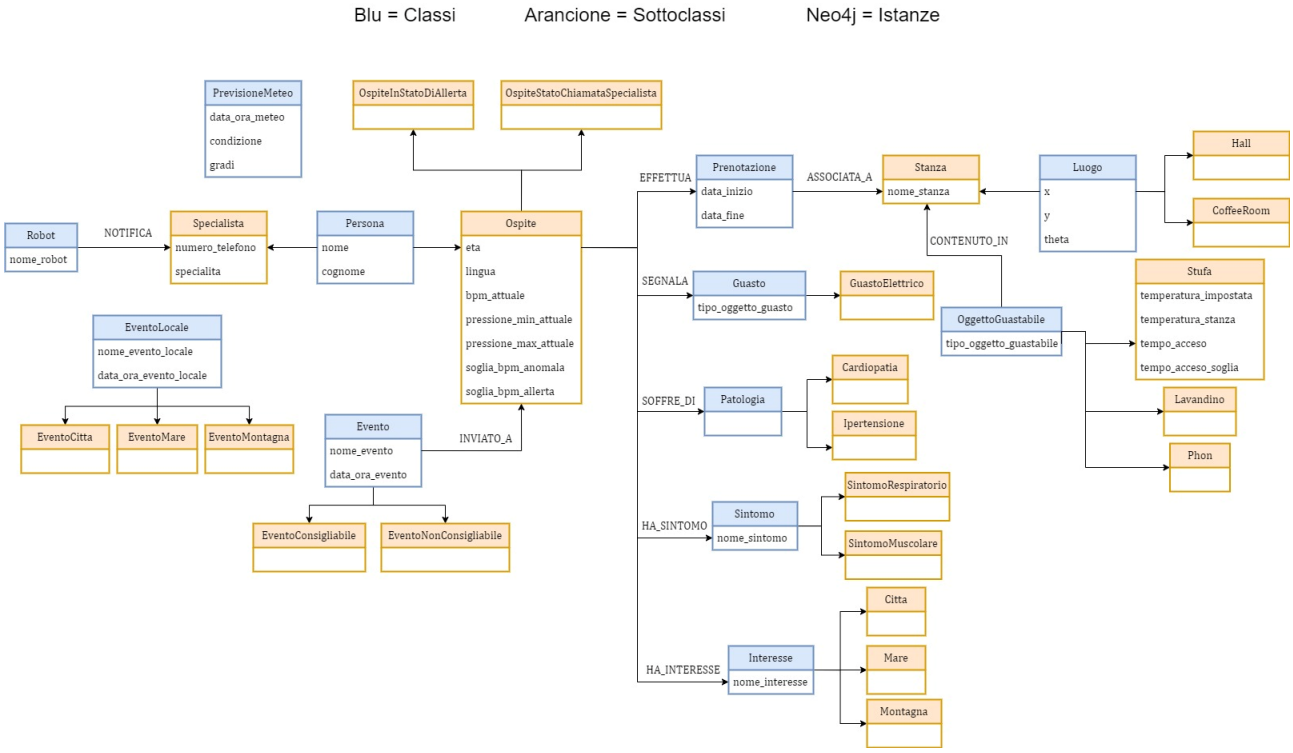
Il ruolo del modello è limitato alla Natural Language Generation (NLG): esso non decide il contenuto informativo, che rimane vincolato alla logica deterministica e sicura della base di conoscenza, bensì si occupa esclusivamente della forma, trasformando gli output strutturati e i dati grezzi del sistema in risposte discorsive, empatiche e grammaticalmente curate, migliorando drasticamente la user experience riducendo rischi di "allucinazioni" operative.

3.2 Base di conoscenza e ontologia

L'architettura della base di conoscenza è stata ingegnerizzata disaccoppiando la fase di definizione strutturale (TBox) da quella di popolamento e persistenza delle istanze (ABox), al fine di ottimizzare sia il rigore logico che l'efficienza operativa. Per la modellazione dello schema ontologico è stato adottato **Protégé**, strumento che ha permesso di definire con precisione formale le gerarchie di classi, le proprietà e le restrizioni assiomatiche, garantendo la coerenza semantica e la validazione del modello prima del suo utilizzo.

Parallelamente, per la gestione dinamica dei dati e l'iniezione delle istanze nel sistema, si è optato per l'utilizzo di **Neo4j**; questa scelta strategica consente di traslare la struttura logica dell'ontologia in un database a grafo nativo ad alte prestazioni, capace di gestire volumi elevati di nodi e relazioni con tempi di latenza ridotti.

Tale approccio ibrido coniuga l'espressività inferenziale tipica del web semantico, garantita dalla progettazione in Protégé, con la scalabilità e la velocità di accesso ai dati offerte da Neo4j, requisiti indispensabili per supportare i processi decisionali del robot in tempo reale.



Si precisa che le classi blu sono tutte figlie di Thing: i collegamenti sono stati omessi per una maggiore leggibilità del grafo.

3.2.1 Rappresentazione ontologia

Classe	Sottoclasse	Attributi	Relazioni
Persona	Ospite: OspiteInStatoDiAllerta OspiteStatoChiamataSpecialista	nome cognome eta lingua bpm_attuale pressione_min_attuale pressione_max_attuale soglia_bpm_anomala soglia_bpm_allerta	(Ospite)-[:EFFETTUA]->(Prenotazione) (Ospite)-[:SEGNALA]->(Guasto) (Ospite)-[:SOFFRE_DI]->(Patologia) (Ospite)-[:HA_SINTOMO]->(Sintomo) (Ospite)-[:HA_INTERESSE]->(Interesse)
	Specialista	nome cognome numero_telefono specialita	
Robot		nome_robot	(Robot)-[:NOTIFICA]->(Specialista)
EventoLocale	EventoCitta	nome_evento_locale data_ora_evento_locale	
	EventoMare		
	EventoMontagna		
Evento	EventoConsigliabile	nome_evento data_ora_evento	(Evento)-[:INVIATO_A]->(Ospite)
	EventoNonConsigliabile		
PrevisioneMeteo		data_ora_meteo condizione gradi	
Prenotazione		data_inizio data_fine	(Prenotazione)-[:ASSOCIATA_A]->(Stanza)
Luogo	Stanza	x y theta nome_stanza	
	Hall	x y theta	
	CoffeeRoom		
Guasto	GuastoElettrico	tipo_oggetto_guasto	

OggettoGuastabile	Stufa	tipo_oggetto_guastabile temperatura_impostata temperatura_stanza tempo_acceso tempo_acceso_soglia	(OggettoGuastabile)-[:CONTENUTO_IN]->(Stanza)
	Lavandino	tipo_oggetto_guastabile	
	Phon		
Patologia	Cardiopatia		
	Ipertensione		
Sintomo	SintomoMuscolare	nome_sintomo	
	SintomoRespiratorio		
Interesse	Citta	nome_interesse	
	Mare		
	Montagna		

3.3 Explainability

La strategia di **Explainability** del sistema è stata concepita per risolvere le ambiguità intrinseche della **Open-World Assumption (OWA)** tipica dei reasoner semantici, dove l'assenza di un'informazione non implica la sua falsità ma solo uno stato di non-conoscenza. Per garantire decisioni sicure e trasparenti, abbiamo adottato un approccio di modellazione esplicita dei vincoli mediante regole SWRL: le condizioni di validità (permessi) e di rischio (divieti) sono modellate nell'ontologia come classi positive distinte (es. EventoConsigliabile vs EventoNonConsigliabile). In questo modo, il motore inferenziale non deve "indovinare" la negazione, ma deduce attivamente la presenza di vincoli ostativi.

L'implementazione di questo paradigma avviene attraverso una pipeline a due stadi orchestrata dal nodo ROS2 SpiegamiTutto. La prima fase, gestita dal metodo logic_crea_ontologia_istanze, si occupa dell'istanziazione contestuale on-the-fly: il sistema estrae dal graph database (Neo4j) solo il sottografo rilevante per l'interazione corrente (es. l'ospite, le sue patologie, gli eventi candidati) e lo transcodifica dinamicamente in un file OWL temporaneo usando la libreria rdflib. Questo passaggio trasforma i nodi e le relazioni del database in NamedIndividual e ObjectProperty semantici, colmando il divario tra la persistenza dei dati e la logica formale.

La seconda fase è il cuore dell'Explainability e viene delegata a un modulo esterno sviluppato in Java, invocato tramite subprocesso per sfruttare la robustezza delle librerie **OWLAPI** e **Openllet**. La scelta di Java è dettata dalla necessità di utilizzare la classe DefaultExplanationGenerator, uno strumento avanzato non disponibile in ambiente Python, capace di tracciare la catena causale delle inferenze (assiomi). Il codice Java non si limita a verificare se un assioma è vero (isEntailed), ma esegue un'iterazione mirata sulle classi target. Per ogni istanza, il reasoner tenta di classificarla sia nelle categorie positive che in quelle negative; qualora l'inferenza abbia successo, il generatore di spiegazioni estrae l'insieme esatto di assiomi e regole SWRL che hanno portato a quella conclusione. Il risultato è un output JSON strutturato che mappa ogni entità non solo al suo stato logico, ma alla giustificazione formale di tale stato. Questo dizionario viene infine restituito al controller Python, che applica la policy di sicurezza: inibisce l'azione se esistono spiegazioni nella categoria dei divieti e propone l'azione se esistono evidenze nella categoria dei permessi, garantendo così che ogni comportamento del robot sia non solo corretto, ma pienamente intellegibile dall'utente.

3.4 Implementazione scenari

3.4.1 Scenario A

Descrizione:

In questo primo scenario abbiamo modellato la fase di accoglienza iniziale e la profilazione dinamica dell'utente. Il sistema gestisce il riconoscimento dell'ospite e il conseguente adattamento linguistico, avviando un dialogo mirato all'estrazione di interessi e preferenze integrabili con i dati forniti in fase di prenotazione online. Tali informazioni vengono iniettate in tempo reale nella base di conoscenza e immediatamente elaborate dal reasoner per filtrare le attività disponibili per l'ospite; il robot è quindi in grado di formulare suggerimenti personalizzati, verificandone la fattibilità rispetto a vincoli ambientali e temporali, e gestendo eccezioni complesse.

Implementazione:

La gestione dello scenario di accoglienza è stata incapsulata all'interno della classe InteragisciScenarioA, la quale implementa una macchina a stati finiti per gestire la sequenzialità dell'interazione. La logica di controllo non si limita a seguire un copione predefinito, ma coordina dinamicamente le chiamate ai servizi cognitivi, al database a grafo e al motore inferenziale.

Per quanto concerne il modulo preliminare di identificazione della lingua, implementato nel metodo rileva_lingua, abbiamo scartato l'adozione di librerie standard basate su modelli statistici n-grams (come langdetect). In casi d'uso con input brevi, come una frase di presentazione "My name is Peppe Rossi", i classificatori statistici tendono a etichettare erroneamente l'input come "italiano" a causa del peso preponderante delle feature testuali del nome proprio rispetto alle parole funzionali inglesi. Per garantire un'esperienza utente deterministica, abbiamo optato per un approccio basato su **Keyword Spotting con dizionari controllati**: il sistema analizza l'input confrontando i token con vocabolari precaricati (italiano e inglese) e assegna la lingua basandosi su un punteggio di frequenza, garantendo precisione assoluta in questo specifico contesto.

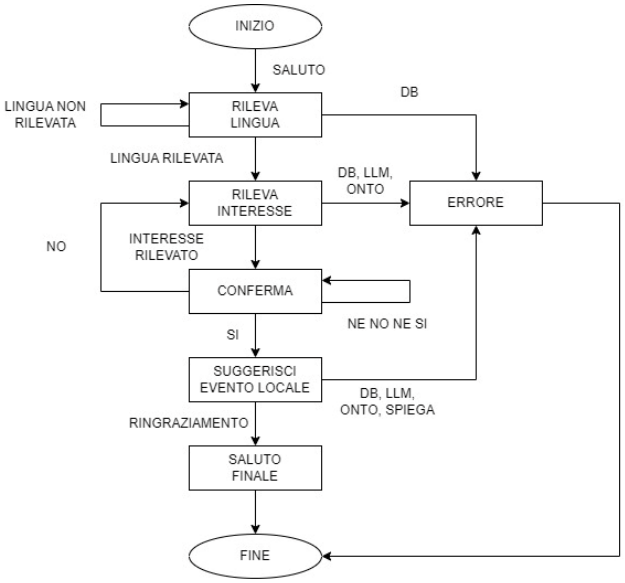
Una volta stabilita la lingua e recuperati i dati di prenotazione tramite query Cypher al database Neo4j, il sistema avvia la fase di profilazione. Qui adottiamo un approccio ibrido: il metodo rileva_interesse utilizza il Large Language Model per l'estrazione semantica dell'interesse dal linguaggio naturale (es. la frase "mi piace fare escursioni" viene ricondotta alla parola "trekking"). Successivamente, il sistema valida questo output mappandolo rigorosamente a una classe ontologica esistente (Montagna) tramite il metodo trova_classe_da_sinonimo, che sfrutta le alt label dello schema ontologico per ricondurre il termine allo specifico concetto formale. Questo passaggio è fondamentale per evitare di inserire nel Knowledge Graph dati non strutturati o ambigui.

Il cuore decisionale dello scenario risiede nel metodo suggerisci_evento_locale, che concretizza la strategia di Explainability e gestione OWA. Il sistema costruisce prima un sottografo di contesto recuperando da Neo4j l'ospite, le sue patologie, i nuovi interessi acquisiti e gli eventi compatibili per data e meteo (recupera_dati_per_suggerimento). A questo punto interviene il nodo ROS2 SpiegamiTutto, che funge da ponte architetturale tra gestione dati e logica formale: per evitare un reasoning computazionalmente insostenibile sull'intero database, i dati contestuali vengono serializzati dinamicamente in un file OWL temporaneo (trasformando i nodi Neo4j in individui ontologici tramite rdflib) e processati da un modulo Java esterno, invocato appositamente per sfruttare librerie inferenziali robuste non disponibili in ambiente Python.

Analizzando il JSON restituito da questo processo, il codice applica la policy di sicurezza globale: verifica la presenza di assiomi nella chiave EventoNonConsigliabile per applicare eventuali veti (es.

conflitti medici o ambientali) e, solo in loro assenza, valuta la chiave EventoConsigliabile per formulare la proposta. La decisione operativa avviene quindi a valle dell'inferenza logica evitando la logica dello "short-circuit" tipica dei costrutti condizionali tradizionali (if/else), che interromperebbero la valutazione alla prima condizione soddisfatta. In questo modo, il robot è in grado di verbalizzare all'utente non solo il suggerimento, ma anche le motivazioni specifiche sia in caso di azione intrapresa sia in caso di rifiuto, garantendo trasparenza ed explainability totale prima di chiudere l'interazione.

Schema:



3.4.2 Scenario B

Descrizione:

Questo scenario è dedicato alla gestione delle richieste di assistenza tecnica e manutenzione all'interno della struttura. Il flusso operativo viene innescato da una segnalazione dell'ospite che richiede l'intervento del robot in camera per un malfunzionamento di un oggetto. Una volta arrivato in camera, il robot chiede qual è l'oggetto malfunzionante; dopo aver ricevuto la risposta la elabora e pone all'ospite delle domande specifiche riguardanti l'oggetto per capire se è necessario l'intervento immediato dello specialista. Se il riscontro è positivo allora il robot lo rende noto all'ospite e notifica lo specialista, in caso contrario pone comunque la possibilità all'ospite di chiamarlo qualora lo desideri.

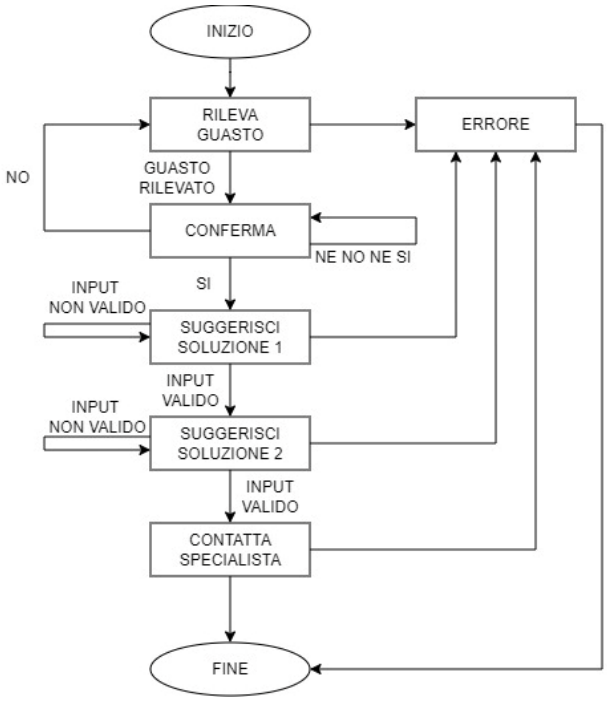
Implementazione:

La logica operativa dello Scenario B è incapsulata nella classe InteragisciScenarioB, strutturata come una FSM sequenziale progettata per guidare l'ospite dall'identificazione del problema fino alla risoluzione tecnica. Il flusso di esecuzione inizia con la fase di navigazione, in cui il sistema imposta le coordinate target della stanza dell'ospite; una volta raggiunto l'obiettivo, il robot avvia l'interazione richiedendo di specificare l'oggetto del malfunzionamento (RILEVA_GUASTO). La risposta dell'utente in linguaggio naturale viene processata tramite un modulo LLM che esegue un'estrazione semantica: il risultato viene mappato univocamente a una classe ontologica nota, salvando il riferimento nel contesto operativo.

Successivamente, il sistema entra nella fase di diagnostica parametrica, ponendo domande specifiche per acquisire lo stato attuale del dispositivo. Questi dati vengono utilizzati per aggiornare in tempo reale le istanze su Neo4j tramite query Cypher di scrittura.

A questo punto si innesca la pipeline di ragionamento: il sistema estrae il sottografo aggiornato, genera un'ontologia temporanea e invoca il Reasoner. La decisione finale adotta un approccio ibrido: se il motore inferenziale deduce formalmente un guasto (es. violazione di vincoli logici tra temperatura e tempo nel caso della stufa), il robot lo conferma all'ospite ed esegue automaticamente la chiamata allo specialista fornendogli la causa tecnica. Viceversa, qualora il reasoner non disponga di informazioni sufficienti per classificare l'anomalia (OWA), il sistema informa l'utente dell'assenza di riscontri logici che possono classificare con certezza l'oggetto come non guasto, ma offre comunque la possibilità di inviare la segnalazione al tecnico, garantendo sempre il benessere dell'ospite.

Schema:



3.4.3 Scenario C

Descrizione:

Questo scenario rappresenta il livello più alto della gerarchia dei comportamenti ed è dedicato alla gestione delle anomalie di salute degli ospiti registrate al momento. La peculiarità architetturale di questo modulo risiede nella sua doppia modalità di attivazione, progettata per coprire sia le richieste esplicite dell'ospite sia le anomalie rilevate tramite i braccialetti dal robot. In modalità **reattiva**, il robot si attiva su comando diretto dell'ospite pigiando il pulsante di emergenza sul braccialetto smart; in modalità **proattiva**, invece, il sistema agisce autonomamente qualora il modulo di filtraggio biometrico (filtro di Kalman) rilevi un'anomalia persistente o un trend preoccupante nei parametri vitali, notificando l'emergenza allo specialista.

Implementazione:

L'architettura software dello Scenario C è incapsulata nella classe InteragisciScenarioC, strutturata come una FSM ad alta priorità progettata per gestire tempestivamente le richieste. La logica di controllo implementa un meccanismo di innesco duale che distingue tra attivazione **proattiva** e **reattiva**. In entrambi i casi al raggiungimento della posizione target dell'ospite, il robot transita nello stato COSCIENZA avviando un timer di sicurezza; qualora l'ospite non risponda entro una soglia temporale di 30 secondi o fornisca riscontro negativo, il sistema salta l'analisi sintomatologica e passa immediatamente allo stato CHIAMATA_SPECIALISTA, in cui viene avvisato lo specialista.

Nel caso di ospite cosciente o di attivazione manuale, il flusso prosegue nello stato DESCRIZIONE_SINTOMI, dove l'input viene processato da una pipeline di estrazione semantica: un modulo LLM estrae i sintomi dichiarati (es. "tosse") mappandoli a classi ontologiche, le quali vengono immediatamente salvate su Neo4j tramite query Cypher. Questo aggiornamento in tempo reale abilita la fase successiva di reasoning: il metodo suggerisci_medico estrae il sottografo aggiornato (comprendente anagrafica, patologie pregresse e sintomi) e invoca il motore inferenziale per verificare la sussistenza di condizioni di rischio.

Il processo decisionale si conclude con una logica ibrida: se il reasoner deduce formalmente uno stato di allerta (OspiteStatoChiamataSpecialista), il robot contatta autonomamente il medico; in caso contrario, ovvero se l'inferenza non rileva gravità immediata, il sistema adotta comunque un approccio cautelativo nello stato MEDICO_SI-NO, permettendo all'ospite di forzare la chiamata al dottore e garantendo sempre il benessere dell'ospite.

Schema: