https://micro.ros.org/[Introduction 2](#_Toc1)

[Background and motivation 2](#_Toc2)

[Research questions and objectives 2](#_Toc3)

[Scope and limitations 2](#_Toc4)

[Literature Review 3](#_Toc5)

[Overview of ROS1 and ROS2 3](#_Toc6)

[Descrizione delle parti di ROS2 3](#_Toc7)

[La strumentazione offerta in ROS 4](#_Toc8)

[Da ROS 1 a ROS 2 4](#_Toc9)

[Introduction to Navigation Stack and Google Cartographer 6](#_Toc10)

[Review of related work on porting navigation stack from ROS1 to ROS2 6](#_Toc11)

[Overview of open source fleet management systems, particularly Open-RMF 6](#_Toc12)

[Methodology 6](#_Toc13)

[Description of research platform and hardware used 6](#_Toc14)

[Steps taken to port the navigation stack from ROS1 to ROS2 6](#_Toc15)

[Hardware adaptation process from Rover Zero to Rover Mini 6](#_Toc16)

[Integration of the navigation stack with Open-RMF 6](#_Toc17)

[Testing and evaluation of the system 6](#_Toc18)

[Results and Analysis 6](#_Toc19)

[Presentation of results from testing and evaluation 6](#_Toc20)

[Comparison of system performance before and after the porting process 6](#_Toc21)

[Discussion on the challenges encountered and how they were addressed 6](#_Toc22)

[Conclusion and Future Work 6](#_Toc23)

[Summary of the research findings 6](#_Toc24)

[Contributions to the field of electrical engineering and robotics 6](#_Toc25)

[Recommendations for future work and improvements 6](#_Toc26)

[References [[references]] 7](#_Toc27)

[List of sources cited in the thesis 7](#_Toc28)

[Appendices 7](#_Toc29)

[Additional technical details and documentation 7](#_Toc30)

# Introduction

## Background and motivation

Elettra è una ente di ricerca internazionale che fornisce due forme di sorgenti di radiazioni (light source): il sincrotrone Elettra ed il laser a elettroni liberi fermi.

https://www.elettra.eu/

Essendo un grande ente di ricerca, si sviluppano applicazioni che posso rendere migliore lo svolgimento delle attività all’interno dello stesso. Uno di questi miglioramenti è l’utilizzo della robotica nelle attività ordinarie, ad esempio trasporto di oggetti, interazioni con gli utenti che utilizzano le macchine, ... Tale moltitudine di applicazioni ha reso l’ambiente ricco di una varietà di robot da diversi produttori ed un problema che si presenta con i diversi produttori è l’interazione in maniera intelligente tra di essi. In questi ultimi anni, dalla open robotics hanno sviluppato Open-rmf[[1]](#footnote-2) un gestore di robot multipiattaforma “multi fleet”.

Inoltre è stato sviluppato, attraverso il Robotics and Remotization initiative, un planner di navigazione D-Star-Based Optimized Trajectory Planner[[2]](#footnote-3) per ROS1 ottimizzato per la navigazione in ambienti industriali grandi e complessi come quello di elettra che si vuole appilcare ad un robot il mini della roverrobotics[[3]](#footnote-4)

## Research questions and objectives

## Scope and limitations

Lo scopo di questa tesi è aggiornare il sistema di planning sviluppato portandolo da ROS1 a ROS2[[4]](#footnote-5) utilizzando la struttura rinnovata di NAV2 rispetto a NAV1, applicarlo al nostro robot MINI della rover robotics ed inserirlo dentro l’ambiente OPEN-RMF che accoglierà in futuro tutte le altre famiglie di robot presenti in azienda.

Le uniche limitazioni che al momento non abbiamo un controllo diretto agli ascensori e non permettono di testare tutte le funzionalità offerte da OPEN\_RMF.

# Literature Review

Adesso descriveremo l’insieme delle tecnologie usate

## Overview of ROS1 and ROS2

Il Robot Operating System[[5]](#footnote-6) (ROS) è un insieme di librerie e strumenti open-source che aiutano a sviluppare applicazione in ambito robotico. Diversamente dal nome non è un sistema operativo ma un software development kit (SDK) che ti fornisce tutti gli strumenti per creare il tuo robot fornendo alcune funzionalità tipiche di un OS: hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management.

I principali vantaggi che offre ROS sono:

* un sistema di comunicazione message-passing system chiamato “middleware” or “plumbing”. Che permette la comunicazione tra i “nodi” del sistema con un anonymous publish/subscribe pattern permettendo fault isolation, separation of concerns and clear interfaces e un sistema synchronous RPC-style communication tra i services
* Sistemi di controllo del software come launch, introspection, debugging, visualization, plotting, logging, and playback. Che migliorano gli strumenti per lo sviluppo.
* provides tools and libraries for obtaining, building, writing, and running code across multiple computers
* Una communità attiva e diffusa con al centro la Open Robotics[[6]](#footnote-7) che mantiene il codice e hosta le release

### Descrizione delle parti di ROS2

La documentazione[[7]](#footnote-8) fornisce una descrizione esaustiva di tutti gli strumenti base offerti da ROS2, alcune funzionalità qui descritte sono solo disponibili in ROS2 e non in ROS1:

* The **ROS graph** è una peer-to-peer network of processes di ROS2 che elaborano dati simultaneamente. Comprende tutti gli eseguibili e le connessioni tra di essi.
* **ROS node** è il processo che dovrebbe realizzare un singolo obbiettivo e dovrebbe essere realizzato in maniera modulare in modo da essere facilmente interfacciabile da alti nodi che descrivono altri specifici funzionalità che può essere concatenato all’altro nodo. Ogni nodo comunica con gli altri nodi con “topics, services, actions, or parameters”.
* **Topics** sono un elemento fondamentale del grafo ROS che funge da bus per consentire ai nodi di scambiarsi messaggi.
* **Services** si basano su un modello di richiesta e risposta, a differenza del modello publisher-subscriber dei topic. Mentre i topic consentono ai nodi di iscriversi a flussi di dati e ricevere aggiornamenti continui, i Services forniscono dati solo quando vengono specificamente richiesti da un client.
* A **parameter** sono i parametri di configurazione dei nodi che permettono di modificarli senza cambiare il codice sorgente
* **Actions** sono stati sviluppati per task lunghi e consistono di 3 parti: a goal,a feedback, and a result. Sono costruiti al di sopra dei topic e dei servizi, la funzionalità è simile hai services ma può esser fermato e fornisce un update continuo sullo stato dell’azione.

### La strumentazione offerta in ROS

Qui di seguito faccio un elenco degli strumenti offerti da ROS per lo sviluppo di ogni applicazione e con una breve descrizione a riguardo:

* **Launch** files struttura che ti permettono di configurare ed eseguire multipli nodi ROS simultaneamente.
* **rqt** is a graphical user interface (GUI) tool for ROS 2, fornisce dei plugin che possono essere aggiunti all’interfaccia per facilitare lo sviluppo, il monitoraggio e la diagnostica del sistema ROS
* **rqt\_console** (log visualizer) è un visualizzatore di log di ros2
* Interazione con diversi **simulatori** come Gazebo(anch’esso un progetto della open robotics) , Webots, etc.
* **RViz** is a 3D visualizer for the Robot Operating System (ROS) framework.
* **rosdep** is a dependency management utility that can work with packages and external libraries.
* The **sros2** package provides the tools and instructions to use ROS 2 on top of DDS-Security.

## Introduction to NAV2 and Google Cartographer

## Descrizione di Nav2

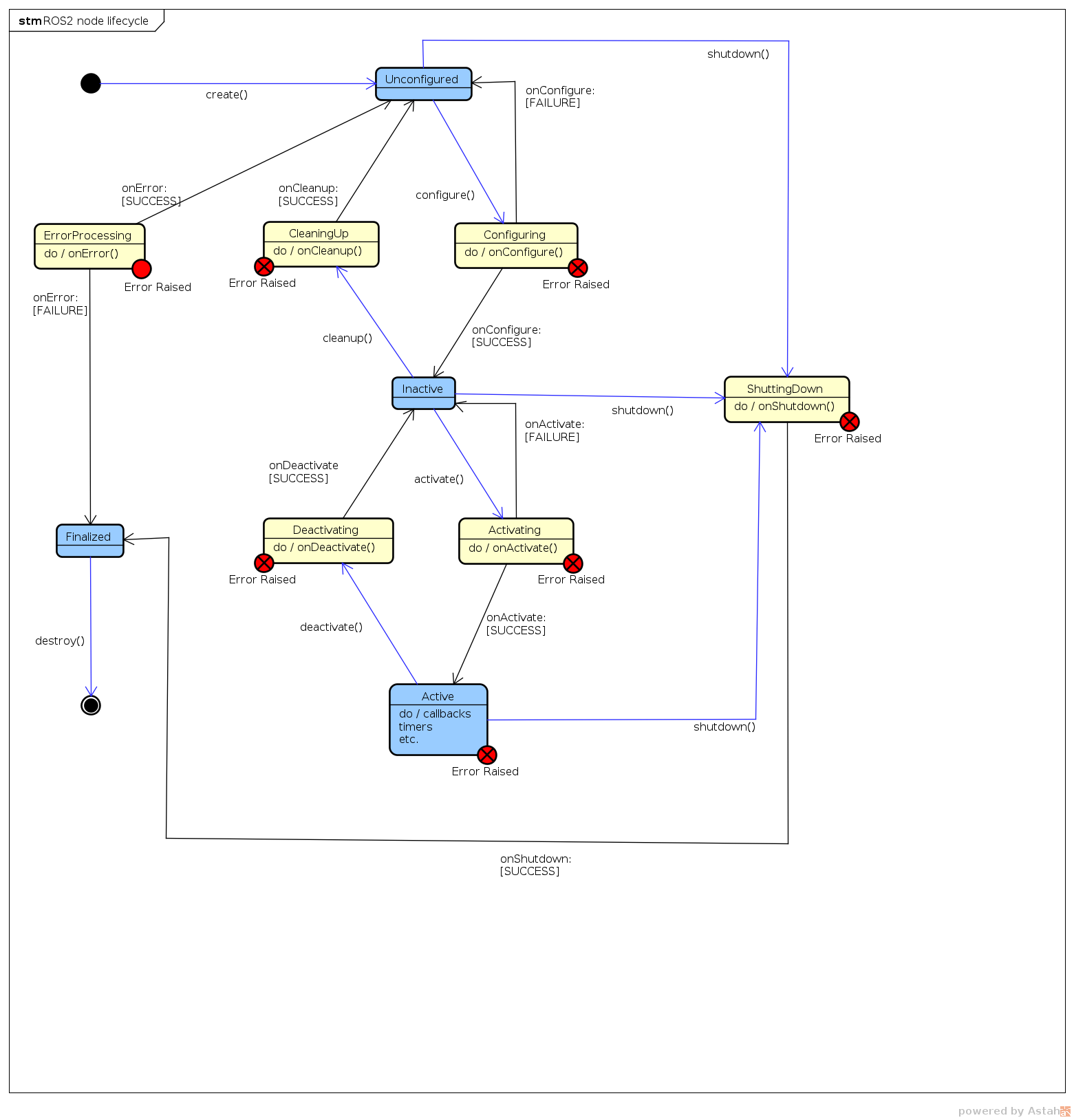
Nav2 sarà il sistema di controllo del nostro rover che, non solo abbiamo il passaggio da ROS 1 a ROS 2 ma anche abbiamo che il vecchio ROS Navigation Stack è stato sostituito da Nav2[[8]](#footnote-9)[[9]](#footnote-10)[[10]](#footnote-11) in sviluppo sotto la Open Navigation[[11]](#footnote-12).

Fornisce tutti gli strumenti (perception, planning[[12]](#footnote-13), control, localization[[13]](#footnote-14), visualization) per permettere a quasi qualsiasi robot di navigare in maniera altamente affidabile ed autonoma attraverso un ambiente complesso. Sviluppa un modello dell’ambiente dai dati dei sensori e dai dati semantici, calcola in maniera dinamica il percorso migliore evitando gli ostacoli regolando la velocità per i motori e fornisce una struttura per il comportamento ad alto livello con i behavior trees.

Complessivamente il sistema si aspetta un sistema di TF transformations conforming to REP-105, a map source if utilizing the Static Costmap Layer, a BT XML file, and any relevant sensor data sources per fornire i comandi di velocità per i motori of a holonomic or non-holonomic robot to follow.

Adesso andremo a descrivere meglio le singole parti che descrivono Nav2[[14]](#footnote-15):

* Riguardo a ROS 2 fa un buon uso dei nuovi strumenti implementati:
  + Action Server: come descritto in precedenza, sono una combinazione di topic e service utilizzati per compiere azioni che richiedono molto tempo. Forniscono un inizio di un azione con un feedback periodico e poi con una risposta sul completamento dell’azione. In nav2 sono implementati nella comunicazione dell’alto livello dei Behavior Tree (BT) di navigazione attraverso un NavigateToPose action message
  + Lifecycle Nodes (wrapper nav2\_util LifecycleNode) and Bond (connessione con il lifecycle manager): sono nodi che tengono una macchina a stati degli stati di ogni server. Dalla loro nascita (bringup) alla loro morte (teardown) in modo da avere un sistema deterministico del comportamento dei nodi di ROS. Gli stati principali dei nodi sono:
    - Partiamo dallo stato Unconfigurated dove costruiamo il nodo senza nessuna parte di ROS networking o lettura dei parametri
    - Inializziamo la configurazione con onConfigure: configurazione dei parametri, della ROS network interface e dinamically allocate memory
    - Che passiamo a Inactive per poi lanciare onActive: Attiva la ROS network interface
    - Arrivando allo stato active
    - Per spegnere la sequenza è, we transition into deactivating, cleaning up, shutting down and end in the finalized state. The networking interfaces are deactivated and stop processing, deallocate memory, exit cleanly, in those stages, respectively.



* Behavior trees[[15]](#footnote-16) (BT): sono una struttura ad albero dei task da completare in maniera human readable. provide a formal structure for navigation logic which can be both used to create complex systems but also be verifiable and validated as provenly correct using advanced tools. Having the application logic centralized in the behavior tree and with independent task servers (which only communicate data over the tree) allows for formal analysis. In Nav2 si usa BehaviorTree.CPP[[16]](#footnote-17) perché permette di essere caricata in altri alberi.
* Action server Planner, Controller, Smoother and Recovery Servers:
  + action servers are the planner, behavior, smoother and controller servers
  + planner, smoother and controller servers sono configurati at runtime con l’algoritmo (plugin),nell’elenco dei plugin registrati all’avvio del server, chiamato dal BT attraverso la loro action interface in modo che possano essercitare diverse funzioni con la stessa base.
  + The behavior server also contains a costmap subscriber to the local costmap, receiving real-time updates from the controller server, to compute its tasks. We do this to avoid having multiple instances of the local costmap which are computationally expensive
* Planners is to compute a valid, and potentially optimal, path (o route) from the current pose to a goal pose avendo accesso alla rappresentazione dell’ambiente globale global costmap e i dati dei sensori memorizzati al suo interno.
* Controllrs (ROS1 local planners) è utilizzato per trovare i comandi (velocità e sterzo) necessarie per seguire il global planner utilizzando il progress checker e il goal checker oppure per completare task locali come il docking usando una rappresentazione locale dell’ambiente local costmap. Molti controller proiettano il robot nello spazio in un tentativo di trovare un percorso fattibile.
* Recovery Behaviors: è il sistema per la gestione in maniera automatica degli errori dovuti a situazioni incognite e guasti.
* Smoothers serve per rimuovere irregolarità o rotazioni brusche dal path che il planner fornisce e ritorna la sua versione migliorata.
* Robot Footprints: realtime base del robot che può cambiare nel tempo
* Waypoint following: dato una serie di waypoinys implementa un metodo di navigazione tra di essi usando NavigationToPose action Server e può tenere un task BT ad ogni waypoint da eseguire.
* State estimation:per Nav2 bisogna avere 2 trasformazioni: una tra map to odom usando un sistema di posizionamento globale (localization, mapping, slam) e da odom to base\_link fornito da un odometry system e dovrebbe essere smooth and continuous local frame based on robot motion secondo la REP 105[[17]](#footnote-18)
* Costmap: 2D grid of cells containing a cost from unknown, free, occupied, or inflated cost. This costmap is then searched to compute a global plan or sampled to compute local control efforts.
* ~~Layer for costmap: implemented as pluginlib plugins to buffer information into the costmap. This includes information from LIDAR. Costmap layers can be created to detect and track obstacles in the scene for collision avoidance using camera or depth sensors.layers can be created to algorithmically change the underlying costmap based on some rule or heuristic. Finally, they may be used to buffer live data into the 2D or 3D world for binary obstacle marking.~~
* ~~Costmap filters: Costmap filters are a costmap layer-based approach of applying spatial-dependent behavioral changes, annotated in filter masks, into the Nav2 stack. Costmap filters are implemented as costmap plugins. These plugins are called “filters” as they are filtering a costmap by spatial annotations marked on filter masks. In order to make a filtered costmap and change a robot’s behavior in annotated areas, the filter plugin reads the data coming from the filter mask. This data is being linearly transformed into a feature map in a filter space. Having this transformed feature map along with a map/costmap, any sensor data and current robot coordinate filters can update the underlying costmap and change the behavior of the robot depending on where it is.~~

Queste parti descrivono il sistema Nav2 e ci forniscono tutti i mezzi necessari per il controllo del robot. Per l’inizializzazione la navigazione il robot deve sapere in che che lo circonda, in questo caso si utilizzano metodi di SLAM (Simultanuosly localization and Mapping) che come dicono nel nome forniscono sia un informazione sulla trasformazione odom -> map ma anche permettono di registrare i dati degli strumenti per creare una mappa da utilizzare in futuro per la navigazione. Attualmente NAV2 incorpora già il pacchetto slam toolbox che fornisce tutti gli strumenti per la creazione del mondo ma questa parte è stata già realizzata con Cartographer.

## Cartographer

Nel nostro caso, precedentemente è stata creata la mappa usando cartographer [[18]](#footnote-19) con un altro robot aziendale. Cartographer [[19]](#footnote-20)è una potente SLAM sviluppato da google realizzato in due subsystem: un local SLAM (chiamato Frontend) che costruisce in successione delle submaps che sono considerate localmente consistenti ma che possono drift durante l’acquisizione e un global SLAM (backend) è il suo obbiettivo è trovare loop closure contraints, facendo scan-matching \*\*scans\*\* (gathered in \*\*nodes\*\*) sulle submaps, in modo da legare le submaps assieme in modo da creare una mappa consistente.

Cartographer è un ottimo strumento per la creazione delle mappe ma però non viene più attivamente mantenuto. Difatti è stato usato per la creazione della mappa ma non sono riuscito ad utilizzarlo in maniera efficace per la sola localizzazione anche se ne prevede il funzionamento. Tutti i problemi riscontrati saranno descritti in seguito.

## Review of related work on porting navigation stack from ROS1 to ROS2

ROS è nato per uso accademico per il controllo di un robot ma si è diffuso anche in ambito commerciale nel quale ha riscontrato nuovi ambienti non pensati per l’uso accademico. I nuovi casi sono i seguenti:

* La gestione di gruppi di robot
* Sviluppo su piccole schede embedded
* Sistemi Real Time
* Network non ideali
* Orientarsi anche in ambito industriale
* Sviluppo di patterns e strumenti per la costruzione di sistemi

Tutto questo sviluppo ha portato allo sviluppo di ROS 2, maggiori informazioni sono disponibili nella documentazione di sviluppo[[20]](#footnote-21).

Inoltre si sono sviluppate nuove tecnologie per il suo sistema di comunicazione anonymous publish-subscribe middleware (RMW = ROS Middleware interface) che ora fa parte del DDS/RTPS dove DDS[[21]](#footnote-22)(Data-Distribution Service for Real-Time Systems) è uno standard industriale[[22]](#footnote-23). Aggiungendo questo layer modificabile è possibile usare una varietà di librerie opensource che permettono una riduzione del codice da controllare per gli sviluppatori di ROS ed inoltre ogni DDS è costruito in modo da adattarsi al meglio per diversi ambienti. Il DDS serve per comunicare facilmente tra i sistemi e condividere i dati. Low-level details like data wire format, discovery, connections, reliability, protocols, transport selection, QoS, security, etc. are managed by the middleware.

* DDS is uniquely **data centric**. data centricity is that DDS knows what data it stores and controls how to share that data.
* DDS conceptually sees a local store of data called the “**global data space**.” To the application, the global data space looks like local memory accessed via an API. information sharing is data-objects within Topics. there is no global place. the global data space is a virtual concept that is really only a collection of local stores.
* data can also be shared with flexible **Quality of Service** (QoS) specifications including reliability, system health (liveliness), and even security. is smart about sending just what it needs.
* **Dynamic Discovery** of publishers and subscribers. This means the application does not have to know or configure the endpoints for communications because they are automatically discovered by DDS. will discover if the endpoint is publishing data, subscribing to data, or both. It will discover the type of data. publisher’s offered communication characteristics and the subscriber’s requested communications characteristics. completed at runtime.
* **Scalable architecture**
* provide authentication, access control, confidentiality, and integrity to the information distribution. DDS **Security** uses a decentralized peer-to-peer architecture that provides security without sacrificing real-time performance.

Di default la RMW è eProsima Fast DDS (rmw\_fastrtps\_cpp). Nel nostro caso useremo Eclipse Cyclone DDS (rmw\_cyclonedds\_cpp) per motivi che verranno descritti in seguito.

Inoltre per gestire il traffico sulla rete utilizzeremo zenoh [[23]](#footnote-24)(Zero Overhead Network Protocol) ed il bridge per ros2 zenoh-bridge-ros2dds[[24]](#footnote-25) che è pub/sub/query protocol unifying data in motion, data at rest and computations. It elegantly blends traditional pub/sub with geo distributed storage, queries and computations, while retaining a level of time and space efficiency that is well beyond any of the mainstream stacks. data liberator protocol. Zenoh liberates data in several dimensions.

Questo ha portato a ROS2 mentre si abbandona ROS1 che conferma l’EOL per maggio 2025.

## Overview of open source fleet management systems, particularly Open-RMF

Attualmente questi sono i open source fleet management system disponibili sul mercato:

* **Isaac Mission Dispatch**[[25]](#footnote-26) del NVIDIA microservices enable fleet management software to submit missions to multiple robots and monitor the robot and mission states. It provides a connection between the robots and the fleet management system ***but does not handle logistics such as task allocation or conflict resolution***, e.g., robots with intersecting paths. The implementation relies VDA5050 protocol as an industry standard between a cloud control service and mobile robots, and uses MQTT as a lightweight, publish-subscribe, machine to machine network protocol designed for devices with resource constraints and limited network bandwidth
* **ROOSTER Fleet Management** [[26]](#footnote-27) : ROOSTER is a ROS 1 based open source project to develop a heterogeneous fleet management solution with task allocation, scheduling and routing capabilities.
* **Open-rmf[[27]](#footnote-28)**: is a free, open source, modular software system that enables sharing and interoperability between multiple fleets of robots and physical infrastructure, like doors, elevators and building management systems

Mentre molti altri forniscono una base per il controllo di una fleet:

* **Transitive Robotics[[28]](#footnote-29)**: is an open-source framework for full-stack robotics. It is designed to make it easy to create capabilities that seamlessly connect robot, cloud, and web front-ends including a dynamically updating shared data model. It offers solutions to many problems developers encounter when building full-stack robotic applications, including unreliable networks, robots that are regularly offline or turned off, as well as cross-device version dependencies.
* **Robot Web Tools[[29]](#footnote-30) :**  Insiemi di strumenti per la comunicazione con robot remoti.
* **Robofleet**[[30]](#footnote-31) : un altro Open Source Communication and Management for Fleets of Autonomous Robots

Per le capacità che offre open-rmf rispetto agli altri indicati viene scelto lui, qui di seguito riporto le abilità che ci hanno portato a sceglierlo.

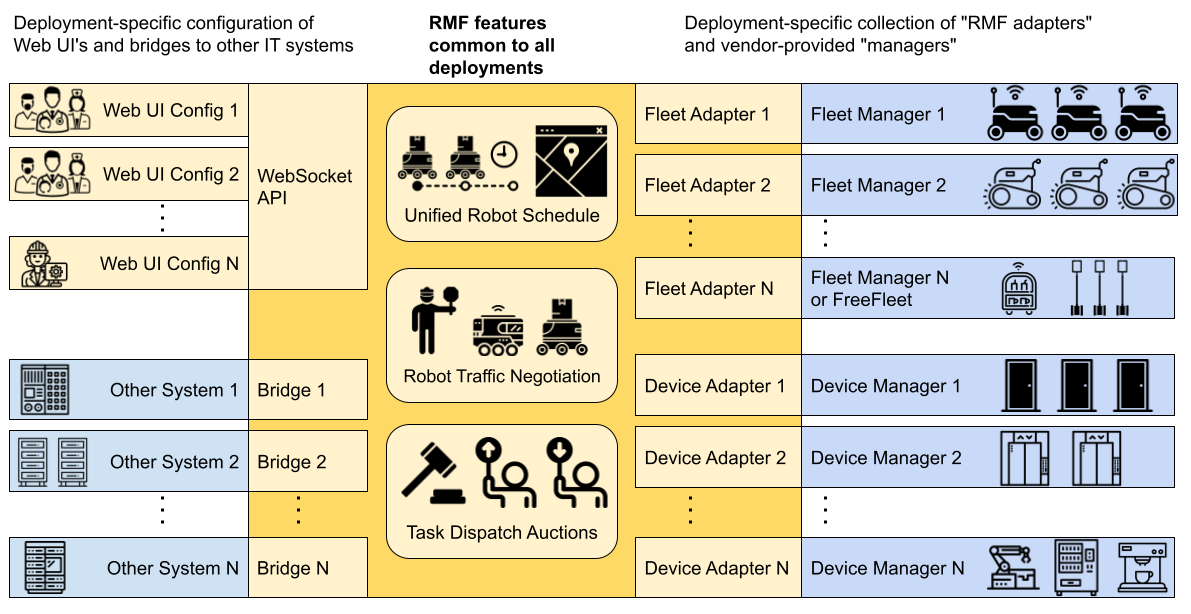
### Open-RMF description

Come descritto prima il sistema è un insieme di strumenti scritti sopra a ros2 per l’interazione tra flotte eterogenee di robot. Utilizza un protocollo standardizzato di comunicazione per le infrastrutture, ambienti e automazioni per ottimizzare le risorse critiche di un sistema di gestione del traffico (come altri robot, ascensori, porte, passaggi stretti). Fornisce intelligenza in un sistema evitando conflitti tra le risorse disponibili e assegnandole in modo corretto. Abbiamo un sistema abbastanza flessibile e robusto in modo che possa esser addattato per ogni struttura di rete e con diversi dispositivi IoT, attraverso un API o un interfaccia è possibile controllare RMF.

RMF è costruito su ROS2 ed integra RoMi-H (intercambiabile con SOSS [[31]](#footnote-32)) che è il gestore di comunicazione tra i vari middleware dei vari sistemi.

The core goal of RMF is to facilitate system integration for heterogeneous mobile robot fleets that may be provided by different vendors and may have different technical capabilities.

Qui di seguito è rappresentata la struttura complessiva di RMF:



RMF fornisce i seguenti strumenti per la gestione della sua infrastruttura:

* RMF demos: fornisce alcune simulazioni di esempio in modo da testare le capacità di Open-RMF
* Traffic Editor: fornisce gli strumenti per creare la mappa di come dovrà RMF gestire il traffico per i piani dell’edificio.
* Free Fleet: è un robot fleet management system che non hanno un proprio fleet manager
* RMF Schedule Visualizer: plugin for rviz per visualizzare e controllare RMF
* RMF Web UI:è una configurazione personalizzabile WEB per il controllo del sistema RMF
* RMF Simulation: Sono tutti i plugin per simulare su gazebo le capacità di RMF per le strutture e per i robot (come porte, ascensori, Crowd, robot slotcar, robot readonly, ...)
* Simulation assets: modelli per gazebo per aiutare lo sviluppo

Mentre gli algoritmi ed il loro funzionamento è il seguente:

RMF core:

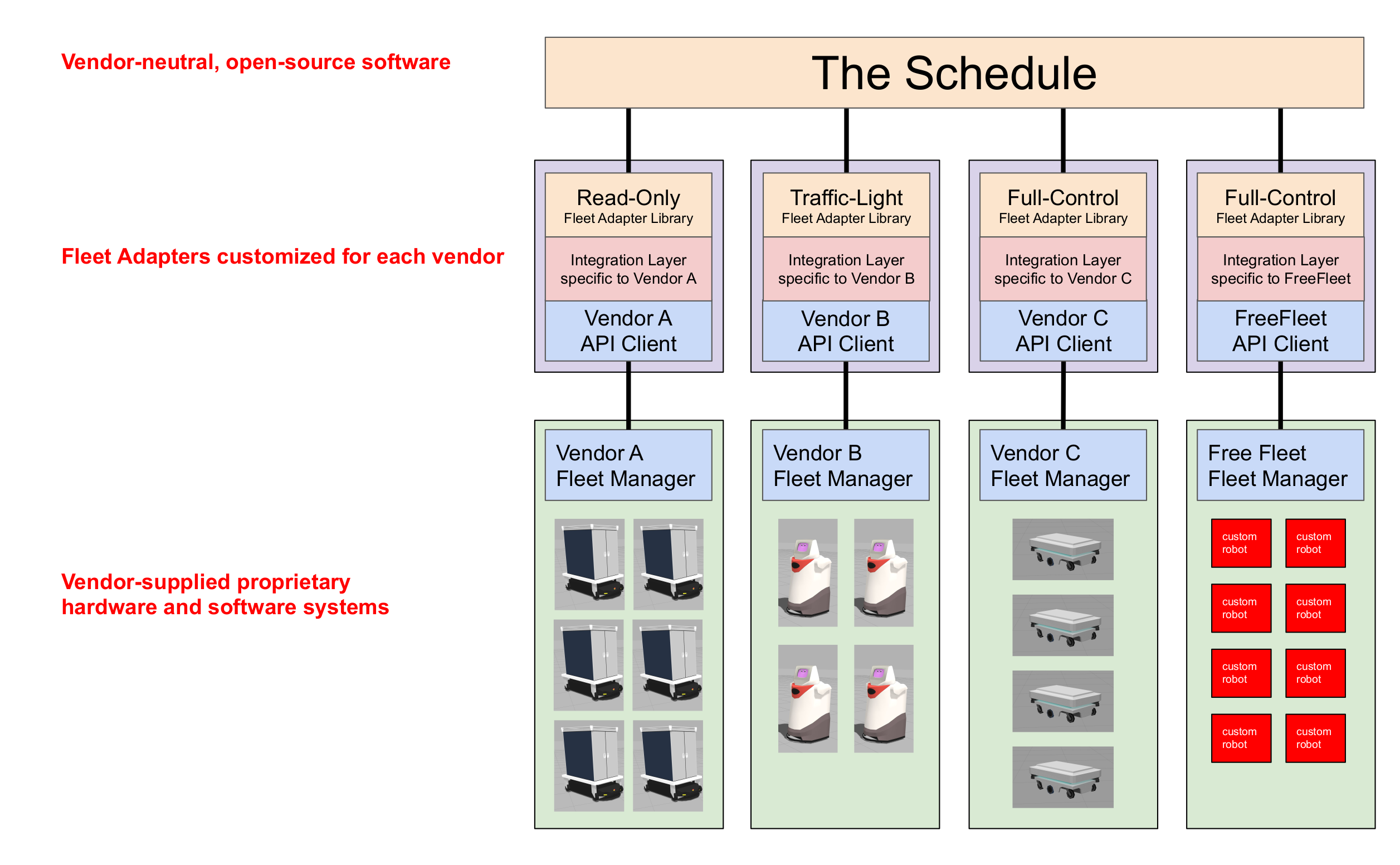
* rmf\_traffic: Core scheduling and traffic management systems
* rmf\_traffic\_ros2: rmf\_traffic for ros2
* rmf\_task: Task planner for rmf
* rmf\_battery: rmf battery estimation
* rmf\_ros2: ros2 adapters and nodes and python bindings for rmf\_core
* rmf\_utils: utility for rmf

Rmf\_traffic: usa i sistemi di traffic deconfliction: prevention e resolution.

* Prevention: la prevenzione è il metodo migliore per evitare la congestione. To facilitate traffic conflict prevention, we have implemented a platform-agnostic **Traffic Schedule** Database. The traffic schedule is a living database whose contents will change over time to reflect delays, cancellations, or route changes. All fleet managers that are integrated into an RMF deployment must report the expected itineraries of their vehicles to the traffic schedule. With the information available on the schedule, compliant fleet managers can plan routes for their vehicles that avoid conflicts with any other vehicles, no matter which fleet they belong to. rmf\_traffic provides a [Planner](https://github.com/open-rmf/rmf_traffic/blob/main/rmf_traffic/include/rmf_traffic/agv/Planner.hpp) class to help facilitate this for vehicles that behave like standard AGVs (Automated Guided Vehicles), rigidly following routes along a pre-determined grid. In the future we intend to provide a similar utility for AMRs (Autonomous Mobile Robots) that can perform ad hoc motion planning around unanticipated obstacles.
* Resolution: in caso che ci siano dei conflitti. rmf\_traffic has a Negotiation scheme. When the Traffic Schedule Database detects an upcoming conflict between two or more schedule participants, it will send a conflict notice out to the relevant fleet managers, and a negotiation between the fleet managers will begin. Each fleet manager will submit its preferred itineraries, and each will respond with itineraries that can accommodate the others. A third-party judge (deployed by the system integrator) will choose the set of proposals that is considered preferable and notify the fleet managers about which itineraries they should follow. There may be situations where a sudden, urgent task needs to take place (for example, a response to an emergency), and the current traffic schedule does not accommodate it in a timely manner. In such a situation, a traffic participant may intentionally post a traffic conflict onto the schedule and force a negotiation to take place. The negotiation can be forced to choose an itinerary arrangement that favors the emergency task by implementing the third-party judge to always favor the high-priority participant.

Traffic schedule:

centralized database of all the intended (it is looking into the future) robot traffic trajectories in a facility.The job of the schedule is to identify conflicts in the intentions of the different robot fleets and notify the fleets when a conflict is identified. Upon receiving the notification, the fleets will begin a traffic negotiation.



Fleet adapter: connects its fleet-specific API to the interfaces of the core RMF traffic scheduling and negotiation system. The fleet adapter is also responsible for handling communication between the fleet and the various standardized smart infrastructure interfaces, e.g. to open doors, summon lifts, and wake up dispensers.

Full control: RMF is provided with live status updates and full control over the paths that each individual mobile robot uses when navigating through the environment. This control level provides the highest overall efficiency and compliance with RMF, which allows RMF to minimize stoppages and deal with unexpected scenarios gracefully.

Traffic light: RMF is given the status as well as pause/resume control over each mobile robot, which is useful for deconflicting traffic schedules especially when sharing resources like corridors, lifts and doors

Read only: RMF is not given any control over the mobile robots but is provided with regular status updates. This will allow other mobile robot fleets with higher control levels to avoid conflicts with this fleet

No interface: Without any interface to the fleet, other fleets cannot coordinate with it through RMF, and will likely result in deadlocks when sharing the same navigable environment or resource. This level will not function with an RMF-enabled environment

can only ever be one "Read Only" fleet in a shared space, as any two or more of such fleets will make avoiding deadlock or resource conflict nearly impossible.

Currently we provide a reusable C++ API (as well as Python bindings) for integrating the Full Control category of fleet management. A preliminary ROS 2 message API is available for the Read Only category, but that API will be deprecated in favor of a C++ API (with Python bindings available) in a future release. The Traffic Light control category is compatible with the core RMF scheduling system, but we have not yet implemented a reusable API for it. To implement a Traffic Light fleet adapter, a system integrator would have to use the core traffic schedule and negotiation APIs directly, as well as implement the integration with the various infrastructure APIs

The API for the **Full Control** category is described in the [Mobile Robot Fleets](https://osrf.github.io/ros2multirobotbook/integration_fleets.html) section of the Integration chapter, and the **Read Only** category is described in the [Read Only Fleets](https://osrf.github.io/ros2multirobotbook/integration_read-only.html) section of the Integration chapter.

<https://osrf.github.io/ros2multirobotbook/rmf-core_faq.html>

These considerations led to the current design of distributed conflict prevention and distributed schedule negotiation

~~Open door is a command of the "fleet adapter" and get this info from the navigation graph that is provided to it. Optained with the traffic editor~~

~~Lisft not yet supported completely there are only demo~~

~~Assengation of a task~~

~~not implemented yet, there is a design worked out for a bidding system where a task request will be converted to a bid request. The bid request will be sent to each fleet adapter, and each fleet adapter that can perform the task will report its best estimate for how soon it would be able to have the task finished. The fleet adapter that offers the lowest bid will be assigned the task.~~

~~Scelta of the robot~~

~~in the current implementation that we are using, we treat all vehicles as equal and choose the resolution that minimizes the net delay across all the robots, without any prioritization or weighting. Vedi doc FAQ su come cambiare~~

~~Distance beetweeen robots~~

~~footprint\_radius and vicinity\_radius. The footprint\_radius represents an estimate of the vehicle's physical footprint. The vicinity\_radius represents an estimate of the region which the robot needs other vehicles to stay clear of. A "schedule conflict" is defined as an instance where one vehicle's "footprint" is scheduled to enter another vehicle's "vicinity". The job of the negotiation system is to come up with a fix to the schedule that keeps all vehicles' "footprints" out of all other vehicles' "vicinities".~~

~~The Dispatch Planner module is currently a work-in-progress.~~

~~rmf\_traffic provides a middleware-neutral implementation of the core traffic scheduling algorithms and utilities. It does not use or depend on ROS 2.~~

~~What is rmf\_traffic\_ros2 ?~~

~~rmf\_traffic\_ros2 provides convenient wrappers for using the rmf\_traffic as part of a distributed ROS 2 system.~~

concerned about identifying conflicts between the intended routes of autonomous vehicles.

Conflict avoidance for AGV's in rmf\_traffic is implemented with time-dependent extension to [A\* search](https://en.wikipedia.org/wiki/A*_search_algorithm) This search takes time into account so that it can find paths through space and time that account for the motions of other agents that are in the traffic schedule.

### [**During negotiation, how do fleets compute their proposals?**](https://osrf.github.io/ros2multirobotbook/rmf-core_faq.html#during-negotiation-how-do-fleets-compute-their-proposals)

Because the system is designed to be extensible and adaptable to a wide variety of scenarios and robot vendor combinations, including many that do not currently exist, it has many pieces and hooks for expansion. The sequence of computing a traffic proposal is as follows:

* First, the fleet adapter node will receive a conflict notification which tells it that it needs to participate in a negotiation to resolve a space-time conflict. This notification is received by the rmf\_traffic\_ros2::schedule::Negotiation class.
* For this to happen, the fleet adapter creates a negotiation-notification subscription, so that it will be told whenever a particular robot under its control needs to respond to a negotiation notification.
* When a robot needs to repond to a negotiation, its implementation gets triggered.
* This implementation will launch a multi-threaded Negotiate service, whose main implementation can be found [here](https://github.com/osrf/rmf_core/blob/master/rmf_fleet_adapter/src/rmf_fleet_adapter/services/detail/impl_Negotiate.hpp)

Every step in the multi-party negotiation is using the Negotiate service the exact same way. The only difference between the various steps is what constraints they need to deal with. Those constraints are described by the rmf\_traffic::schedule::Negotiation::Table::Viewer object that gets passed to the respond(~) function. Because the same object can be used to describe the constraints of all the different blocks in the diagram, we can use the same code to solve every block.

There are also reject and forfeit code paths that may be invoked as necessary:

* The "rejection" mechanism is used when it's impossible for one of the fleets to accommodate a proposal that came from another. When a rejection is performed, the rejecting fleet will provide a set of feasible trajectories (usually anywhere from 10-200 trajectories) and the fleet that receives the rejection should try once again to find an ideal proposal for itself, but that ideal proposal must accommodate at least one of the trajectory alternatives that were provided with the rejection.
* The "forfeit" mechanism is used when the planner is having an inordinately difficult time finding any kind of solution. This can happen when the negotiation has numerous participants that are all actively on the move, which can lead to situations that are seemingly impossible to resolve due to inconsistencies across time. Typically, when a forfeit is used, there will be another feasible combination of accommodations that gets found by the negotiation. In the worst case scenario, if negotiations keep failing, the robots may experience a real-life deadlock. When a deadlock happens, the participants will be sitting still so the negotiation will reach a steady state and will not be negatively affected by async inconsistencies. When that happens, a successful resolution is practically assured.

~~TASK~~

task requests out of the box:

* Clean: For robots capable of cleaning floor spaces in facilities
* Delivery: For robots capable of delivering items between locations in facilities
* Loop: For robots capable to navigating back and forth between locations in facilities

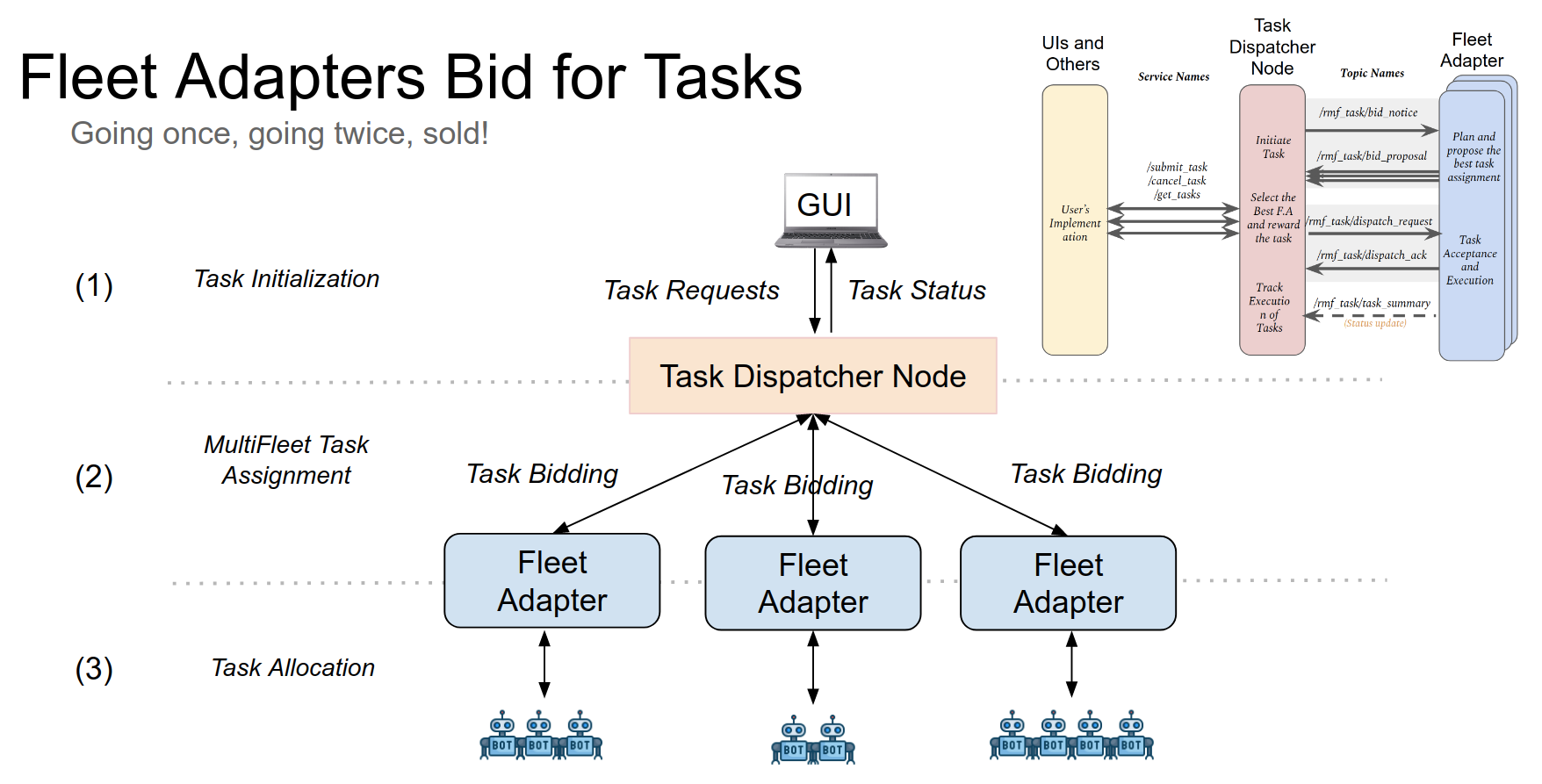
RMF simplifies task allocation and management across multi-fleet systems. When a user submits a new task request, RMF will intelligently assign it to the robot in the fleet that can best perform the task

tasks are awarded to robot fleets based on the outcome of a bidding process that is orchestrated by a Dispatcher node, rmf\_dispatcher\_node. When the Dispatcher receives a new task request from a dashboard or terminal, it sends out a rmf\_task\_msgs/BidNotice message to all the fleet adapters. If a fleet adapter is able to process that request, it submits a rmf\_task\_msgs/BidProposal message back to the Dispatcher with a cost to accommodate the task. An instance of rmf\_task::agv::TaskPlanner is used by the fleet adapters to determine how best to accommodate the new request

The Dispatcher then compares all the BidProposals received and submits a rmf\_task\_msgs/DispatchRequest message with the fleet name of the robot that the bid is awarded to.

Dispatcher evaluates the proposals such as fastest to finish, lowest cost,

ChargeBattery tasks are optimally injected into a robot's schedule when the robot has insufficient charge to fulfill a series of tasks. Currently we assume each robot in the map has a dedicated charging location as annotated with the is\_charger option in the traffic editor map.



Clean

hence many varieties of cleaning robots, the workflow remains identical across them all. Floor space in facilities is divided into a number of “zones” or sub-regions for cleaning. Each zone has a start and finish location for the robot. In-between these locations, the robot traverses along a special path while performing the cleaning operation.

Further, RMF can intelligently assign cleaning jobs to available cleaning robots based on capability and available resources, while optimizing overall productivity. Most cleaning robots have a pre-configured cleaning routine for each zone that can be run from a given starting point. RMF’s goal is to guide the robot to this starting point, trigger the execution of the cleaning routine and then guide the robot to a holding waypoint once the cleaning is complete.

To estimate the resource drain from the cleaning process which is essential for optimal task allocation planning, the fleet adapters require the list of waypoints that the robot will traverse while cleaning

Delivery Task:

delivery typically involves the robot heading to a pickup location where it gets loaded with items and then navigating to a dropoff location where the items are unloaded. At the pickup and dropoff sites, the mobile robot may have to interface with robotic arms, conveyors or other automation systems. We term systems that load items as dispensers and those that unload as ingestors.

To integrate these systems with RMF core systems, a set of [dispenser](https://github.com/open-rmf/rmf_internal_msgs/tree/main/rmf_dispenser_msgs/msg) and [ingestor](https://github.com/open-rmf/rmf_internal_msgs/tree/main/rmf_ingestor_msgs/msg) messages are defined.

A Delivery task is designed in RMF which guides the mobile robot to the pickup location where the dispenser is located. Once here, its rmf\_fleet\_adapter publishes a DispenserRequest message which the workcell receives and begins processing. When the loading is successful, the dispenser publishes a DispenserResult message with SUCCESS status. The rmf\_fleet\_adapter then guides the robot to the dropoff waypoint where the ingestor is located. Here, a similar exchange of messages ensures. The rmf\_fleet\_adapter publishes an IngestorRequest message which instructs the ingestor to unload its payload. Upon completion, it publishes an IngestorResult message with a SUCCESS status.

Loop Task:

A Loop task can be submitted to request a robot to navigate back and forth between two waypoints for a given number of iterations (loops).

ChargingTask

This is an self-generated task, self generated by RMF fleet adapter. The robot will get directed back to the charging station when charging condition is met. User can set a valid charging station by setting is\_parking\_spot to true in traffic\_editor. Note that currently finishing\_request arg also supports: [charge, park, nothing]. These are considered as autogenerated tasks.

cUTOM TASK

When dealing with RMF tasks, there are two packages:

* [rmf\_task](https://github.com/open-rmf/rmf_task/tree/main/rmf_task)
* [rmf\_task\_sequence](https://github.com/open-rmf/rmf_task/tree/main/rmf_task_sequence)

rmf\_task provides APIs and base classes for defining and managing Tasks in RMF. A Task is defined as an object that generates phases which are a meaningful sequence of steps that results in a desirable outcome.

rmf\_task\_sequence provides an out of the box implementation of rmf\_task where a Task object is defined by a sequence of phases. The phases that such tasks generate will thus match the sequence of phases used to define them. The phases defined in rmf\_task\_sequence are in turn a collection of events which also have components to model the end state and command the robot during the event.

*There's been significant progress made on the foundational tools for this, most of it taking place in the* [*bevy\_impulse*](https://github.com/open-rmf/bevy_impulse) *repo (documentation pending). A proof of concept for executing directed acyclic graphs is finished, and I'm close to finishing a major rework which will support generalized workflow diagrams, much like this discussion originally described, although more streamlined.*

*The other pieces that are currently in flight include*

* [*rclrs*](https://github.com/ros2-rust/ros2_rust) *which will be needed to communicate between Rust-implemented nodes*
* [*site editor*](https://github.com/open-rmf/rmf_site) *which will allow users to create more expressive robot behaviors, and will likely provide a GUI foundation for developing a workflow editor*
* [*mapf*](https://github.com/open-rmf/mapf) *which will enable much more sophisticated multi-agent path finding than what we currently get out of* [*rmf\_traffic*](https://github.com/open-rmf/rmf_traffic)

*Once all these pieces are in place we'll be well positioned to migrate the fleet adapter implementations over to this new architecture that can accommodate general task workflows. I put together some slides about the vision and motivation for this new architecture* [*here*](https://docs.google.com/presentation/d/1XE4A_72Y0qLHpkoXuEI4dEKK3Jf0zM7cgX1yj1XTEJg/edit?usp=sharing)*.*

*Clearly we still have a lot of ground to cover before this capability is usable, but our progress on this has been ramping up rapidly in the last few months, and this transition will soon become our central focus.*

<https://github.com/open-rmf/rmf/discussions/632>

<https://osrf.github.io/ros2multirobotbook/task_userdefined.html>

The new flexible task system introduces the concept of a Phase. A task is an object that generates phases. In other words, a task is typically made up of a series or combination of phases as its building blocks.

Users can use the following public API phases to construct their own tasks:

rmf\_fleet\_adapter/schemas/event\_description\_\_perform\_action.json

Users can build and send their own tasks by publishing [ApiRequest](https://github.com/open-rmf/rmf_internal_msgs/blob/main/rmf_task_msgs/msg/ApiRequest.msg) messages. You will need to fill in the request\_id and json\_msg fields according to the types of phases that make up the task, as well as whether the task is intended for a specific robot or the best available fleet.

## [**Task Management Control**](https://osrf.github.io/ros2multirobotbook/task_new.html#task-management-control)

You may take additional control over your tasks by sending requests to RMF to cancel a task or skip a phase. A full list of JSON schemas for such requests are defined [here](https://github.com/open-rmf/rmf_api_msgs/tree/main/rmf_api_msgs/schemas).

# [**SOSS**](https://osrf.github.io/ros2multirobotbook/soss.html#soss)

This chapter describes the system-of-systems synthesizer (SOSS), a tool which provides protocol translation between different subsystems. Such composite systems can be called The ROS-SOSS

The ecosystem of different message passing systems is vast and diverse. Without any one single system that is unanimously considered the best for all applications, we are left to consider how we can tie together disparate message passing systems to bridge the gap between the different kinds of applications that all play a critical role in a modern, intelligent robotics solution. The best protocols to use for inter-robot communication might not be the best for remote operator communication or for end-user communication.

The integration service we use for this is called System-of-Systems Synthesizer, or SOSS. The base SOSS package is simply some abstract interfaces defined in a C++ library, along with a single soss application. Each different message-passing system will have its own plugin library, e.g. DDS-SOSS, Websocket-SOSS, ROS-SOSS, that implements the abstract interfaces of the base SOSS. When you run the soss application, you provide a single configuration file that describes how you want your different message-passing systems to bridge with each other. The soss application can then find plugins that meet the requirements of your configuration file and load those plugins as it starts up. When messages start to move within each message-passing system, the soss application will grab, translate, and push the messages across the different system boundaries according to the configuration file that was given to it. Any number of soss instances can be run at once, but they will run independently of each other, so it is important to make sure that their configurations do not overlap.

Esempio non è detto che usano proprio questo:

<https://integration-service.docs.eprosima.com/en/latest/index.html>

<https://openrmf.readthedocs.io/en/latest/index.html>

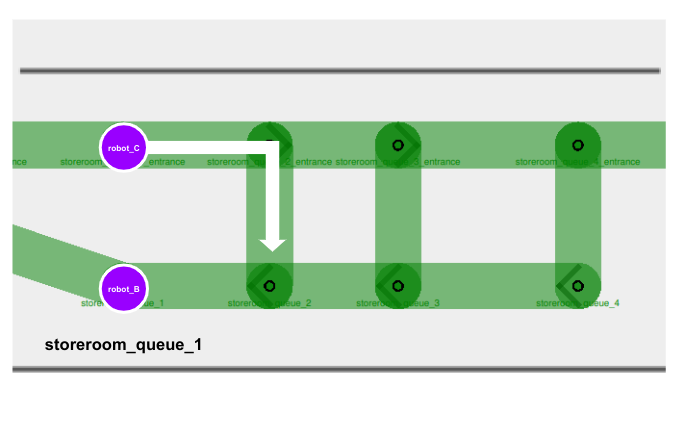
It works by translating the languages of the systems involved to a common representation language that follows the [Extensible and Dynamic Topic Types for DDS](https://www.omg.org/spec/DDS-XTypes/About-DDS-XTypes/) (**xTypes**) standard by the [OMG](https://www.omg.org/); specifically, *Integration Service* bases its intercommunication abilities on eProsima's open source implementation for the *xTypes* protocol, that is, [eProsima xTypes](https://github.com/eProsima/xtypes).

The translation is mediated by system-specific plugins, or **System Handles**, allowing to communicate the middlewares involved with the **core**, that speaks the common *xTypes* representation language. This is carried out by interfacing each entity in the user application with a complimentary "mirror" proxy in the *System Handle* (SH), able to communicate with the former according to either a pub/sub or service pattern.

Navigation

RMF uses robot route maps to predict the navigation paths of robots working in the environment. RMF generates path predictions for all active robots in the environment which can be used to proactively avoid conflicts between the various robot path plans. This is often referred to as "traffic management" in RMF. Along with the traffic management, RMF can help enable multi-fleet visualization to building/robot operations staff, improve scheduling of resources (such as lifts and corridors), reduce robot deadlock and more.

In your fleet adapter or a separate ROS 2 node, you can add a function to detect waypoint occupancy. If the waypoint storeroom\_queue\_1 is occupied, publish a LaneRequest to close the waypoint's bridging lane. Any other robots that come along will be restricted to join the queue on the next waypoint.



Ti fornisce codice per il full control:  
The C++ API for **Full Control** automated guided vehicle (AGV) fleets can be found in the [rmf\_fleet\_adapter](https://github.com/open-rmf/rmf_ros2/tree/main/rmf_fleet_adapter) package of the rmf\_ros2 repo. The API consists of four critical classes:

* [Adapter](https://github.com/open-rmf/rmf_ros2/blob/main/rmf_fleet_adapter/include/rmf_fleet_adapter/agv/Adapter.hpp) - Initializes and maintains communication with the other core RMF systems. Use this to register one or more fleets and receive a FleetUpdateHandle for each fleet.
* [FleetUpdateHandle](https://github.com/open-rmf/rmf_ros2/blob/main/rmf_fleet_adapter/include/rmf_fleet_adapter/agv/FleetUpdateHandle.hpp) - Allows you to configure a fleet by adding robots and specifying settings for the fleet (e.g. specifying what types of deliveries the fleet can perform). New robots can be added to the fleet at any time.
* [RobotUpdateHandle](https://github.com/open-rmf/rmf_ros2/blob/main/rmf_fleet_adapter/include/rmf_fleet_adapter/agv/RobotUpdateHandle.hpp) - Use this to update the position of a robot and to notify the adapter if the robot's progress gets interrupted.
* [RobotCommandHandle](https://github.com/open-rmf/rmf_ros2/blob/main/rmf_fleet_adapter/include/rmf_fleet_adapter/agv/RobotCommandHandle.hpp) - This is a pure abstract interface class. The functions of this class must be implemented to call upon the API of the specific fleet manager that is being adapted.

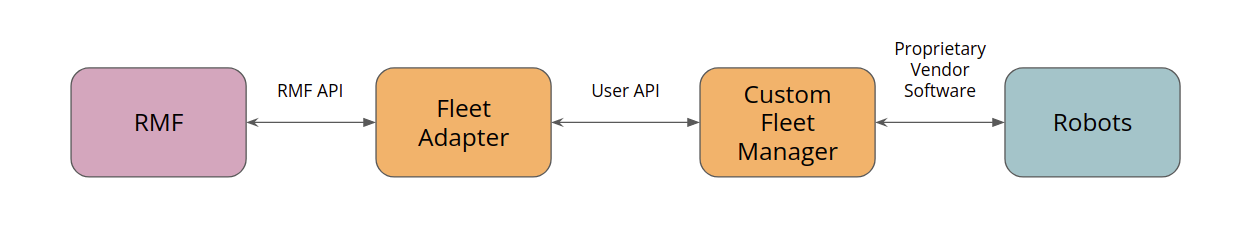
Ma c’è una versione semplice Easy Full control : Full Control library without having to modify its internal logic.

The C++ API for Traffic Light Control fleets (i.e. fleets that only allow RMF to pause/resume each mobile robot)

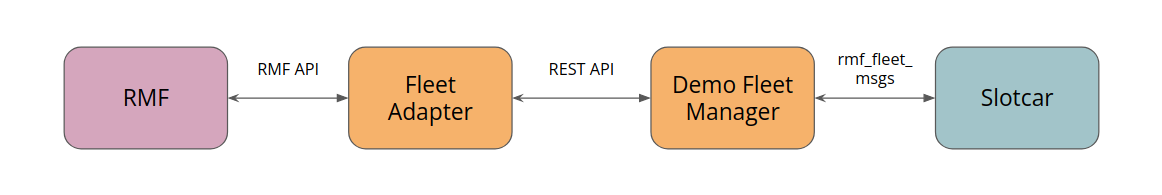
Also python bindings - rmf\_ros2/rmf\_fleet\_adapter\_python/

Ancora piu semplice: **Full Control** [template package](https://github.com/open-rmf/fleet_adapter_template) where users only need to update certain blocks of code with the API calls to their specific robot/fleet manager. This template uses the EasyFullControl API, where user-implemented robot callbacks are executed by RMF

Comandi con with REST or websocket based APIs



In caso che non ci sia un fleet manager fornisce anche un manager



The fleet\_adapter receives information (position, current ongoing tasks, battery levels etc.) about each robot in the fleet and sends them to the core RMF system for task planning and scheduling.

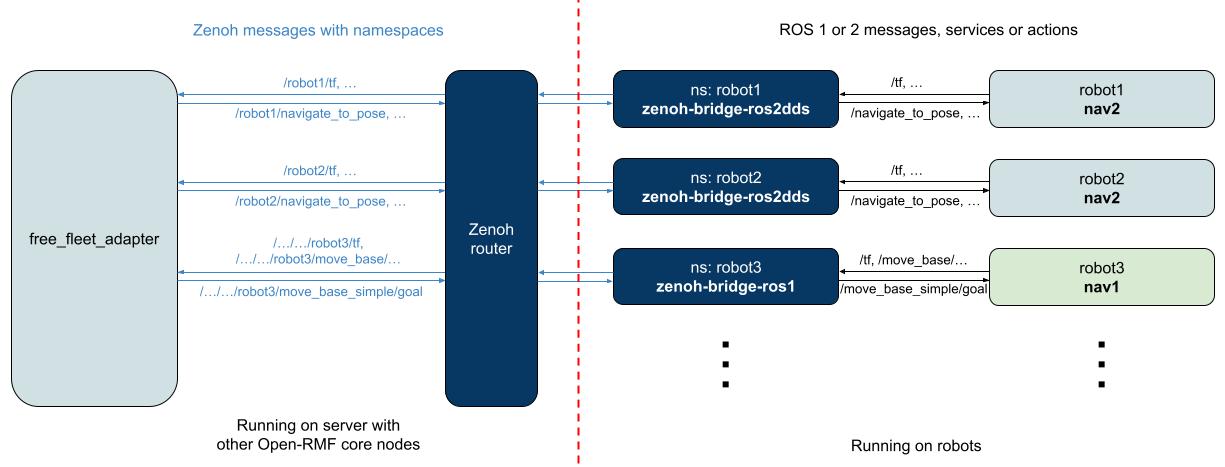
* When the core RMF system has a task to dispatch, it communicates with the various fleet adapters to check which fleet is suitable for taking this task.
* It sends a request, to which fleet adapters respond by submitting their fleet robots' availability and statuses.
* RMF determines the best fleet for the task and responds to the winning bid, i.e. the fleet that is selected. The response contains navigation commands relevant to the delegated task.
* The fleet adapter will then send the navigation commands to the robot in appropriate API.

Fleet adapter template

<https://osrf.github.io/ros2multirobotbook/integration_free_fleet_adapter.html#free-fleet-adapter>

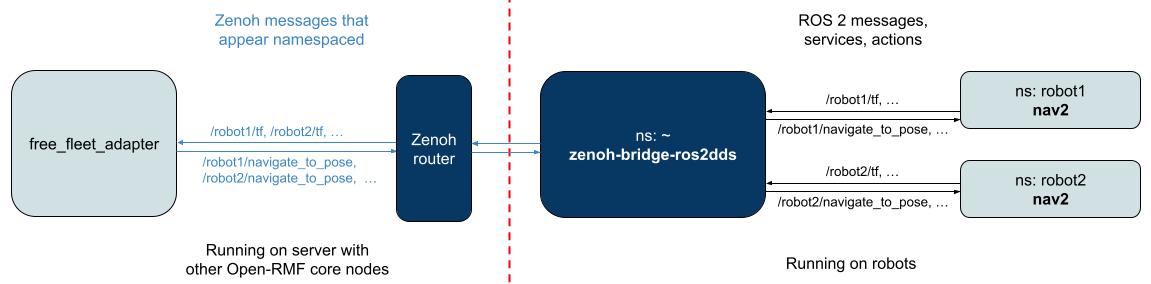
The free fleet adapter formerly known as Free Fleet, allows users to integrate standalone mobile robots to form a heterogeneous fleet, to work with Open-RMF.

The free\_fleet\_adapter implements the Easy Full Control fleet adapter API, and communicates with individual robots over Zenoh bridges. Out-of-the-box, it supports robots running [ROS 1 Navigation](https://wiki.ros.org/navigation) as well as [ROS 2 Navigation2](https://github.com/ros-navigation/navigation2).



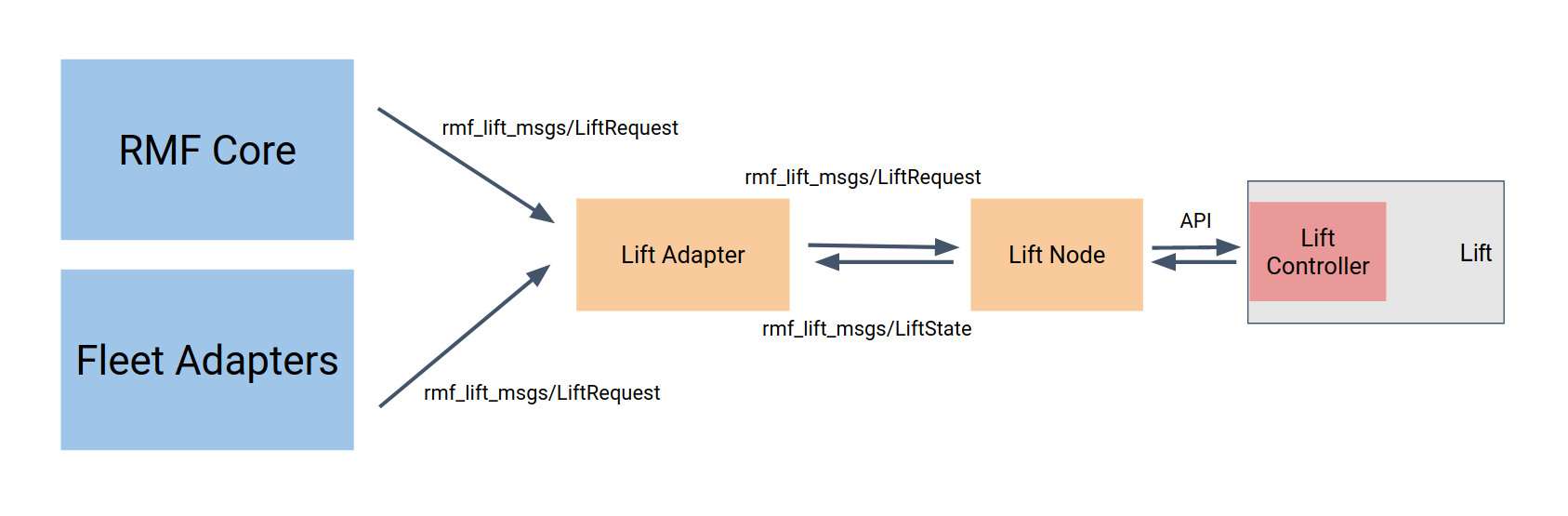
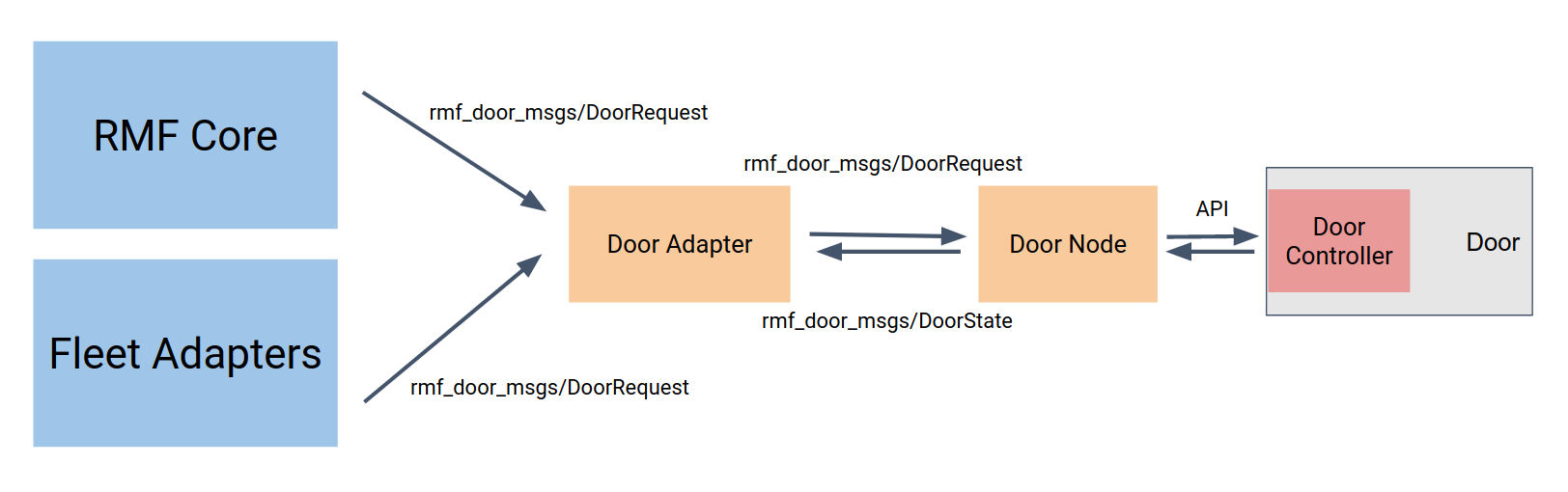
With Zenoh bridges, users should be able to integrate their robot as-is, without additional namespacing. The Zenoh bridges are responsible for filtering topics to reduce network traffic, and to namespace the various Zenoh keys, which the free\_fleet\_adapter operates on. Handling message definitions across domains have been greatly simplified too,

For ROS 2 Navigation2, the integration is enabled by [zenoh-bridge-ros2dds](https://github.com/eclipse-zenoh/zenoh-plugin-ros2dds). With an appropriately configured bridge running on the robot, the free\_fleet\_adapter is able to communicate and control the robot, via its navigation behavior trees.



The Navigation2 integration currently uses the simple navigate\_to\_pose behavior to provide navigation commands from point to point following the RMF waypoints defined.

Users looking to integrate their own custom navigation stack, can implement the abstract interfaces in RobotAdapter [here](https://github.com/open-rmf/free_fleet/blob/main/free_fleet_adapter/free_fleet_adapter/robot_adapter.py).



Currently RMF has 2 types of sample workcells, namely: Dispenser and Ingestor.

security

This chapter describes how to use DDS Security tools to provide authentication, encryption, and access control to an RMF system.

The security of the RMF system can be divided into two main parts: its ROS 2 elements and the dashboard. The security of the ROS 2 elements is provided by the DDS security tools which help ensure authentication, encryption and access control. The dashboard provides the user with an instruments panel while ensuring encryption, integrity and authentication of the connection to the server through TLS. User authentication and access control is made by user/password checking against a database and then providing that user with access to the secured ROS 2 network at a level that corresponds to the role of that user.

<https://docs.ros.org/en/rolling/Tutorials/Advanced/Security/Introducing-ros2-security.html>

@misc{vilches2022sros2usablecybersecurity,

title={SROS2: Usable Cyber Security Tools for ROS 2},

author={Victor Mayoral Vilches and Ruffin White and Gianluca Caiazza and Mikael Arguedas},

year={2022},

eprint={2208.02615},

archivePrefix={arXiv},

primaryClass={cs.CR},

url={https://arxiv.org/abs/2208.02615},

}

<https://github.com/ros2/sros2?tab=readme-ov-file>

<https://github.com/ros2/rmw_zenoh/tree/rolling>

## [**RMF Web Dashboard Security**](https://osrf.github.io/ros2multirobotbook/security.html#rmf-web-dashboard-security)

[RMF Web] (https://github.com/open-rmf/rmf-web) is a web application that provides overall visualization and control over the RMF system. It is served over TLS to ensure encryption, integrity and authentication of the communication with the final user. The server uses [openid-connect (OIDC)](https://openid.net/connect/) for authentication, an open standard based on [oauth 2.0](https://oauth.net/2/) and [JOSE](https://jose.readthedocs.io/en/latest/). Currently the dashboard makes use of [Keycloack](https://www.keycloak.org/), an open source implementation of OIDC. It provides an user management system, which is used to create/delete users. Each user gets assigned a role which is reflected on an id token generated on the user. This id token is signed securely and sent to the api server which it can then look at the role of the user and act accordingly. The api server runs a secured ROS 2 node per each role and provides access to them based on the id token of each user.

# Methodology

## Description of research platform and hardware used

## Steps taken to port the navigation stack from ROS1 to ROS2

## Hardware adaptation process from Rover Zero to Rover Mini

## Integration of the navigation stack with Open-RMF

## Testing and evaluation of the system

# Results and Analysis

## Presentation of results from testing and evaluation

## Comparison of system performance before and after the porting process

## Discussion on the challenges encountered and how they were addressed

# Conclusion and Future Work

## Summary of the research findings

## Contributions to the field of electrical engineering and robotics

## Recommendations for future work and improvements

# References [[references]]

## List of sources cited in the thesis

# Appendices

## Additional technical details and documentation

1. <https://www.open-rmf.org/> [↑](#footnote-ref-2)
2. <https://doi.org/10.1007/978-3-031-45770-8_29> [↑](#footnote-ref-3)
3. https://roverrobotics.com/ [↑](#footnote-ref-4)
4. S. Macenski, T. Foote, B. Gerkey, C. Lalancette, W. Woodall, “Robot Operating System 2: Design, architecture, and uses in the wild,” Science Robotics vol. 7, May 2022.

   @article{

   doi:10.1126/scirobotics.abm6074,

   author = {Steven Macenski and Tully Foote and Brian Gerkey and Chris Lalancette and William Woodall },

   title = {Robot Operating System 2: Design, architecture, and uses in the wild},

   journal = {Science Robotics},

   volume = {7},

   number = {66},

   pages = {eabm6074},

   year = {2022},

   doi = {10.1126/scirobotics.abm6074},

   URL = {https://www.science.org/doi/abs/10.1126/scirobotics.abm6074}

   } [↑](#footnote-ref-5)
5. <https://www.ros.org/> [↑](#footnote-ref-6)
6. <https://www.openrobotics.org/> [↑](#footnote-ref-7)
7. <https://docs.ros.org/en/rolling/index.html> [↑](#footnote-ref-8)
8. <https://docs.nav2.org/> [↑](#footnote-ref-9)
9. @article{macenski2023survey, title={From the desks of ROS maintainers: A survey of modern & capable mobile robotics algorithms in the robot operating system 2}, author={S. Macenski, T. Moore, DV Lu, A. Merzlyakov, M. Ferguson}, year={2023}, journal = {Robotics and Autonomous Systems}} [↑](#footnote-ref-10)
10. @article{macenski2024smac, title={Open-Source, Cost-Aware Kinematically Feasible Planning for Mobile and Surface Robotics}, author={Steve Macenski and Matthew Booker and Josh Wallace}, year={2024}, journal = {Arxiv}} [↑](#footnote-ref-11)
11. <https://www.opennav.org/> [↑](#footnote-ref-12)
12. @article{macenski2023regulated, title={Regulated Pure Pursuit for Robot Path Tracking}, author={Steve Macenski and Shrijit Singh and Francisco Martin and Jonatan Gines}, year={2023}, journal = {Autonomous Robots}} [↑](#footnote-ref-13)
13. @InProceedings{vslamComparison2021, author = {Merzlyakov, Alexey and Macenski, Steven}, title = {A Comparison of Modern General-Purpose Visual SLAM Approaches}, booktitle = {2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)}, year = {2021}} [↑](#footnote-ref-14)
14. @InProceedings{macenski2020marathon2, author = {Macenski, Steven and Martin, Francisco and Whit@InProceedings{vslamComparison2021, author = {Merzlyakov, Alexey and Macenski, Steven}, title = {A Comparison of Modern General-Purpose Visual SLAM Approaches}, booktitle = {2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)}, year = {2021}}

    @InProceedings{macenski2020marathon2, author = {Macenski, Steven and Martin, Francisco and White, Ruffin and Ginés Clavero, Jonatan}, title = {The Marathon 2: A Navigation System}, booktitle = {2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)}, year = {2020}}

    e, Ruffin and Ginés Clavero, Jonatan}, title = {The Marathon 2: A Navigation System}, booktitle = {2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)}, year = {2020}} [↑](#footnote-ref-15)
15. @misc{Colledanchise\_2018,

    title={Behavior Trees in Robotics and AI},

    ISBN={9780429950902},

    url={http://dx.doi.org/10.1201/9780429489105},

    DOI={10.1201/9780429489105},

    publisher={CRC Press},

    author={Colledanchise, Michele and Ögren, Petter},

    year={2018},

    month=jul } [↑](#footnote-ref-16)
16. <https://www.behaviortree.dev/> [↑](#footnote-ref-17)
17. <https://www.ros.org/reps/rep-0105.html> [↑](#footnote-ref-18)
18. <https://github.com/cartographer-project/cartographer> [↑](#footnote-ref-19)
19. W. Hess, D. Kohler, H. Rapp, and D. Andor,

    [Real-Time Loop Closure in 2D LIDAR SLAM](https://research.google.com/pubs/pub45466.html), in

    \*Robotics and Automation (ICRA), 2016 IEEE International Conference on\*.

    IEEE, 2016. pp. 1271–1278. [↑](#footnote-ref-20)
20. <https://design.ros2.org/> [↑](#footnote-ref-21)
21. <https://www.dds-foundation.org/> [↑](#footnote-ref-22)
22. <https://www.omg.org/omg-dds-portal> [↑](#footnote-ref-23)
23. <https://zenoh.io/> [↑](#footnote-ref-24)
24. <https://github.com/eclipse-zenoh/zenoh-plugin-ros2dds> [↑](#footnote-ref-25)
25. <https://github.com/NVIDIA-ISAAC/isaac_mission_dispatch> [↑](#footnote-ref-26)
26. <https://github.com/ROOSTER-fleet-management> [↑](#footnote-ref-27)
27. <https://www.open-rmf.org/> [↑](#footnote-ref-28)
28. <https://transitiverobotics.com/> [↑](#footnote-ref-29)
29. <https://robotwebtools.github.io/> [↑](#footnote-ref-30)
30. @misc{sikand2021robofleetopensourcecommunication,

    title={Robofleet: Open Source Communication and Management for Fleets of Autonomous Robots},

    author={Kavan Singh Sikand and Logan Zartman and Sadegh Rabiee and Joydeep Biswas},

    year={2021},

    eprint={2103.06993},

    archivePrefix={arXiv},

    primaryClass={cs.RO},

    url={https://arxiv.org/abs/2103.06993},

    } [↑](#footnote-ref-31)
31. <https://github.com/osrf/soss> [↑](#footnote-ref-32)