POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Modular software architecture for assistive and rehabilitative exoskeletons: a proof of concept

Tesi di Laurea Magistrale in
Biomedical Engineering - Ingegneria Biomedica

Authors: **Dario Comini, Daniele d'Arenzo**

Student ID: 928348, 952883
Advisor: Prof. Marta Gandolla
Co-advisors: Stefano dalla Gasperina, Mattia Panzenbeck
Academic Year: 2021-22

Abstract

Exoskeletons are increasingly being used for rehabilitation and assistance of people with disability. Indeed, robots are designed in order to be patient-specific and to provide an effective treatment and support in daily activities. However, current exoskeleton designs are mostly all-in-one solutions, from low-level physical hardware, to high-level software functionalities. This is not ideal for continuously evolving scenarios like rehabilitation and assistance. Instead, it could be more convenient to individually and independently develop the different components of the system, choosing and connecting them afterwards through a common platform. For this purpose, a proof of concept for a modular software architecture for exoskeletons has been implemented, using ROS platform and a set of compatible and standard frameworks. To assess system modularity, the components of the architecture are individually and collectively tested, using a test-bench and BRIDGE – an upper-limb assistive exoskeleton. Results show that the architecture components are all capable of performing their function, regardless of the implementation choice. Adoption of standardized frameworks and modular design strategies can help researchers concentrating their expertise on tasks they are proficient in, allowing future solutions to become more robust, reusable and adaptable.

Keywords: modular, exoskeleton, disability, assistance, ROS2, robotic architecture

Abstract in lingua italiana

Gli esoscheletri sono sempre più utilizzati per la riabilitazione e l'assistenza di persone con disabilità: possono infatti essere progettati a misura del paziente, e al contempo fornire un trattamento e un supporto efficaci nelle attività quotidiane. Tuttavia, gli attuali esoscheletri sono per lo più progettati in un'unica soluzione fissa – dall'hardware fisico a basso livello, alle funzionalità software di alto livello. Questo non è ideale per scenari in continua evoluzione come la riabilitazione e l'assistenza. Potrebbe invece essere più conveniente sviluppare indipendentemente i diversi componenti del sistema, scegliendoli e collegandoli successivamente attraverso una piattaforma comune. In questo progetto è stato implementato un proof-of-concept per un'architettura software modulare per esoscheletri, utilizzando la piattaforma ROS e un insieme di framework compatibili e standard. Per valutare la modularità del sistema, i componenti dell'architettura sono stati testati sia indipendentemente che collettivamente, utilizzando un banco prova e BRIDGE – un esoscheletro assistivo per arto superiore. I risultati mostrano che i componenti dell'architettura sono tutti in grado di svolgere la propria funzione, indipendentemente dalla scelta implementativa. L'adozione di framework standardizzati e una strategia di progettazione modulare possono aiutare i ricercatori a concentrare le proprie competenze nei settori in cui sono più esperti, consentendo alle soluzioni future di risultare più robuste, riutilizzabili e adattabili.

Parole chiave: modularità, esoscheletri, disabilità, assistenza medica, ROS2, architettura robotica

Contents

1 | Introduction

1.1. Disability

Disability refers to the interaction between individuals with a health condition and personal and environmental factors [35]. Over 1 billion people are estimated to live with some form of disability, about 15% of the global population [36]. Moreover, it's correlated with ageing and chronic health conditions, resulting in a dramatic increase of people with disability [26].

People with disability experience poorer health outcomes, have less access to education and work opportunities and are more likely to live in poverty than those without a disability [40]. Often they do not receive the healthcare services they need or find them inadequate or under-resourced. This is due to **prohibitive costs** (over 50% of people with disability cannot afford healthcare [36]); **limited availability of services**, especially in rural or remote areas; **physical barriers** to access building or medical equipment; **inadequate skill and knowledge** of health workers, who sometimes don't meet their needs, treat them badly or deny them care [12]. Disability is extremely diverse. However, all people with disability have the same general healthcare needs as everyone else, and therefore need access to mainstream health care services [36].

There are two main interventions to contrast the impact of disability: **rehabilitation** and **assistance**.

1.1.1. Rehabilitation

Rehabilitation is a field that favors the development of the potential of people with disability. It consists of measures that range from physical interventions to improve body functions, to mental, social and vocational forms of aid, to strategies that promote inclusion and independence [32]. It involves the identification of a person's problems and needs, the related environment in which the person lives, the goals to be reached, planning and treatment processes and assessing the results [15].

Rehabilitation is always voluntary, but for some people with disability it is essential to participate in education, the labour market and civic life [36]. Moreover, better health is experienced when people with disability and their families are partners in rehabilitation [14]. An important factor is indeed **education**, not only of acquaintances, but also of all other people interacting with a person with disability, even on a general level.

Sometimes rehabilitation is distinguished from *habilitation* in that the latter is considered for disabilities acquired early in life or congenitally [36]. The rehabilitation process can be therefore divided according to the time passed since the disability is first experienced, as in the case of a traumatic event or injury, from acute and sub-acute to chronic [38].

Focusing on motor disabilities, the main goal of rehabilitation is the recovery of motor functionality. This is usually possible when the traumatic event has an orthopedic source, but becomes non-trivial when there is a neurological cause [3]. The effectiveness of a rehabilitation treatment and the consequent **motor relearning** depend on several factors [7][37][25]:

- **Repeatability** of the exercises
- **Intensity** and **dosage** of the training
- **Adaptation** of the treatment to the patient (needs, status, recovery stage), through **performance assessments**
- **Customization** of the therapy and **individualization**
- **Direct involvement** of the patient and motivation

The continuous increase in people with disabilities results in higher requests for rehabilitation. This in turn increases the burden for therapists and the healthcare system. Although the need of a specialist for the care of a patient is always needed, the workload and cost required for rehabilitation can be reduced through technological support.

1.1.2. Assistance

The purposes of **assistance** are different from those of rehabilitation, but not separate: both have the potential to improve health and quality of life of people with disability. However, while rehabilitation concentrates on the development of personal, physiological capabilities to contrast disability, assistance is mainly focused on providing support in activities of daily living (ADLs) [31]. This introduces another important topic in the context of disability: **autonomy**. The lack of assistive services can make people with disability overly dependent on family members and caregivers, causing economical and social inclusion difficulties.

Again, technology can come in handy in alleviating the issue of lacking autonomy. For example, a person with lower limb disability may require a wheelchair to be able to move independently. This is a fairly easy solution to the problem of mobility, but it requires to adapt the environment of people with this kind of disability to the developed solution, removing physical, economical and educational barriers: a collective effort is almost always required.

In the scope of assistance to motor and muscular disability, research and industrial effort has been mostly concentrated on powered wheelchairs or respiratory assistance: respiratory problems are indeed not only a matter of survival, but they can be both cause and effect of a wide variety of muscular and motor disabilities [42].
However, an important topic that is sometimes left aside and undervalued regards the disability and assistance of upper limbs. It has been shown that arm disability plays a key role in reducing patient's autonomy and worsening quality of life [17][10][22]. Also in this case, assistive technology has been demonstrated to significantly improve the ability to perform activity of daily living [17].

1.1.3. Robotic therapy

In both fields of rehabilitation and assistance, particularly for motor disabilities, rehabilitative and assistive robots are increasingly integrated into clinics [24] and industrially produced.

Robotic and mechatronic devices have been proposed to reinforce the process of rehabilitation on several levels. Indeed, robots can be designed in order to follow the aforementioned general rules for an effective rehabilitation (1.1.1): they should provide high repeatability and variety of exercises, high intensity and dosage, adaptation and customization of the treatment, high level of safety, possibility to assess performance during training, to provide task-oriented exercises and to increase motivation (for example through virtual-reality elements).

With a closely connected intent, assistive robotics aims at empowering people with motor disability to keep an effective motor functionality even in a domestic and personal environment, allowing them to be more independent from other people's assistance.

Surveys about stakeholders attitude toward robotic therapy confirm the good satisfaction levels and perceptions. This emphasizes the importance to continue improving such technologies, with the aim of maximizing the overall health status of people with motor

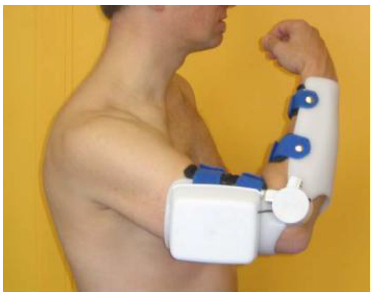


Figure 1.1: Elbow orthoses [18]

disabilities [11].

There are many classifications possible for rehabilitative and assistive robots. Also, literature is sometimes not consistent with these taxonomies [7]. A first general classification falls within the separation between **prostheses** and **orthoses** [18], depending on whether the device replaces an anatomical part, or helps it perform its function by interacting with it externally.

A remarkable type of robotic orthoses are **exoskeletons** (figure 1.2), devices that allow direct control of individual joints of a person's limbs, minimizing abnormal postures. This is possible because the robotic structure of exoskeletons interacts with the human body through multiple *ports*[7], generally represented by ergonomic cuffs.

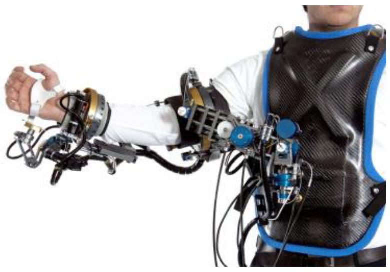


Figure 1.2: 16 DoFs X-Arm-2 exoskeleton [18]

By contrast, upper-limb assistive technology has been mostly concentrated to **end-effector** devices and **external manipulators**: in end-effector devices only the end-effector of the robotic structure is in contact with the human body; external manipulators are usually not even attached to user's arm [16]: this in fact can affect the dignity and self-esteem of people with upper-limb disability.

Differently, surveys confirm the great impression that active exoskeletons have made on patients and healthcare professionals [21]. In this case, people who wear the exoskeleton really are the ones performing the action with their own body, not an external object.

This tends to be more accepted: people with disability prefer to use their own skills and the abilities that they perfected and learned to overcome the disability itself. Deciding to use a surrogate device for a functional task is not always a pleasant choice, even if the function to be restored is necessary.

Robotic orthoses and exoskeletons are of particular interest also because of their complexity. The functioning of robotic orthoses and exoskeletons themselves is in a sense just a portion of the real objective of these devices: the correct functionality of the anatomical part that they have to support. In fact, exoskeletons constitute a complex inter-related system, that is often referred to as **human-robot system** (1.2).

One of the main problems in current exoskeleton implementations is that the designed solutions are often fixed, all-in-one systems with little room for change and adaptation, tailored to specific dysfunctions [13]. While this practice can still yield patient-specific solutions – an important requirement in the scenario of disability – it doesn't account for the evolution of patient needs and of the system at play, which can happen at any of the different elements it is constituted by: hardware requirements, control strategies, interaction mechanisms and more. This can also represent a problem when an already developed strategy must be implemented onto a new system, where incompatibility issues may appear, slowing down the development process.
Therefore, a **standardized** environment is needed to build all the different elements of a complex exoskeleton system on a common ground; on the other hand, a **modular** framework is required to merge all the system functionalities together ensuring they can communicate instead of going into conflict with each other.

1.2. Exoskeletons and human-robot systems

An exoskeleton system is a complex human-robot coupled system in which the wearer and its exoskeleton interact both physically (pHRI, physical human-robot interaction) and cognitively (cHRI, cognitive human-robot interaction) [18].

More specifically, to complete a functional task, the intention of movement from human perspective is elaborated at the Central Nervous System (CNS), which delivers motor commands to its actuating ports: the muscles. Physiological sensory systems (visual, somatosensory and kinematic) provide feedbacks that the CNS analyzes to adjust and correct the strategy, comparing the original intention and the executed movement.

The purpose of an exoskeleton is to replicate the kinematics and dynamics of human musculoskeletal structure and to support the human motion. Mimicking the human system, the exoskeleton system cooperates with the human by superimposing to the muscular effort the (external) robotic contribution (fig: 1.3).

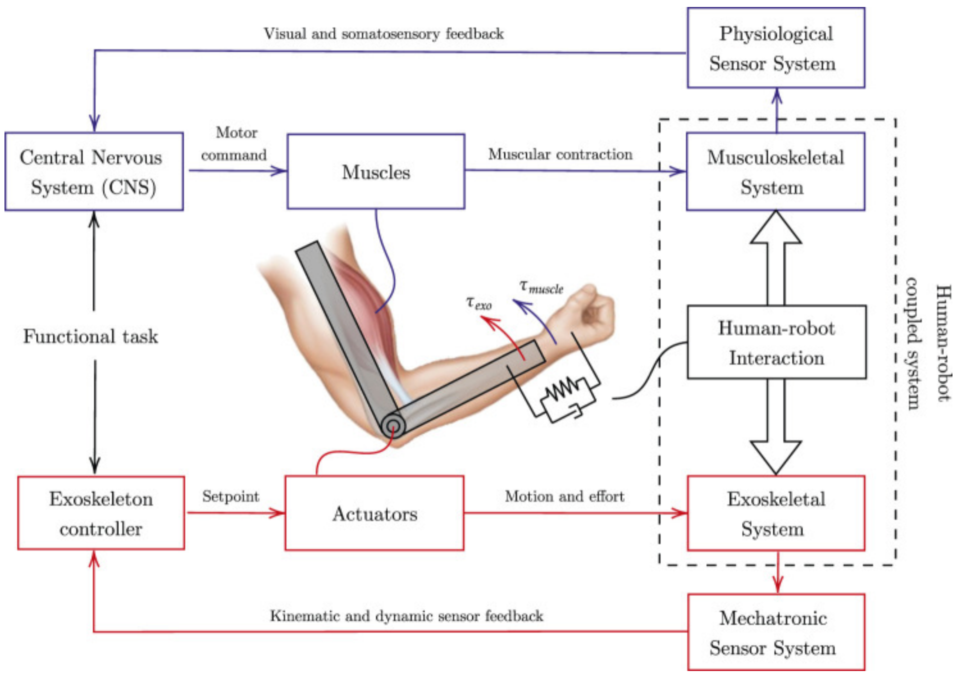


Figure 1.3: Human-robot interaction [7]

The interaction between exoskeleton and human subsystems is called, not by chance, **human-robot interaction**. It is important to understand that this interaction happens both at a mechanical level, exchanging forces and torques through the exoskeleton interaction *ports*, and more abstractly through the exchange of information between the two main subsystems. To understand the complexity of the overall system, we can start by classifying its **task environment** [39], which is an high-level model description:

- **Multi-agent**: Human and artificial systems interact and affect each other's decisions through competitive, complementary and cooperative information.
- **Partially observable** or **imperfect information**: the artificial and human agents are required to estimate information that cannot be retrieved by sensors (e.g. spatial localization, perception, state estimation).
- **Uncertain**: due to partial-observability, noise terms, stochastic environment and other uncertain or unmodeled elements.
- **Sequential**: past decisions may affect future ones.
- **Dynamic**: the environment may change while human and artificial agents are making decisions

- **Discrete** (artificial agent) and **continuous** (human and environment) systems.

This wide categorization of the system can already explain the difficulty in implementing an efficient exoskeleton agent, especially under the limitations of resources and cost typical of an assistive scenario. That's why usually the implementation of assistive exoskeletons is made through even rough simplifications, such as that of considering a completely passive control mechanism (1.2.2).

Robotic motion is often shaped abstractly through the concept of **degree of freedom** (DoF), which consists in all the independent directions in which the robot can move. The degrees of freedom of a robot define its spatial position, called spatial configuration or **pose**. If the first or second derivative of the robot pose is available, they define respectively the **kinematic** and **dynamic state** of the robot [39].

Non-rigid bodies as the human system possess additional degrees of freedom. Due to the complex anatomical structure, there is not a unanimous model available for the human body in bio-mechanics literature [19]. For example, the shoulder joint is usually modelled as a three degrees of freedom mechanism. However, the instantaneous center of rotation of the shoulder joint changes with movement of the human upper limbs, and it is required to accommodate this effect while modelling an exoskeleton shoulder mechanism. Joint axes misalignments can lead to high interaction forces and torques at the interacting ports of a human-robot system, as well as high cognitive load, contact pressure and discomfort [18]. Therefore it is important to minimize this effect by correctly assessing the specific bio-mechanics of an exoskeleton wearer while designing the artificial system.

There are many elements that compose an exoskeleton system, that will be described below. Their implementation depends on how much complexity is considered in the design phase. More importantly, a structure able to merge them all together into an unique system is needed: the advantage of a **modular** implementation is that - independently on the choice that will be made in the design process around these elements - they're granted to be all compatible together and customizable depending on the evolution of the scenario.

1.2.1. Exoskeleton hardware

The hardware of an exoskeleton system is mainly composed by actuators, sensors, passive components and transmissions.

Actuators are the active effectors of the system, and the means by which exoskeletons are able to autonomously move. The overall robotic device is often classified as **active** or

passive depending on whether it has at least one actuated degree of freedom [1].

Actuators themselves can be classified according to the nature of the energy source used to generate mechanical power [1]:

- **Electric**: 73% of actuators have an electric power source. In particular brushed and brushless DC motors are mainly used.
- **Pneumatic**: 13% of actuators use pneumatic forces from compressed gas.
- **Hydraulic**: 9% of actuators are hydraulically powered using pressurized fluids (figure 1.4.

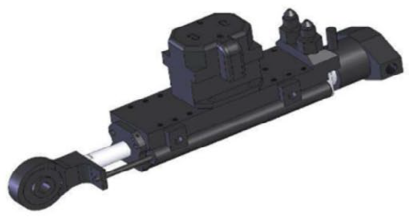


Figure 1.4: Hydraulic actuator [41]

Actuators can be placed directly at the exoskeleton joint, avoiding the need of a mechanical transmission, but increasing the inertia of the system and consequently its power consumption. By placing them proximally to the joint instead, a transmission mechanism is required. This may introduce nonlinear behaviours such as hysteresis and friction. Mechanical **transmissions** consists in cables, gearboxes, elastic bands and tendons, ball-screw drives, and other linkage mechanisms [18].

Sensors constitute the perceptual interface between the exoskeleton and its environment. There are both **passive** and **active** sensors, depending on whether they capture signals generated by other sources or they send energy into the environment.
Exoskeleton sensors are needed both for understanding the robot internal configuration and location, and for sensing the state and outputs of the human system: the artificial agent can use both these information to derive the state of the human-robot system and apply the correct decisions.

The main sensors used in exoskeleton systems are inertial IMU sensors and accelerometers, dynamic sensors such as force and torque sensors, position sensors such as encoders and goniometers, cameras for localization and object recognition, EMG and electrodes for recording muscular activity. Also, other sensors can be integrated to derive the intention of movement (1.2.3) or measure performances.

Both actuators and sensors are matter of discussion in a user-specific design [1]. Some sensors are notably expensive and cannot fit into an assistive scenario, but they are often a requirement in rehabilitation to implement a compliant control (1.2.2) and monitor a patient progress. Different users possess a different anatomical and functional bio-mechanics: this may require even very distinct kinematic chains, degrees of freedom, structural supports, materials and actuating hardware. Moreover, sensory systems requirements may be extremely diverse between individuals simply considering the intention-detection mechanisms (1.2.3). Also users' preference in choosing the solution that best suits them shouldn't be neglected. **Modularity** is therefore an useful feature, enabling to select the best choice among all the possible hardware combinations, while still allowing them to work together.

Sensors and actuators communicate with the artificial systems by means of mostly wired connections, adopting a specific **communication protocol**, which defines how two or more entities transmit meaningful information. According to the protocol, different speeds, distances and number of connecting devices are supported. The most common protocols are [33]:

- Inter System Protocols: USB and UART are very common protocols used to communicate between different devices through an inter bus system.
- Intra System Protocols: I2C, SPI and CAN are protocols used to communicate at lower level, typically between two devices within the same circuit board.
- PLC Communication Protocols: protocols used in PLC (programmable logic controller) includes Ethernet, USB and Profibus standards but used over a network.

The integration of the artificial system with actuators and sensors that use different communication protocols is another design challenge. This fragmentation may slow down the development process and even increase the costs of solutions: companies often prefer to buy components with the same protocols and manufacturers, so that they do not have to develop the control units from scratch. The high cost of many technologies is known to limit access for people with disabilities [36]. Again, a **modular** framework can help alleviating this issue, combining different technologies together while keeping intact the rest of the system.

1.2.2. Controllers

The term **control** refers to all the systems that regulate the behavior (output or state) of other systems. The agent that performs the control is called **controller**. A controller

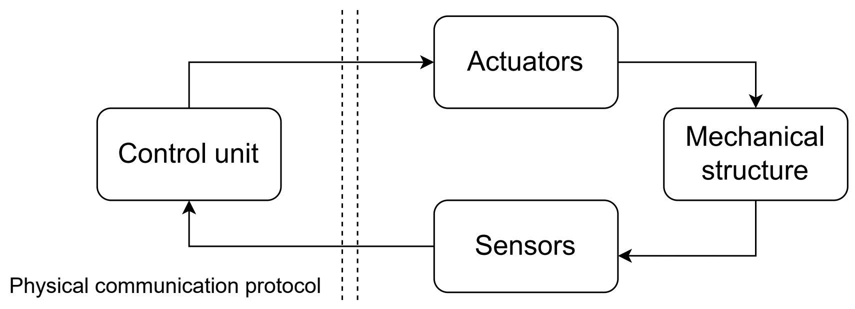


Figure 1.5: Communication protocol established between control unit of the artificial system and actuators/sensors of the human-robot system

doesn't always refer to a physical entity: it can be a more abstract agent given by the interaction of different subsystems. Moreover, controllers are sometimes regarded to as simple actuators controllers, whose goal is simply the driving of the robotic effectors; other times they're considered at an higher-level, including in their function also **perception** and **decision-making** mechanisms (1.2.5). Controllers represent one of the cores of a robot: they enable the system to compensate for errors and also act as an interface between physical hardware and higher-level functions, such as planning.

The general schema of a control system is a **closed-loop** model (fig: 1.6). The controller receives inputs from sensory data and a desired reference for the physical quantities that it has to control. In the scope of human-robot systems, these quantities can include state variables of the exoskeleton or human system, which can be hardly measured and are usually estimated from other sources of information; also, they can be the measurable output variables of the human-robot system. The controller then performs a transfer function, translating the input reference errors to control signals that are fed to the actuator system. A well-designed controller should be stable, responsive and able produce a control signal such that error is minimized and the real state of the controlled-system is as much as possible close to the desired one.

A further classification divides controllers into **feed-back** and **feed-forward** modalities (fig: 1.7), depending on whether they use or not sensory feedback information. Usually controllers are used in feed-back modality, because they can provide more stability and control over the system's variables. These are for example impedance and admittance control schemes [7]. However, if enough system's parameters are known and they're more or less static, the control can be made feed-forward, as in the case of some gravity-counterbalance or friction-compensation strategies [23].

On top of this general control scheme, several additional features can be added to achieve

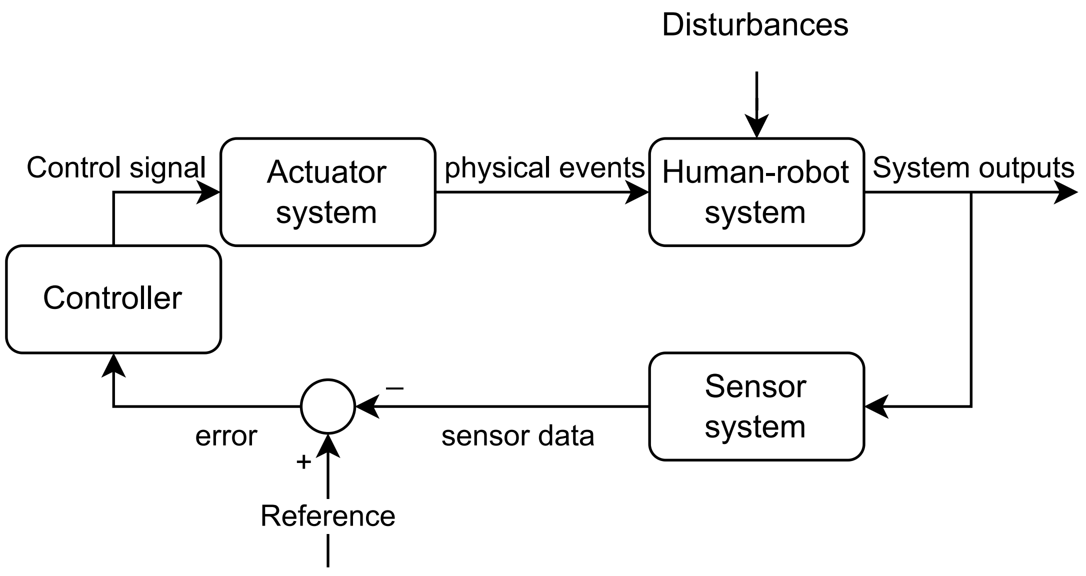


Figure 1.6: Robot control system

inter-joint coordination, promoting physiological synergies, or to adapt the assistance according to the patient conditions [7].

In an exoskeleton system, in particular in robotic rehabilitation scenarios, control systems are usually classified into hierarchical levels [7].

High level training modalities are control strategies that abstract the exoskeleton hardware, related to the desired human-robot interaction behavior. Depending on the level of assistance, they range from **passive** (movement is performed by the robot regardless of subject's response), to **active-assisted** (both robot and patient control the movement cooperatively, also called compliant interaction), **active** (movement arises from human contribution only) or **resistive** (the robot opposes the movement making it more difficult for the human).

Passive modalities are of particular importance also in the assistive field, as they can relax some of the constraints that rehabilitation has on the effectiveness of motor relearning and plasticity mechanisms, especially in the assistance to advanced disability. They can include:

- **triggered mode**: the wearer triggers the exoskeleton assistance. This encourages the patient to self-initiate movements, essential feature for motor relearning. The trigger is derived from intention-detection mechanisms (1.2.3).
- **teach-and-replay**: the exoskeleton pose and trajectory is recorded during a teach-

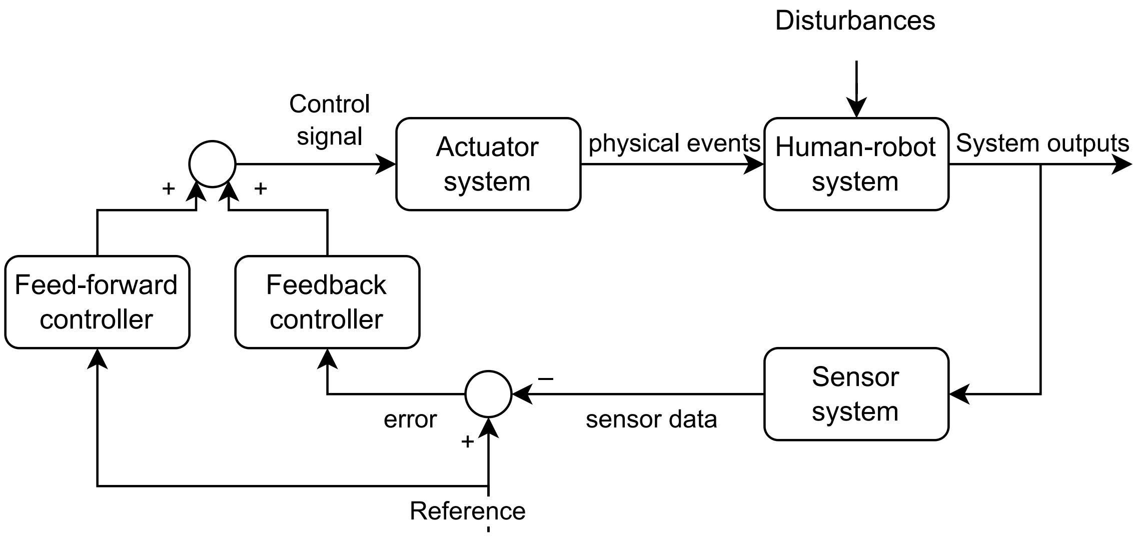


Figure 1.7: Feedback and feed-forward controllers

ing phase, in which it is operated in *transparent mode*. Then, the user can decide to replay the recorded trajectory.

- **mirrored**: the exoskeleton passively mimicks the behavior of an healthy limb, on which measures are performed in transparent mode by another exoskeleton. It's also called master-slave mode.

Low level control strategies are more related to the hardware implementation. They depend on output (sensing) and state (mechanical properties) of the system to be controlled, but also on the nature of system's inputs that the controller can drive. Their role is to shape and control the relationships between dynamic and kinematic variables, often modelled through a mechanical **impedance**, in Laplace's notation:

$$F(s) = Z(s) \cdot \dot{x}(s), \tag{1.1}$$

Where F is the force, Z is the impedance and $\dot{x}$ is the velocity. The impedance Z is generally approximated up to the second order:

$$Z(s) = C^{-1}s^{-1} + D + I \cdot s, \tag{1.2}$$

Where C is the **compliance**, D is the **damping** or viscosity, I is the **inertia**. The inertial term, dependent on the mass, is usually compensated in feed-forward, neglecting it from a feedback control as it is usually static (with some exceptions such as picking up objects).

Instead, a more important role is given by the compliance C. Compliance is the inverse of rigidity and models the relationship between a change in position dx and a change in force dF (i.e. spring model):

$$dF = C^{-1} \cdot dx, \tag{1.3}$$

In rehabilitation, a compliant control is needed to adjust how much force the subject has to exert in order to achieve a certain change in position. Usually, the subjects compliance is low not because of an intrinsic rigidity of its body, but because of an impairment that makes them weak and unable to apply a significant force. The total compliance (and impedance) of a human-robot system is given by the combination of the properties of the two systems. Therefore, by controlling the input variables of the human system (e.g. force and position) we are able to adapt the compliance to different patient's conditions, enabling them to move despite their weakness.

Most compliant controllers implement nested control loops, usually with an inner high-accuracy loop for fast-response and an outer "flexible" loop, which includes the human contribution and implements the interaction control. Such approaches mainly rely on two control schemes: **impedance control** (force/torque-based) and **admittance control** (position-based).

Impedance control uses an inner torque-feedback loop that promotes mechanical compliance and an outer position-feedback loop that corrects for trajectory errors by applying forces or torques. The torque (or force) signal τ is updated by varying a reference torque (or force) τ_{ref} around a function of the position error:

$$\tau = Z(s) \cdot (x_{ref} - x_m) + \tau_{ref} \tag{1.4}$$

Where $Z(s)$ is the mechanical impedance model, x_{ref} is the reference or equilibrium position (linear or angular) and x_m is the actual measured position.

Admittance control instead uses an inner position-feedback loop, which stiffens the control and an outer force/torque-feedback loop that softens the interaction behavior. With a dual structure with respect to impedance control, the position signal x is updated by varying a reference position x_{ref} around a function of the torque/force error:

$$x = A(s) \cdot (\tau_{ref} - \tau_m) + x_{ref} \tag{1.5}$$

Where $A(s)$ is the admittance model, τ_{ref} is the reference force/torque and τ_m is the actual measured force.

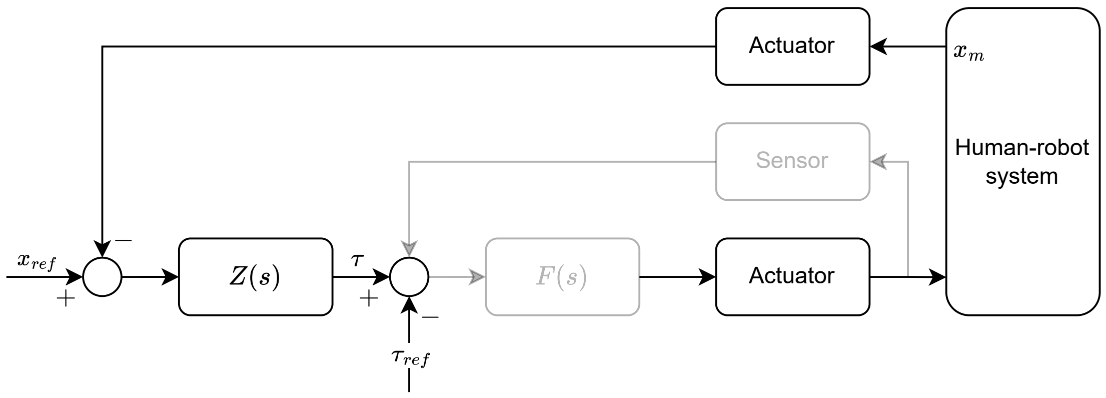


Figure 1.8: Impedance control. $Z(s)$ is the impedance controller, while $F(s)$ is the force/-torque controller of the inner loop, which can be neglected if the torque control is implemented through an open loop [7]

.

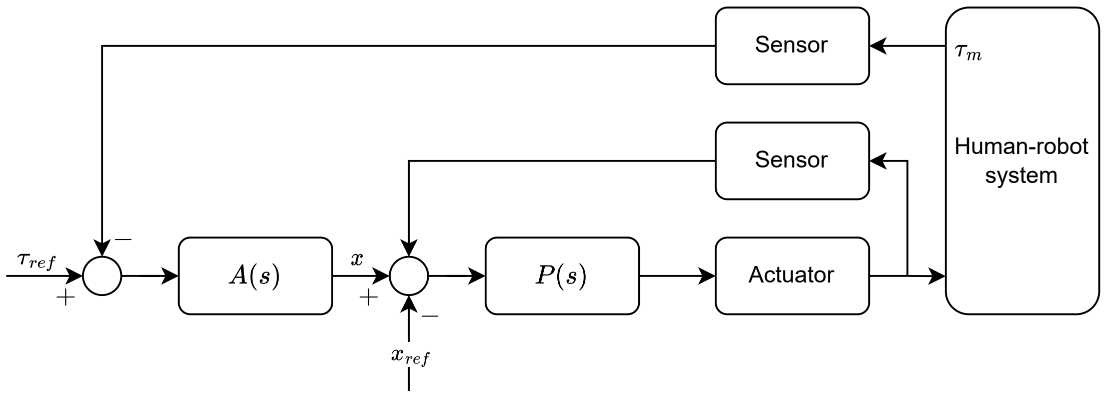


Figure 1.9: Admittance control. $A(s)$ is the admittance controller, while $P(s)$ is the position controller of the inner loop [7]

.

In rehabilitation, the compliance and in general the control schema and parameters of the system should be adapted to the patient's conditions. If rehabilitation is addressed to the recovery from a traumatic event for example, the acute phase is generally characterized by a passive modality: the patient has low residual force and the therapy is aimed at reducing muscular atrophy through the simple controlled movement of the body. In later stages, the patient is given more and more responsibility in performing the movement, and the robot should not interfere providing more than the required help. This is one of the important conditions to ensure an effective promotion of neural plasticity and motor recovery from the training [7].

In **assistive** scenarios, the focus shifts from motor plasticity, towards being able to perform a certain function autonomously. However, also assistive exoskeletons have to adapt to the patient's conditions, but for a different reason: to ensure that the functionality that the robot assists is still effectively and comfortably granted.

When the patient is severely impaired, usually compliance is kept close to zero, meaning total rigidity is perceived by the human system. Thus, the subject is not able to move whatever force it applies: it's completely controlled through the exoskeleton movement (passive modality). However, the subject can still control the movement, as the artificial system, through sensors, is able to detect human intention (1.2.3).
In other situations, people with less compromised motor disability may prefer to still be able to use their residual muscular force in order to move, with the necessity of a more complex control system, hardware and intention detection mechanism. This is often difficult to achieve in assistive scenarios, where costs and resources are limited. However, the connection between rehabilitation and assistance could be better exploited if both realities are able to follow the evolution of the user's condition cooperatively.

These situations represent again a reason why **modularity**, also at control level, is an auspicable property for an exoskeleton system. Designing an exoskeleton to be modular allows the control systems to work independently on the physical hardware located at a lower level, provided that the hardware is able to transmit and receive the physical information that the controller model requires.

1.2.3. Intention detection and user interfaces

Intention detection mechanisms are used by the artificial agent of the exoskeleton system to derive the willingness of the human system to perform a task. Intentions can be classified as **implicit** when they're derived from neural, muscular or force signals from the user, and **explicit** when the intention is expressly and voluntarily asserted by the user [28].

There is a wide variety of intention detection strategies:

- **Brain computer interfaces** (BCI): user's intention is derived from electrical and hemodynamic signals from the brain. Examples are electroencephalography (EEG), magnetoencephalography (MEG), which record the electrical activity of the brain; fMRI and NIRS, which rely on the measurement of task-induced blood oxygen level-dependent response (BOLD signal). These strategies need long training periods to translate the raw encoded brain signals to relevant decoded information about human intention. Moreover, they are often expensive, affected by background noise activity and poor information transfer rate. However, they're still in an early phase of research and development.

- **Muscle activation interfaces**: EMG-based interfaces are widely used because of their accessibility and direct correlation to the human intention. Indeed, they're one of the closest signals to the movement drive, while other signals may be delayed with respect to user's intention. The main requirements consist in signal processing mechanisms and proper filtering of external noise contributions and artifacts such as unrelated movements and tremors. Moreover, not all people with motor disability are able to generate isolated and repeatable contractions. Reliable muscle-activation patterns are indeed a requirement to be able to map the specific movement intentions.

- **Muscle-contraction interfaces**: muscular vibration, dimensional change, force, stiffness and hemodynamics can be all used to detect motion intention. The main advantage of these interfaces is that they're free from electromagnetic noise and they're a cheaper solution with respect to the strategies mentioned before.

- **Movement and force interfaces**: other implicit interfaces consist in sensing body segment's motion, through IMUs, camera-based systems, goniometers or encoders. Force-based interfaces make use of force and/or torque sensors by implementing control strategies that relate motion to the input force. One advantage is that these sensors can be embedded in the mechanical structure of the robot, avoiding the preparations needed in the previous strategies to position the sensors on the users. A disadvantage is that some people with severe muscular weakness may not have enough residual force to be detectable by these devices.

- **Explicit interfaces**: explicit interfaces are also called *parallel systems*, as they do not directly derive from the physiological pathway of a movement execution. These interfaces are extremely diverse, some examples include eye interfaces (movement or gaze-trackers); tongue interfaces through hall-effect and pressure sensors; head interfaces that detect head movement, direction or inclination; speech and voice-recognition interfaces using conventional microphones and language-processing algorithms; hand interfaces, such as joysticks.

Each of these methods present several advantages and disadvantages. Recent approaches are directed toward fusing data from multiple interfaces and sensors, in order to derive **hybrid** intention-detection interfaces. These solutions can combine advantages of different solutions, while solving the individual limitations that they possess, in a cooperative or hierarchical manner. Hybrid interfaces also have the potential to better adapt to the needs of a user, giving them more control over the preferred interfaces, more accuracy and reliability.

Another important topic is the **feedback** that the exoskeleton gives to the user. For active exoskeletons, this consists at least in a force feedback, due to the movement of the robot itself. Other two main feedbacks are used in this context: **haptic** and **visual**. haptic feedback generates forces to interact with the user through the sense of touch. Visual feedbacks instead consist in visual interfaces and virtual environments used to support user's proprioception and sense of localization of the human-robot system [1].

Concerning intention detection and user-interfaces, **modularity** can both allow to select the most desirable interface and to effectively separate the desired functionality from the interface itself. This allows for example to easily fuse together multiple intention-detection mechanisms, obtaining the valuable advantages that result from an hybrid interface, as discussed above.
Assistive devices and assistive technologies are often incompatible with smartphones, the Internet and other mainstream systems that people with disability are often used to. To overcome this problem, it is required that the device is adaptable to a wide range of user capabilities and compatible with a wide range of user interface devices. In this sense, a standardization of the system allows to keep up with the rapid evolution of interface technologies.

1.2.4. Planning

The classical **planning** refers to what to do, and in what order, without any information about time, neither how long an action takes to be executed, nor when it occurs. The real world also imposes many resource constraints that must be considered into planning activity. Time-parameterization of a path is called **scheduling**, that is the process of adding temporal information to the plan to ensure that it meets resource and deadline constraints. The overall problem [39] can be divided into a planning phase in which actions are selected accounting for constraints and to meet the goals of the problem, and a later scheduling phase, in which actions are time-parameterized according to deadline constraints.

Working space and its representation is a core point to implement the point-to-point motion problem, so to deliver the robot or its end effector to a designated target location. The space of robot states called **configuration space** is defined by location, orientation, and joint angles and is a better place to work than the original 3D space which is computationally expensive to deal with. The path planning problem then, concern the action of find a path from one configuration to another in the robot configuration space and

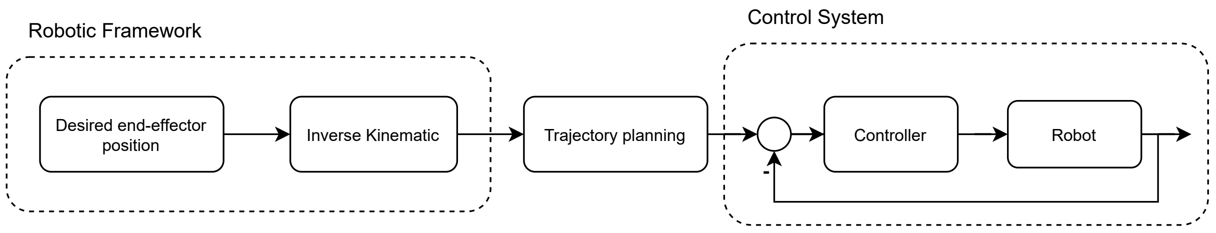


Figure 1.10: Generic planning integrated into a robotic system

simplifying the robot in **simpler structures** with techniques such as cell decomposition, which decompose the space of all configurations into finitely many cells, or skeletonization technique, which project configuration spaces onto lower-dimensional manifolds.

Even with the simplification of using the robot configuration space, the task of a robot is usually expressed in 3D workspace coordinates, requiring mapping techniques to relate workspace coordinates and configuration space. These chain of transformations are linear for prismatic joints and trigonometric for revolute joints and are known as **direct kinematics**.

The inverse problem of calculating the configuration of a robot whose effector location is specified in workspace coordinates is known as **inverse kinematics**. Calculating the inverse kinematics is expensive, especially for robots with many DOFs because the solution usually is not unique.

Also accounting for obstacles is not simple: one usually probes a configuration space instead of constructing it explicitly, so the planner generate a configuration and then test to see if it is in free space by applying the robot kinematics, then it checks for collisions in workspace coordinates. In a space high densely with obstacles (the problem is greater if obstacles have complex shapes) replanning due to collisions may require some time to complete.

Planners are integrated into high-level frameworks so user can not be aware of planning processes. Robotic frameworks (1.2.6) include visualization, user interaction tools and simulations that integrate perfectly with planning workflow: in a simple task (figure 1.10), user move virtually the end-effector in a 3D space, its workspace coordinates are then converted into configuration space coordinates with inverse kinematics and sent to trajectory planner as a goal state. Once the trajectory has been calculated, planner communicates with controller which will take care of executing the commands.

In medical applications, trajectory planning helps to improve the activities of daily living of stroke patients who will not have to control directly the exoskeleton for the entire trajectory. Once defined an acceptable amount of everyday positions in the workspace, these can be saved and recalled quickly by the patient. However, it is difficult to accurately replicate human kinematics with robots due to the morphologic variability of patients and the complexity of joint kinematics; this "kinematic incompatibility" could lead to hyperstaticity or overconstraint. In literature [27], using more DoFs into the configuration of an exoskeleton has been indicated to improve kinematic compatibility. Biomedical tasks use frequently 2 type of approaches[34]:

- Cartesian motion planning and inverse kinematics using **polynomial-interpolation** or computational methods such as minimum-jerk, minimum-torque-change, and inertia-like models. These methods don't have high computational costs, but generated trajectories are not physiological. Optimization-based methods with human-like motions can be used only for simple point-to-point movements with a limited range of motion, so, not really usable for activities of daily living.
- **learning by demonstration** using learning models such as neural networks, Gaussian mixture models, and DMP models. They show more physiological trajectories but have high computational costs.

Motion planning is a very time-demanding task, depending on tasks and user residual abilities a planner choice must maximize the performances: for this purpose a platform which gives the possibility of a fast planner switch is preferred rather than solutions which require code intervention when adapting to a new patient.

1.2.5. Decision system

The artificial "brain" of a robotic system consists in a **decision module**, able to translate the information interpreted from sensors into one or more actions that the robot has to perform, in order to fulfil its goals. The interpretation of sensory data is called **perception**: it is a kind of state estimation which enables the artificial agent to convert sensor data into higher-level representations of information, suitable for the robot decision system.

Agent decision systems can be classified according to the complexity of decision-making [39]:

- **Simple reflex agents** are stateless controllers with no memory of past perceptions; the inputs trigger some *condition-action rules*, which are simple if-then rules that can be a-priori given or learned through machine-learning processes.
- **Model-based reflex agents** are still reflex agents, but with additional structures

that constitute memory. These agents maintain a model of the environment, but the action is still completely determined by just the current perceptions and model.

- **Goal-based agents** try to predict an action by thinking about what the state will be, exploiting knowledge of actions consequences and perceptions of the environment. They can select a goal and decide which path to follow in order to pursue it.
- **Utility-based agents** are goal-based agents in which the goal is not binary (reached or not reached). Instead, it has an utility which provides the degree of "goodness" to the different possible actions.

A complex artificial agent such as an exoskeleton system is composed by many different decision systems.
Some low-level modules may require fast and real-time decisions. These can be for example safety mechanisms to prevent failure of the different hardware or software components, fault-tolerant structures able to compensate a malfunctioning element, units to enforce safety limits for the movement of the exoskeleton and so on.
Higher-level components on the other hand may be allowed to perform more complex and time-costly decisions, such as sensory perception tasks, localization, planning of trajectories and adaptation of the exoskeleton parameters depending on the evaluation of user's performances.

There is not an unique way to organize and interlace the different elements of an exoskeleton decision system. Some applications in assistive scenarios may rely more on a lightweight and fast-paced structure, focusing on direct, real-time controls and simplified calculations. A rehabilitation context, instead, may lean more towards high-level functionalities able to pursue physiological complexities and reinforce the motor recovery.

Usually these functionalities are performed on robotic platforms (1.2.6). Their collective organization and structural arrangement is what constitutes the system's **architecture** (2).

1.2.6. Robotic platform

Control unit integrates controllers and high-level frameworks which have to communicate each other, for this reason a **messaging protocol** must be implemented with a set of rules, formats, and functions for exchange messages with data integrity. An application protocols is evaluated according to performance metrics such as packet loss rate, message size, bandwidth consumption and latency [43].

Despite the large number and type of protocols, there are two distinct categories of frameworks that are relevant to automation purposes:

- **low level** framework messaging protocols (figure 1.11), including ZeroMQ, DDS and LCM. A basic messaging protocol provide service layers in each message with functionalities that are protocol purpose-based. A real-time communication system must include a timestamp or an integrity check layer.
- **high level** frameworks, also called **robotic platforms**. Most of them are **middleware**; a middleware is a software that act as intermediary among two services or applications and lies between an operating system and the applications itself. Among them, ROS, PX4, ArduPilot and NVidia Isaac SDK are mentioned because of their market spread. Usually a middleware is much more that just a messaging protocol and includes more stuff such as visualizations, transforms graphs, dynamic configuration, integration with simulations and much more.

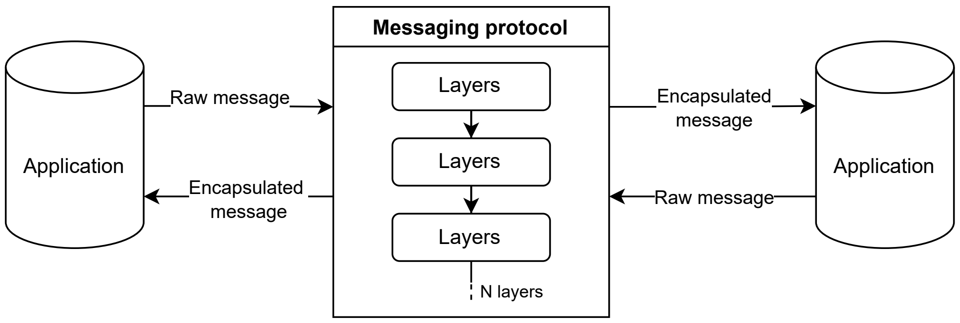


Figure 1.11: Two applications exchange data using a message protocol.

From a biomedical perspective, middleware are frameworks of great interest thanks to their support to hardware implementation and visualization, which is a very common tool in a rehabilitation process both for the patient and the specialist. Furthermore, medical monitoring applications prefer protocols whose goal is real-time data exchanging due to high frequency of physiological data collected from patient. Most of middleware have also the capability to ensure system stability even if a process dies due to errors. Application processes must be identified in a well-known state, usually active/inactive/running, that prevent errors of a single process influences the execution of the others; In an exoskeleton scenario, a joint fail must not stop the execution of counterbalance processes, because it can cause hardware breakdown and consequently harm to patients.

The adoption of a **standardized** robotic platform is a key element in enabling all the parts of a **modular** design to be compatible, as it constitutes a common ground for development in the robotic field. Moreover, standardization carries with it the benefits of a continuously reviewed and updated infrastructure, such as security measures, feature improvements, testing protocols, all aimed at reducing risks and constantly improving the system.

1.3. Modularity and standardization

No one model of support service will work in all contexts and meet all needs. Person-centred services are preferable, so that individuals are involved in decisions about the support they receive and have maximum control over their lives [36]. One of the philosophies that leans toward pursuing this objective is the **user-centered design** [20] or *co-design*. The inclusion of people with disabilities in the design process, alongside caregivers and multidisciplinary expertise, is what mostly allows to remain close to the needs, sensations and objectives of each user. Self-assessment is an important part of this process: it is not always easy for service users to articulate their needs, so supported decision making should be indicated.

Given the impressive complexity of human-robot systems for rehabilitation and assistance previously described, the unfeasibility of an unique solution suitable for everyone becomes easily clear. At the same time, methods and technologies to support disabilities may improve more rapidly if collective effort is directed toward the cooperation and the interchange of a wide variety of studied solutions, described on a common reference ground. In this view, a strong importance should be dedicated to **modularity** and **standardization**.

Adaptability and modularity of the different elements of a robotic system is an extremely important factor. Modularity can help improve all stages of an exoskeleton life-cycle:

- **Design phase**: a modular framework and way of thinking can help the development of technologies, dividing the overall problem into simpler blocks that are easily connectable, replaceable and isolated from the rest of the complex overall system. The simplification of this process can help not only the user, but also the clinicians and developers to understand the different components and functionalities that the system could offer, and allows to make decisions more easily.
- **Context of use**: the utilization of robotic technology by the end-users can also benefit from modularity and standardization, allowing them to rely on a safe, robust and peer-reviewed infrastructure, which was designed to satisfy as much as possible their requirements.
- **Update process**: a modular framework is intrinsically easily customizable and

adaptable to the evolution of user's needs, without requiring to change all the structure. Developers can switch to different variants of the system according to the will of the user without too much effort.

In robotic rehabilitation, automatically adjusting the device support based on the patient progressive recovery is essential [1] and requires high interchangeability at all levels of the human-robot system. The same is true for a device which is used by multiple different people, as it happens in the rehabilitative scenario.
Nonetheless, exoskeletons used for assistance of people with disability should be enough customizable and adaptable to meet the needs of each individual, at least during the design process.

The purpose of this work is to provide a proof of concept for the construction of a modular and adaptable software architecture, in particular addressed to upper-limb assistive exoskeletons. Even if the proposed solution was implemented for this specific area, the general concepts and way of thinking are not limited to it. We believe that, with a careful evaluation of the trade-offs, this work can be extended to other fields of application aiming at the contrast of disability.

2 | Materials

2.1. Software architecture requirements

One of the cores of a **modular architecture** is the software: while the hardware (actuator, sensors and the physical structure) of an exoskeleton can be made customizable and modular through engineering effort, a software application enables much more degrees of flexibility. Moreover, most of the algorithmic and high-level elements that compose an exoskeleton system do not come in pre-built physical boards, but have to be designed and implemented through software programming. Even if they did, the adaptability given by software is simply unmatched.

A **software architecture** is a methodology to structure, organize and assemble all the algorithms, entities and functionalities that compose a system. It also includes languages, tools and the overall philosophy for how programs can be brought together [39]. Software architectures are often hierarchical, contain multiple elements running in parallel but also sequential instructions, are usually designed in layers of abstraction, networks and through multiple techniques supporting and complementing each-other.

As already introduced for the decision system of an artificial agent (1.2.5), a software architecture, from an overall decision-making perspective, may contain low-level and high-level functionalities. Low-level functionalities are called **reactive**: they're sensory-driven and often real-time, but they rarely yield plausible solutions at global level. High-level functionalities are called **deliberative**: they are model-based, depend on information elaborated through perception and are relatively slow-paced. Most software architectures for robotic applications are focused on optimally fuse the two principles, to combine their strengths while reducing their weaknesses.

A software architecture for robots interacting with people, especially if they have health conditions, has many requirements (1.2), one of which is its adaptability to the user's needs.

First, it has to be able to provide an **abstract description** of the physical structure of the exoskeleton system: this includes the kinematic chain, the dynamic properties of the physical elements (mass, inertia...), the shapes and measures of the various links (3D model), the degrees of freedom and the physical limits and constraints. An abstract description is important to have an unique global and easily adaptable description of the physical components. Each other element of the architecture should be able to derive all the important information that it needs about this description by interpreting and parsing it in its own way.

A second requirement is the possibility to connect **hardware** with different characteristics. To make this independent from the overall system, the low-level hardware instructions must pass through a layer of **hardware abstraction**: this means that, whatever the hardware is, the software will be able to provide abstract instructions that will be understood by the physical hardware, thanks to the decoding function offered by the hardware abstraction, and vice-versa.

From the **control** perspective, the architecture needs to be able to switch between different control modalities (1.2.2), especially if the exoskeleton will be used in a rehabilitative scenario. This includes not only simple close-loop and open-loop structures, but also the nested, hierarchical and other complex control schemes which are continuously developed by research. Also, controllers must be abstract entities able to switch between different control commands, such as position, current, force, without caring about how the hardware implements them. The hardware abstraction or some other component could care about decoding this information. Both hardware abstraction and control functions should be as possible **reactive** and real-time implementations.

The deliberative part of the architecture must include **planning** functionalities (1.2.4), in particular motion planning and trajectory execution. Planning must be highly customizable in order for it to be user-specific and respect the physiological bio-mechanics required for a safe movement execution. Also other mechanisms of control of movement should be implementable, such as direct control of end-effector, mechanisms to avoid collisions, singularity and abnormal position avoidance and similar.

The architecture must be able to implement several different **user interfaces** and **intention - detection** mechanisms (1.2.3) by abstracting the interface hardware from the functionalities that it must implement. Different functionalities and interfaces must be easily convertible and possibly fused together. Also, the software should be able to provide feedback to the user, whether haptic or visual through a virtual environment.

The architecture should be able to run all these components independently, in parallel and asynchronously, possibly including high-level parent-entities that regulate the synchrony

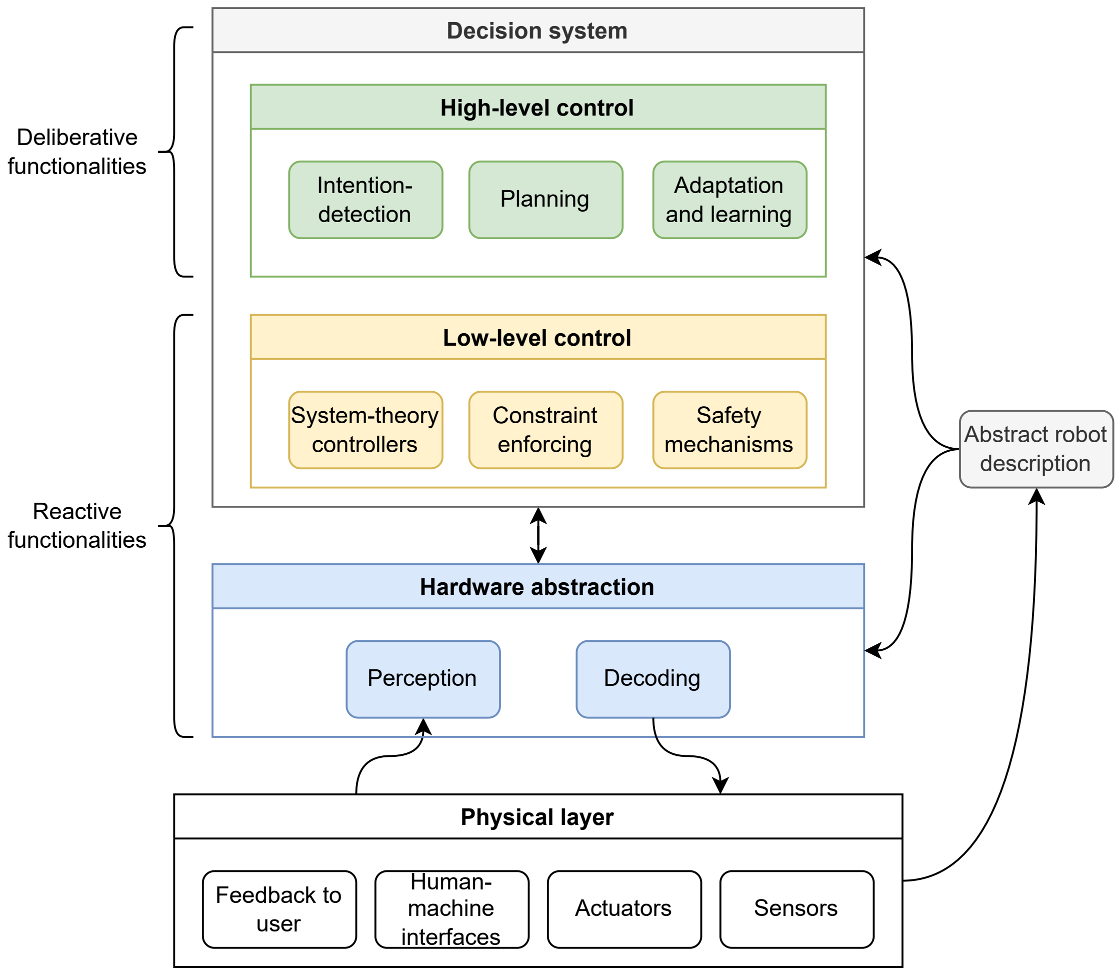


Figure 2.1: Robotic system architecture design

and the behavior of the multiple different processes. There should be an high possibility of designing the network of connections between components, even dynamically. Also, most components should be parameterized and dynamically adaptable.

All these components should preferably include both packages already implemented and **standardized** from a common database, and the possibility to construct one's own implementation through a software template reference. For accessibility, the architecture should also be compatible with other systems and languages commonly used such as Matlab, Simulink, Labview, Python and so on. Finally, it is preferable for the software to be independent from the physical machine it will run on, by abstracting the operating system.

2.2. Selection of the robotic platform

The high-complexity of the system required to have an high modularity, made it necessary to choose a robust high level framework rather than a simple messaging protocol such as DDS. Moreover, a robotic platform (1.2.6) in the form of a middleware integrates itself a messaging protocol and, beyond this, also other essential tools - such as visualizations, transforms graphs, dynamic configuration, simulation environments - which help in the development of modular software.

Among all the alternatives such as ROS, ArduPilot, NVidia Isaac SDK or PX4, the first one was chosen: it guaranteed the greatest compatibility and is the most widespread possible.

ROS advantages over competitors are:

- an easy to use, cross-language **inter-process communication** system that is fairly versatile (works via IP address or shared memory)
- it allows easy integration of a wide range of **tools** including visualization of robot kinematics and sensor data, path planning and perception algorithms, as well as low level drivers for commonly used sensors.
- Management tools that allow **monitoring and inspecting** messages.
- the **messaging system** approach encourages a high-level of modularity when writing code, although, a drawback is its overhead, which can accumulate in complex systems consisting of hundreds of processes
- open source

2.2.1. ROS 2

ROS (Robot Operating system) is an **open-source** collection of libraries, drivers and other tools for building robot applications [30]. It is a relatively recent project, and it contains many of the main **standards** accepted and used by the robotic communities all around the world. ROS 2 is the most recent version of this software, and it is increasingly being adopted in both academic and industrial environments.

Although ROS is in core a middleware, it provides much more functionality and tools than a middleware such as hardware abstraction, inter-process communication mechanisms and package management. Despite its name, it's a meta-operating system (meta-OS), meaning it runs on top of an host OS.

The conceptual architecture of ROS is composed by a peer-to-peer network of elements communicating one another: ROS is designed to be modular, and robots constructed through ROS consists of many elements separating resources and functionalities.

ROS Nodes: Each single module of the ROS infrastructure constitutes a node. Nodes are processes that perform computation, derived from a common standard library, but highly customizable. All nodes can be uniquely identified by the rest of the system, and their configuration state is given by dynamically reconfigurable **parameters**.

Standard nodes follow a **life-cycle** state-machine that governs their functionality. There are 4 main states:

- **Unconfigured**: the node becomes unconfigured immediately after it's instantiated and initialized.
- **Inactive**: represents a node that is currently not performing any processing. The main purpose is to allow a node to be (re-)configured without altering its behavior while running.
- **Active**: main state of node life-cycle. The node performs processing, responds to service requests, reads data, produces output and so on.
- **Finalized**: states in which the node is immediately before being destroyed, used for debugging and introspection.

The transition between states are made possible by **supervisory processes**, unless an error occurs; figure 2.2 shows the various transition states that can occur in a standard node.

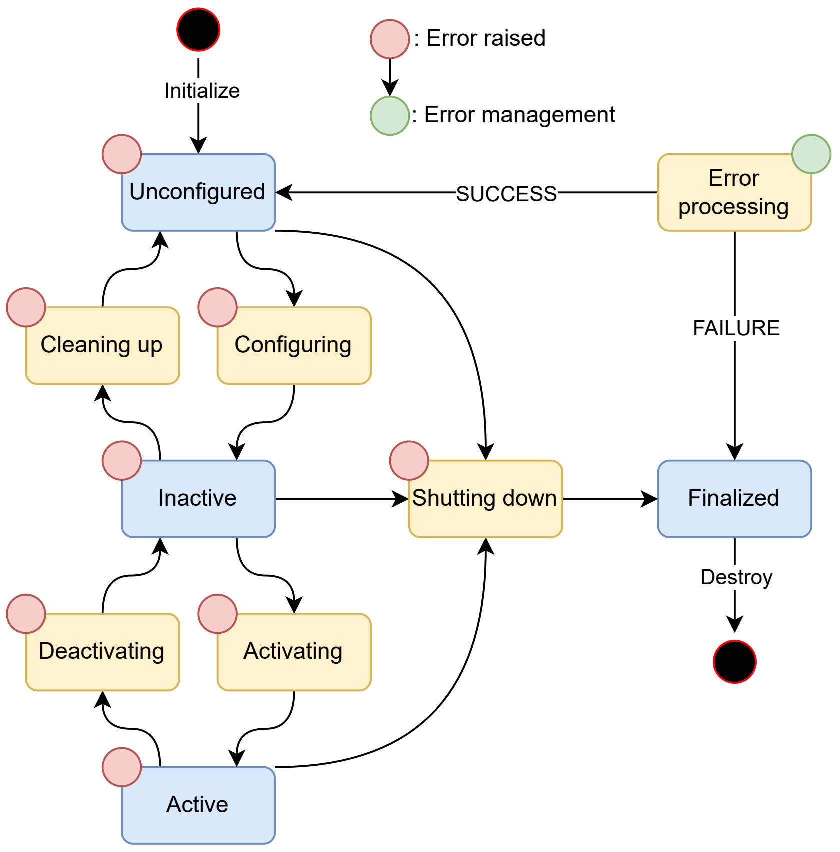


Figure 2.2: ROS node standard life-cycle

ROS interfaces: Nodes communicate with one another through ROS interfaces. All interfaces work by exchanging **messages**; the semantics of the information contained in a message is defined by the message type. Interfaces are usually not real-time, but they possess a series of **communication policies** that can be customized (for example retrying sending messages, deadlines, queuing...). There are three main types of ROS interfaces:

- **Topics**: Topics act as a bus for nodes to exchange messages. Communication via topics is N-to-N through a **publisher-subscriber** model: nodes sending messages are publishers for that topic, while nodes receiving messages are subscribers (fig: 2.3).
- **Services**: Services are another method of communication for nodes. Services are based on a **request-response** model: they only provide data when they are specifically called by a client (fig: 2.4).
- **Actions**: Actions are yet another communication type intended for long-running tasks. Actions consist of a goal, a feedback and a result. The goal is sent by a client

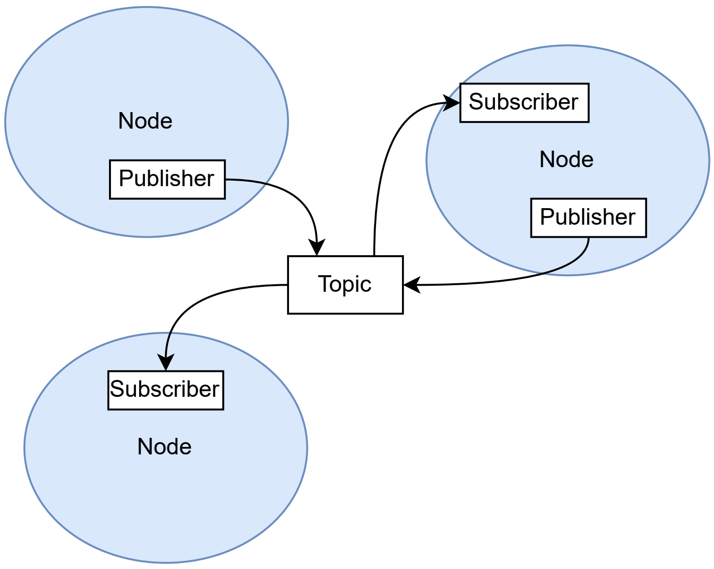


Figure 2.3: ROS2 topics

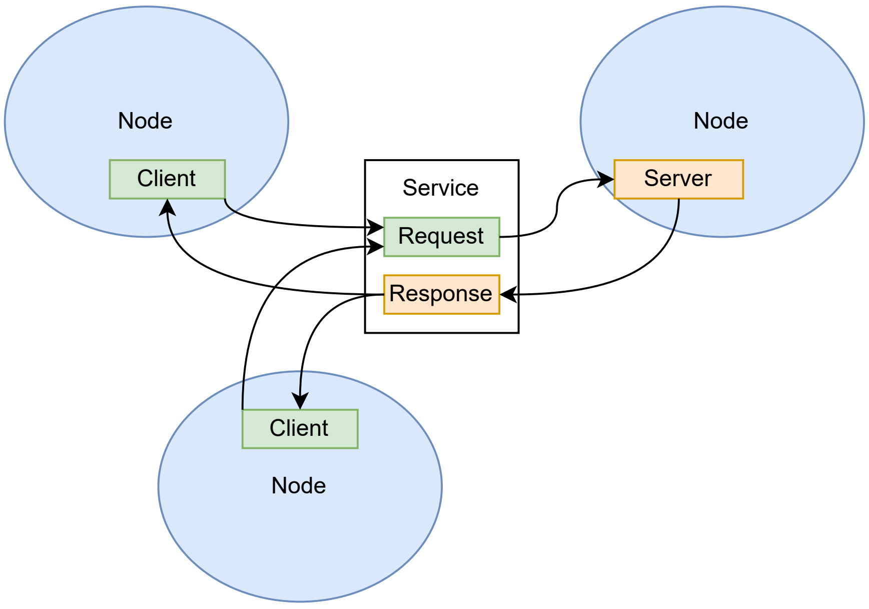


Figure 2.4: ROS2 services

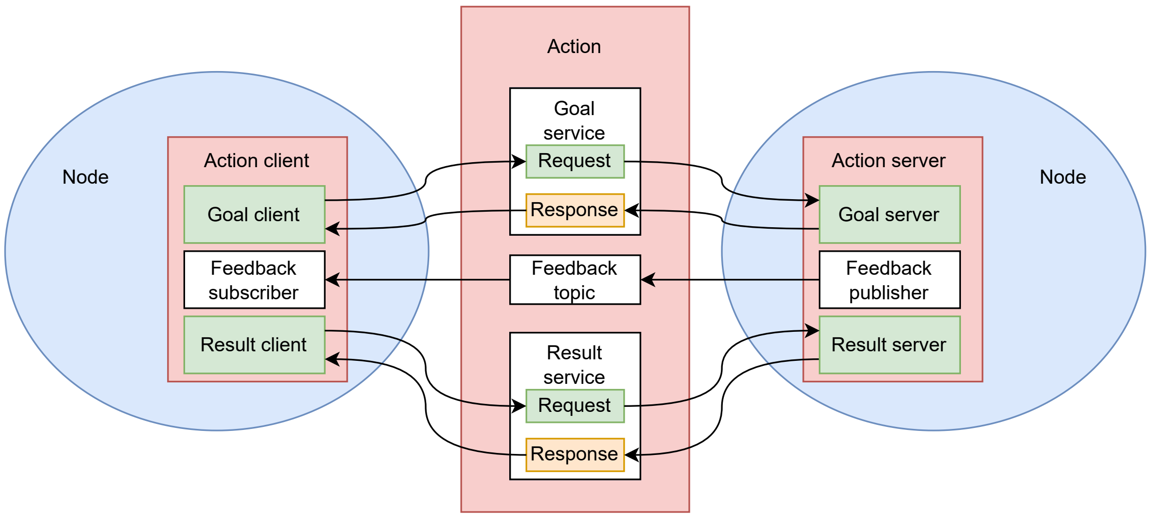


Figure 2.5: ROS2 actions

node to be fulfilled by a server node. While executing the task, the server sends back a stream of feedback messages to the client. When the goal is reached, or the action is interrupted, the server sends a message to the client containing the result (fig: 2.5).

On top of this architecture, several libraries and packages are constructed. Table 2.1 shows a concise list of the aformentioned architecture requirements, with the corresponding implementation alternatives given by ROS. Thanks to the underlying platform, these elements can all be developed independently and successively fused together, assuring modular functionalities.

Requirement	**ROS2 implementation**
Multi-processing and task independence	Intrinsic in the design as a network of nodes, constituting different processes running in parallel
Customization	Low-level customization thanks to programming and code-based implementation; high-level customization given by node-network parameterization
Hardware and Exoskeleton abstract description	Implemented as a markup language (URDF and SRDF) and additional parametric files
Hardware abstraction	Provided with real-time safe functionalities and resource management by the **ROS2 control** library
Control functionalities	Implemented with real-time safe functionalities by the **ROS2 control** library
Planning and other movement-control functionalities	Supported through the compatibility with **MOVEIT2** framework and Navigation2 stack
Graphical interfaces and visualization	Implemented by the **RViz** module, Gazebo physical simulator and Qt-based applications

Table 2.1: Software architecture requirements: implementation through ROS 2

2.3. Robot description

As previously mentioned, an abstract robotic description allows to generalize the physical model of the robot, providing semantic information that can be interpreted by all layers of the system, or even by different robotic platforms. Hence, it ensures that the individual elements of the architecture are independent from the robot model, making the architecture modular.

The logical components of the robotic system are arranged using the Unified Robotic Description Format (**URDF**). This allows to provide a **kinematic** and **dynamic** description of the robot, **visual representations** and **collision models**. URDF is a markup language (XML) that is shared across the ROS2 system and subsequently parsed by the different nodes. These information are not strictly static: once initialized, it is also possible to change them at run-time if the structure of the robotic system varies. The drawback is that only tree structures with rigid links can be represented, ruling out parallel robots and flexible elements.

The description of a robot consists of a set of link elements and a set of joint elements (fig. 2.6). Links represent the skeleton of the robot. The link element describes a rigid body with an inertia, visual features and collision properties. The joint element describes the kinematics and dynamics of the joint, also specifying safety limits of the joint. Links and joints coordinates are always expressed with respect to parent reference frame.

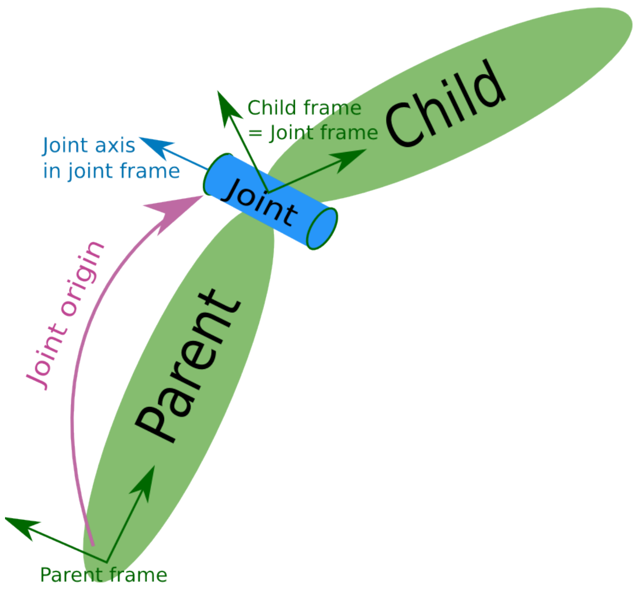


Figure 2.6: URDF hierarchy with a joint connecting parent and child links; child frame coordinates are with respect to parent frame

To write an URDF description, an "higher-order" markup language can be used: **XACRO**. XACRO allows to define macros so that the XML source files become simpler. Macros are then expanded before instantiating the source as URDF.

2.4. Hardware abstraction layer

The lowest level of the conceptual architecture, just before the physical layer, is the **hardware abstraction layer**. It has core importance in the hardware modularity, as it acts as a transition system between hardware and software, assuring their compatibility by encoding and decoding information. Moreover, being at low-level, the implementations

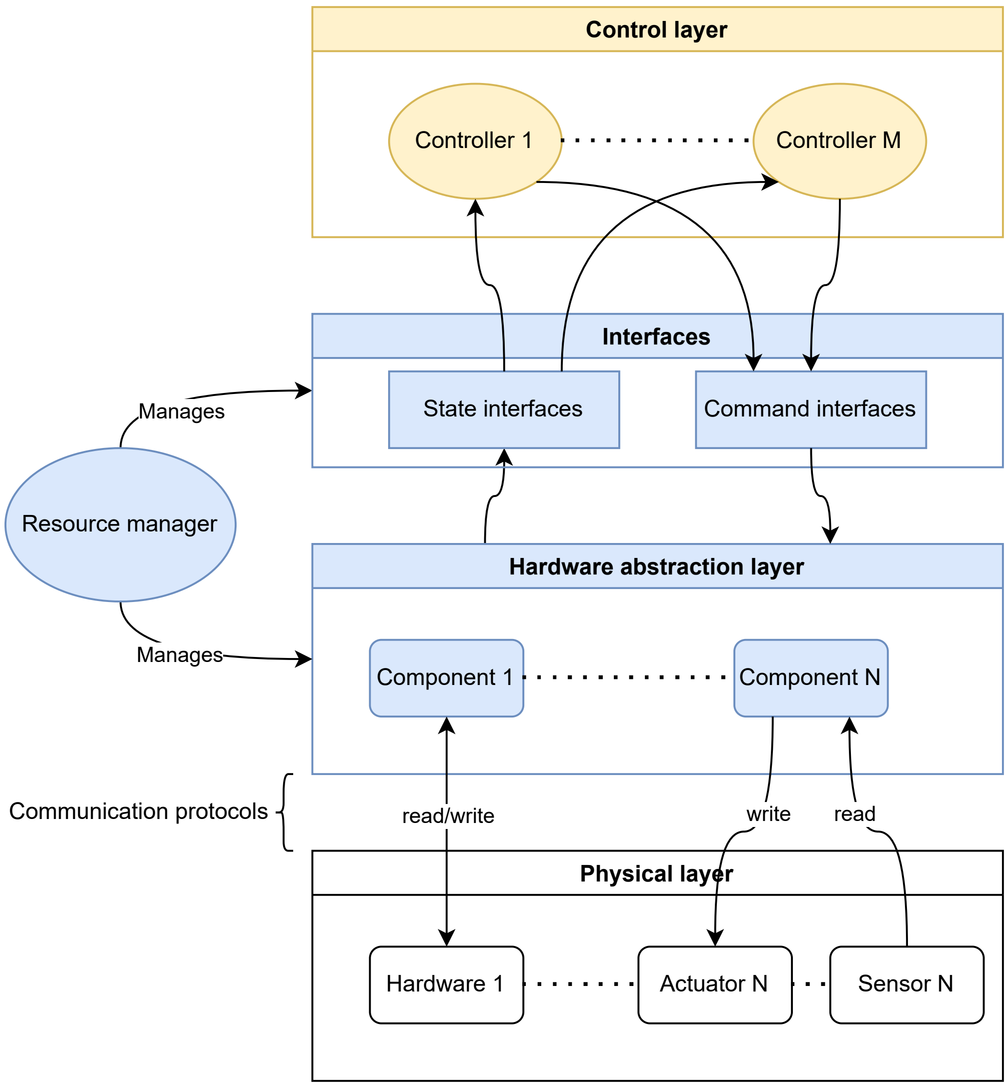


Figure 2.7: Hardware abstraction operates encoding/decoding functions between controllers and physical layer

are constrained to real-time requirements.

In ROS2, the hardware abstraction can be implemented by the **ROS2 control** package. The hardware abstraction layer is composed by **hardware resources** or **components**. Each hardware component acts as an intermediate system between controllers and physical hardware, translating abstract information at the control level to hardware directives at the physical hardware level, and vice-versa (fig: 2.7).

We can distinguish **logical components**, that represent the different parts of a robotic model (e.g. joints, links) or more abstract structures, and **physical components**, that are the actual hardware implementation of the robot (e.g. actuators, sensors and transmissions). The information at the hardware–controller interfaces is related to logical

components. For example, an input command for an hardware component may refer to a desired joint state. It is then responsibility of the hardware abstraction to send the proper signals to the actuators in order to achieve that joint state, considering reductions, coupling of multiple actuators and complex transmissions.
Similarly, information coming from different physical sensors can be pre-processed and combined by the hardware abstraction, then sent as a logical joint "state" information to the controllers' interfaces.

There are three basic types of hardware components:

- **System**: a system component represents a complex robotic hardware abstraction (e.g. multi-DOF). It can expose several interfaces to the controllers and use complex transmissions.
- **Actuator**: an actuator component represents a simpler robotic abstraction, like single motors, valves or similar. It can still be used with a multi-DOF robotic system if its hardware enables a modular design (i.e. multiple components with multiple communication channels to the physical hardware).
- **Sensor**: a sensor component abstracts the robotic sensor system. Sensors are related to logical components, such as joints (e.g. an encoder) or links (e.g. a force sensor). Unlike system and actuator components, it has only reading capabilities.

Hardware components can be dynamically loaded and unloaded at run-time. In particular, they follow the **life-cycle** common to ROS2 nodes (2.2.1). The main phases are:

- **Initialization phase**: the main attributes are initialized, robot description is parsed from URDF to derive logical components and transmissions.
- **Configuration phase**: connection with physical drivers is established and the local attributes are shared with controllers as hardware–controller interfaces (2.4.1).
- **Activation phase**: consists of a final, quick preparation before starting executing the main component activity.
- **Execution phase**: hardware components cycle between **writing** commands to the hardware and **reading** back the hardware state from sensors, driven by the requests of controllers.

The loading/unloading of components and their life-cycle is managed by the **Resource Manager**, a central entity that governs the hardware abstraction functioning, ensuring that hardware components requests do not collide with each other, distributing the resources and managing their access.

Each component has usually a single communication channel with the physical hardware, but the actual implementation depends on the application and on the machine that is used to run ROS2. Typical communication protocols for a personal computer are USB serial or Ethernet, but any means of communication fast enough to keep up with the required control cycle may be implemented, if the hardware supports it.

2.4.1. Hardware–Controller interfaces

The communication between hardware abstraction and controllers can be bi-directional, M-to-N, and is made through default (e.g. "position", "velocity", "effort"...) or custom **hardware–controller interfaces** (fig: 2.7). More specifically, each hardware component exposes ('exports') its interfaces, while each controller claims the interfaces it requires.

There are two main types of hardware–controller interfaces:

- **Command interfaces**: representing information going from controllers to the hardware abstraction (hardware input).
- **State interfaces**: representing information going from hardware abstraction to controllers (feedback).

These specialized interfaces are **real-time** channels of communication, implemented as a shared memory between hardware abstraction and controllers. The hardware abstraction initializes these variables as component attributes, then their address in memory is shared with controllers. Because of that, the access to interfaces needs to be correctly managed in order to avoid colliding retrieval of resources. Again, this is handled by the Resource Manager.

2.5. Control layer

To ensure modularity and adaptation of the control modality, especially required in rehabilitation, the control implementation should also be system-independent, at least as much as the control model allows it. This can again be accomplished through the **ROS2 control** package.

Controllers in ROS2 have the same functionalities as defined in control theory. One one side, they communicate with the hardware abstraction through command and state interfaces (2.4.1), gathering feedback states from hardware and outputting the driving commands. On the other side, they share information with the rest of the ROS2 system through the standard interfaces (2.2.1), receiving the *desired* hardware states and com-

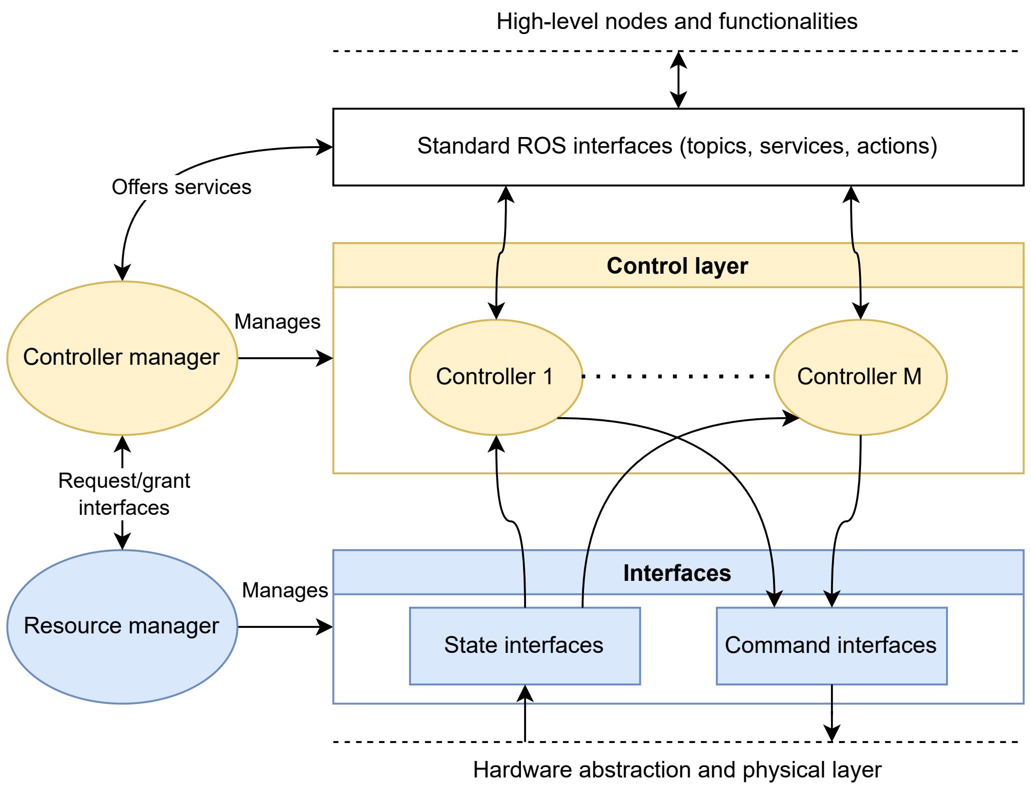


Figure 2.8: Control layer operates between high-level functionalities and hardware abstraction layer

municating the *actual* states to other ROS entities (fig: 2.8). Internally, they perform the operations that implement the required transfer function (e.g. PID control) between inputs and outputs.

Similarly to hardware components, controllers are dynamic objects whose life-cycle is managed by a central node, the **Controller Manager**. The life-cycle is similar to that of hardware components, but the execution phase consists in the **update** of the controller state, which performs the transfer function between input and outputs.
The Controller Manager communicates with the Resource Manager to claim the interfaces that the controllers request, in order to access the hardware abstraction without conflicts. It also offers ROS2 services (2.2.1) for the user, that can be exploited to directly manage controllers and their activity. The Controller Manager also synchronizes the functioning of the different controllers by maintaining a unique clock that determines the control-loop update frequency.

The current implementation of controllers in ROS2 requires that the actual real-time channel of communication is only that between controllers and hardware abstraction. However, this doesn't allow to construct the nested architectures used in rehabilitation for a compliant control, as it requires controllers communicating one another. This is a downside that is currently under development. Up to now, to implement nested architectures they have to be all included in a single high-level controller, or the real-time requirement has to be relaxed by using standard ROS2 interfaces for communication between controllers.

2.6. Planning layer

Planning functionalities are useful in all those situations where the intention of the user is more abstract, like moving from one point to another, leaving many possibilities on how to achieve that. It is also needed when the user executes some frequent or repeated tasks that he does not want to (or he is not able to) control directly. In this case, it is not the user himself - but the artificial system - that thinks and decides the course of actions to perform a certain task. Being an high-level function, planning has an even more wide variety of possibilities on how to implement it, from trajectory interpolation methods, to collision-avoidance strategies and environmental constraints, to kinematic algorithms, to inter-joint cooperation for the execution of more natural movements.

ROS2 fits perfectly the planning prerequisites, because with its open interfaces it can integrate any kind of planning framework, with high customization capabilities. However, since ROS2 is compatible with **MoveIt2** out of the box, its choice is almost obvious by users because it doesn't require adaptation interventions. Moreover, MoveIt2 is the most widely used software for manipulation and has been used on over 150 robots [4]. It is released under the terms of the BSD license, and thus free for industrial, commercial, and research use; this makes it easy to be adopted by as many people as possible. MoveIt is also capable of managing control movement mechanisms, such as direct control of end-effector, mechanisms to avoid collisions, singularities and abnormal position avoidance and similar.

MoveIt2 is a robotic manipulation platform for ROS 2 led by PickNik Robotics [4], which runs on top of ROS2. MoveIt framework runs on top of ROS building on its messaging and build systems and using ROS pre-existing tools like the ROS Visualizer (Rviz) and the ROS unified robot description format (URDF). Its functionalities includes:

- **Motion Planning**: generation of trajectories avoiding local minima. A specific C++ interface helps to integrate easily a custom planner.
- **Manipulation**: interaction with local environment with grasps.
- **Inverse Kinematics**: extraction of joint parameters from end-effector pose, even

in over-actuated arms.

- **Control**: possibility to execute time-parameterized joint trajectories using motion planning adapters and to fix start/goal state if outside joint limits bound, near to singularities or in collision with an object.
- **3D Perception**: handling different kinds of sensor input, MoveIt is able to generate point clouds (octomaps) and depth images.
- **Collision Checking**: detection and deviation of obstables using octomaps or mashed geometric primitives.

MoveIt2 architecture is simple (fig. 2.9): a **central node** can be chosen from a series of default ones, each of them with peculiar features. For example, motion planning is implemented by a node called **move group** [2.6.3] while direct control by the node **MoveIt servo** [2.6.5]. Each central node shares the ability to access the Scene, which is a representation of environment and its constraints. The central node can also communicate with hardware using controller interface. Human machine interfaces are implemented with ROS visualizer plugins, custom implementation with C++ or raw input from a computer terminal.

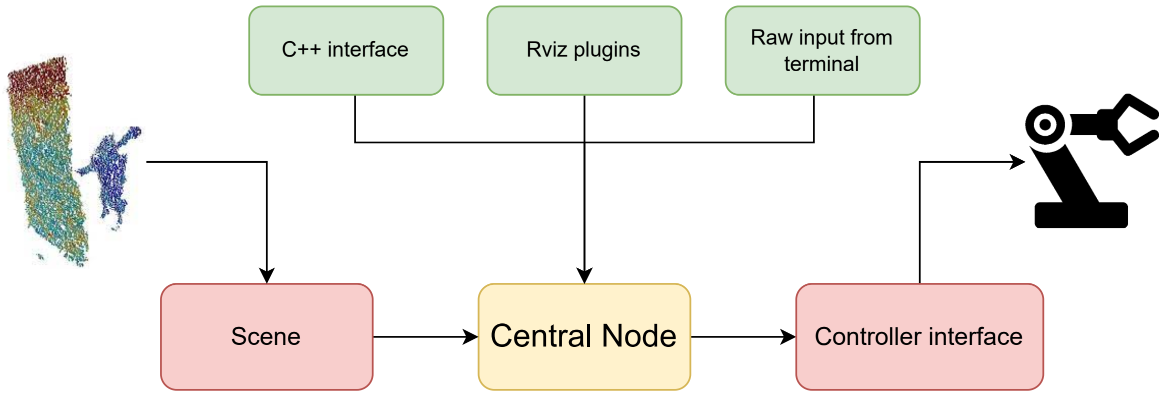


Figure 2.9: MoveIt architecture; functionalities change switching central node

Despite the possibility of using ROS2 topics and services for communication between nodes, MoveIt must be aware of external packages used to extend its functionalities: human hardware interfaces, planners and collision detection systems are implemented as **plugins**, giving a further modularity to the architecture.

Central node parameters can be customized with configuration files, so every detail of a particular functionality can be customized and easily switched with another configuration set.

2.6.1. Planners

Moveit2 architecture allows to fast switch between multiple planning plugins or to use different planners for each joint group. Currently available planners include Open Motion Planning Library (OMPL), Open Motion Planning Library, Stochastic Trajectory Optimization for Motion Planning (STOMP), Covariant Hamiltonian Optimization for Motion Planning (CHOMP).

Default plugin used is **OMPL** [44] (fig. 2.10), an open source C++ implementation of more than **40 different sampling-based** algorithms. A sampling-based motion planning approximates the connectivity of the search space with a graph and so reasons over a finite set of configurations in the state space.

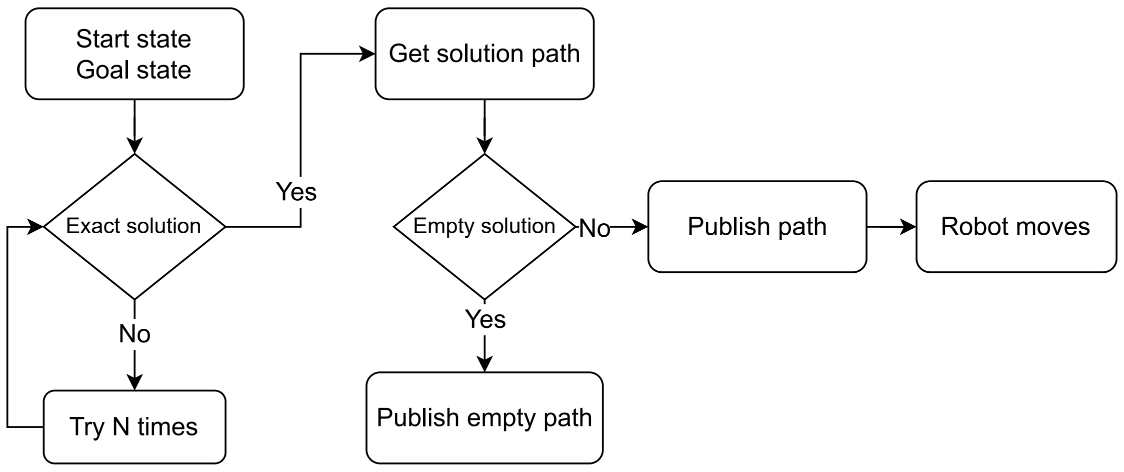


Figure 2.10: OMPL functional scheme

Within OMPL planners are divided into two categories:

- **Geometric planners**: they accounts only for the geometric and kinematic constraints of the system.
- **Control-based planners**: used when the system under consideration is subject to differential constraints which restrict the allowable velocities at each point. In robotics most problems involves differential constraints due to dynamics of robots.

To set up a planning context in OMPL are required:

- **Space**: is expressed as simple state space for geometric planners or a representation of controls for control-based planners. OMPL indeed, is flexible and applicable to a wide variety of robotic systems, so the library does not explicitly represent the geometry of the workspace or the robot operating in it.

- **Space Information**: needed to check the validity of state and motion
- **Problem Definition**: necessary to set initial and goal states
- **Planner**: planner class with parameters

OMPL itself makes sure that all the objects are properly created before the planning operation begins.

OMPL choose planner in accordance with constraints and set parameters: for example, if the state space has a default projection (e.g. joints projection), then OMPL will use LBKPIECE. In other circumstances RRTConnect can be used. These two planners, in particular, have been shown to work well consistently across many real-world motion planning problems, which is why they are the default choice.

2.6.2. Kinematics

There are both analytical and numerical solutions for Kinematics [2] which are available to be integrated in MoveIt2 without any particular effort. An **analytical solution** is a direct calculation of the joint values, while a **numeric solution** is often an optimization problem (for example start in a known position and apply gradient until a solution is found). Analytical solutions are therefore faster and often generate solutions when numeric methods fail, but they are very hard to calculate for complex manipulators.

The most commonly used numerical IK implementation in the robotics community today is the joint-limit-constrained pseudoinverse Jacobian solver found in the Orocos Kinematics and Dynamics Library [2].

Nevertheless, KDL's inverse kinematics implementation has the following issues:

- if the robot has joint limits, the convergence can fail.
- no alternatives are available if the search gets stuck in local minima
- tolerances can't be used in the KDL solver

A KDL alternative, which has been adopted more and more in recent years, is **IKFast** that can analytically solve the kinematics equations of any complex kinematics chain with solutions extremely stable that can run as fast as 5 microseconds on recent processors.

2.6.3. Offline planning

In a typical static scenario, planning is performed before the movement execution: this modality is known as **offline planning**. Offline planning can be implemented with a central node called Move group, a key node provided by MoveIt2 that act as a centralized point to connect user actions and core components to let plan and execute trajectories in an easy way. In particular, using URDF, SRDF and other Moveit configuration files, it is able to:

- Communicate with robot using ROS topics and actions to get joint state information or 3D perception from sensor data. Move group listens on a topic for determining the current state information and on a transformation topic to monitor transform information (i.e. pose of the robot).
- Send commands to controllers using the **controller interface**, which is a ROS action interface.
- Using the **Planning Scene Monitor**, can monitor the planning scene, a representation of environment and robot constraints, including attached objects (fig. 2.11).

Move group exposes a service interface to receive motion plan requests through 3 inputs: Rviz plugins that make use of MoveIt2 C++ interface and standard QT libraries, an alternative is a C++ requests without the use of a GUI (motion plan integration with others high-level tasks); the last input type is through user's terminal input thanks to MoveIt2 API (green blocks in fig. 2.11). The motion plan request includes planner preference and parameters (otherwise defaults are used), motion constraints and final pose of end-effector or new position in joint space.

Once motion plan request is received, move group works with custom or default planners through the **motion planning plugin**, making MoveIt2 easily extensible (blue block in fig. 2.11). The motion plan result is not just a path from start to desired location; move group builds a chain including path and adapters which result in a **trajectory**, that is a path bound by specific, time constraints, joints velocity and acceleration for each way-point.

Motion plan adapters allow for pre-processing motion plan requests and post-processing motion plan responses, depending on the motion constraints and environment characteristics. Some adapters are useful for fix starting position if the robot is slightly outside the specified joint limits, others can time-parameterize trajectories.

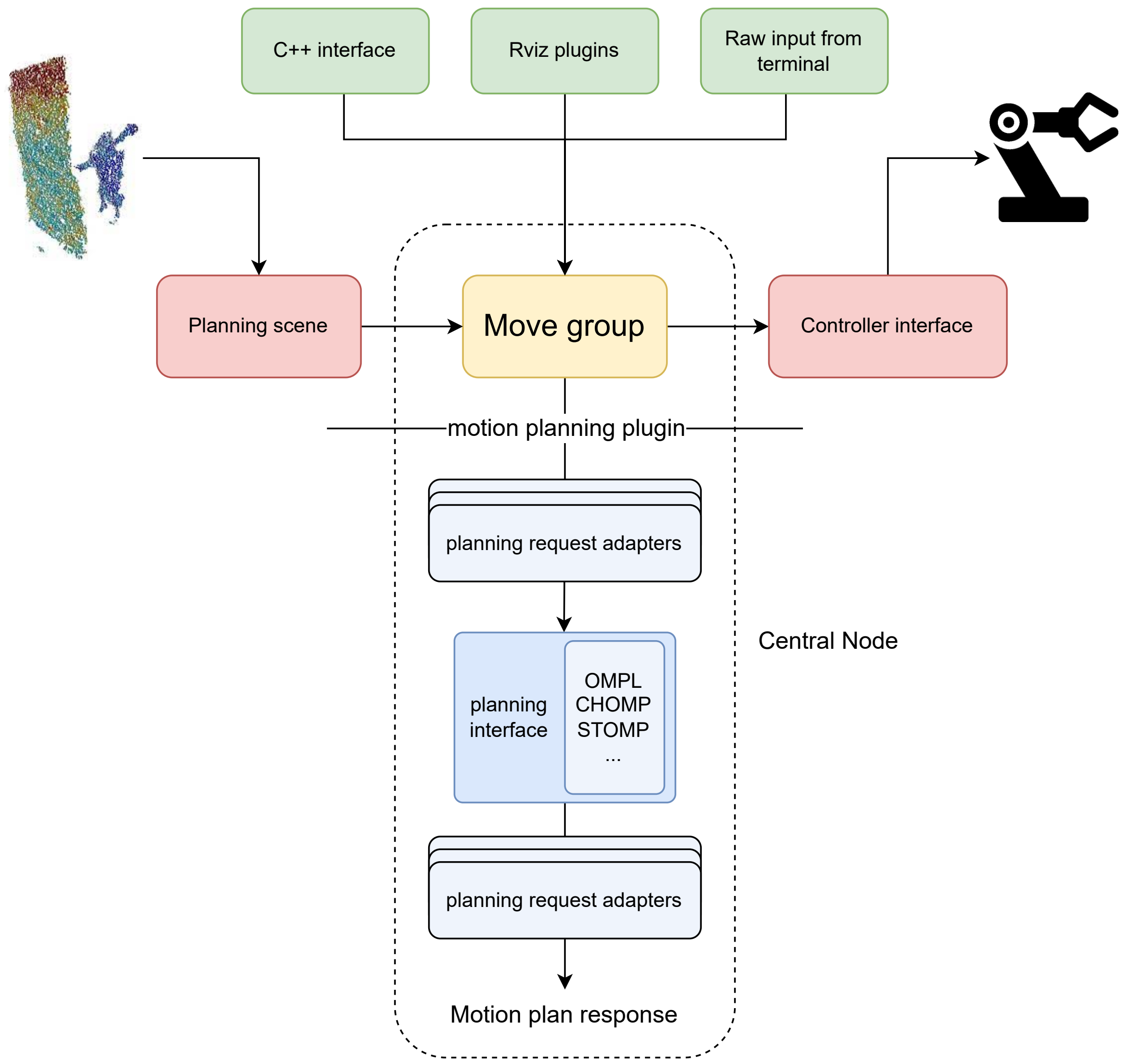


Figure 2.11: Motion plan workflow

2.6.4. Online planning

Offline planning works well in well-known static environments but not in unstable or dynamic situations. Industrial tasks may involve moving object during trajectory execution like other robot arms, workers, conveyor belts; therefore, is necessary to check real-time for collisions. From a biomedical perspective, tasks like writing and manipulating objects require adapting pressures and forces, but even simply moving in a dynamic environment where other people interact with the user requires real-time adaptation. The **Hybrid Planning** architecture attempts to solve this problem by combining a pair of global and local planners that run in parallel and recurrently with different planning speeds and problem scopes.

Hybrid planning manager receives the motion plan request from API, C++ interface or Rviz plugins and then coordinates the planners asking global planner to publish a new solution trajectory when local planner identify a local constraint. Planner logic plugin defines how to react to events and is highly customizable (2.12).

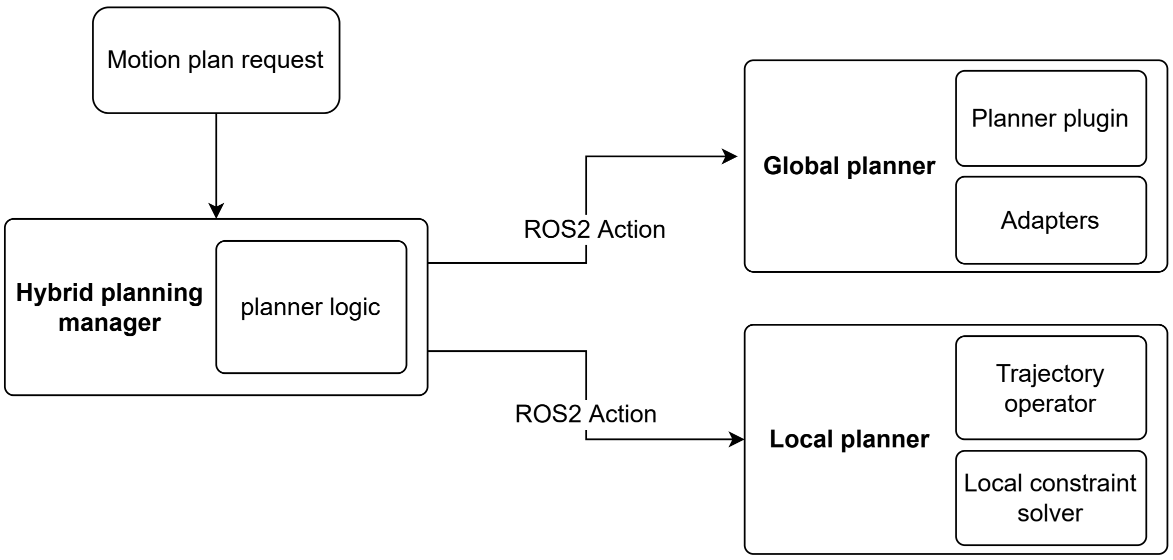


Figure 2.12: Hybrid planning architecture

The **global planner** runs slow and, by default, it's the move group pipeline with defined planner plugin and adapters, that build a trajectory made of way-points (single trajectory points in the joints space). Its target is to publish the solution trajectory to the Local Planner for further processing; indeed, once published, the trajectory is managed by the local planner trajectory operator, which extract way-points that local constraint solver has to test, for example, to avoid collisions.

The **local planner** is running at high frequency during movement execution, because local constraint solver has to be sufficiently sensible to constraint detection. If, during an iteration, a local constraint is identified, local planner sends a feedback to the planning manager and activates the planner logic. If no constraint is detected, commands from the current way-point are sent to the robot controller and a `Success` feedback is notified to planning manager.

2.6.5. Direct control

Direct control of the exoskeleton can be implemented with MoveIt Servo central node, which is a node that allow to stream end-effector velocity commands from a wide range of inputs, including joysticks, voice commands, keyboards and every hardware-mapped system. The main feature that distinguishes this node from a simple robot direct control, is the capability of preserving self collision check and singularity avoidance, inherited from core functionalities of MoveIt2.

Servo keep an high modularity by sharing libraries with other core nodes, indeed a cooperative approach lets developers instantiate Servo and move group nodes together, allowing the user to reach a pre-defined location with planning, but also control directly end-effector for precision movements, although the two nodes cannot be active at the same time.

The two primary inputs to MoveIt Servo are Cartesian commands and joint commands (fig. 2.13):

- End-effector **twist commands** are expressed as velocity in free space broken into its linear and angular parts (x-y-z linear, x-y-z angular) referencing to a specific frame, declared in the header of the message. A twist message linked to a body reference frame is in agreement to robot movement, while referencing to world is useful for calculations which require an external, absolute frame of reference.
- **Joint commands** enable the direct joint control given a joint name.

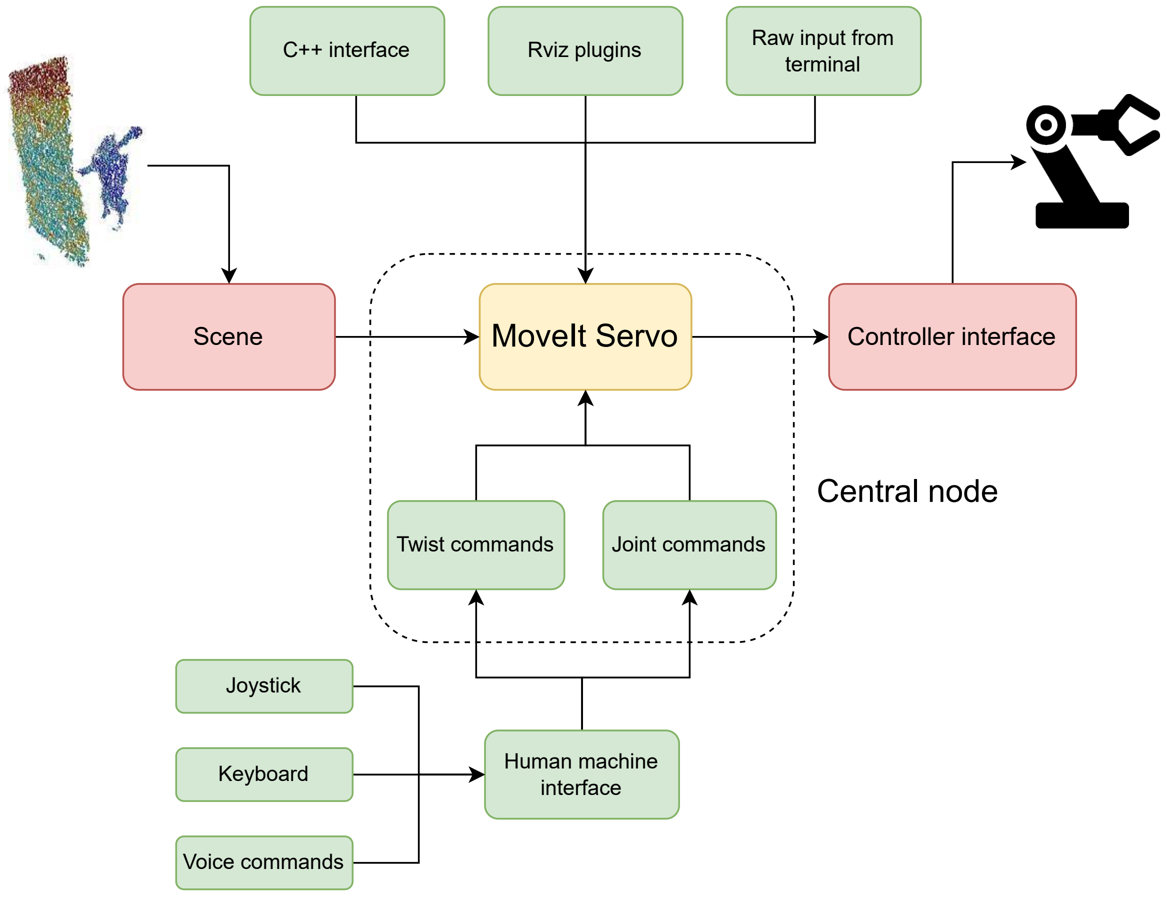


Figure 2.13: MoveIt servo workflow - human machine interfaces are outside central node architecture

MoveIt Servo need specific configuration files whose purpose is to define how to handle incoming input and how sensitive has to be the output (3.7).

2.7. User interfaces

Thanks to ROS topic and services, robots can be visualized or controlled with external interfaces, allowing the user to use the software he needs at will. In a medical scenario, it is not always necessary to see a 3D reconstruction of robot, furthermore, the selection of appropriate human machine interfaces requires plug-and-use modules quickly exchangeable.

ROS has several pre-implemented tools for interacting graphically with robotic systems:

- **Rviz** is the primary tool for visualizing your robot and its sensor data in 3D
- **Gazebo** is a set of tools for advanced robot visualization and physical simulation

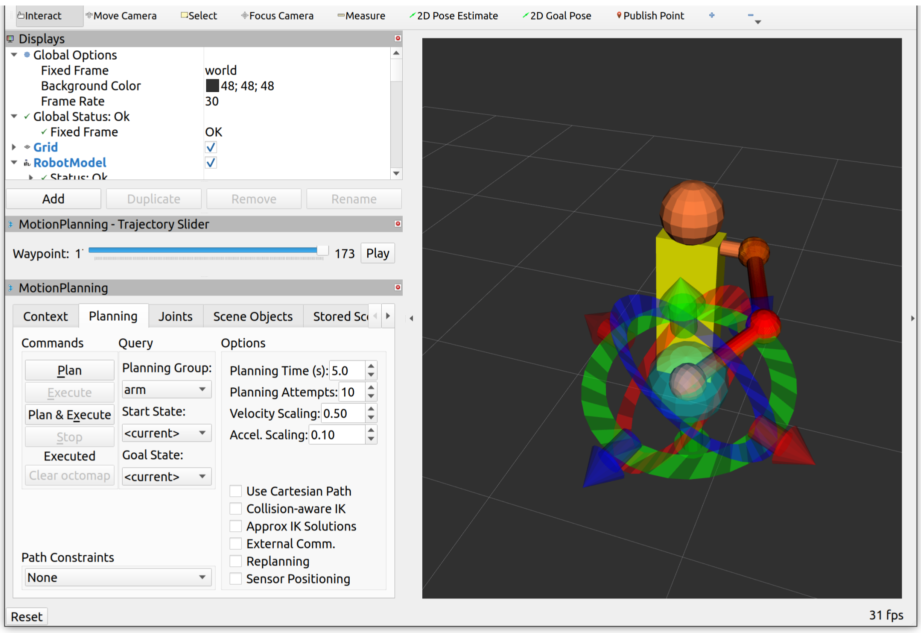
- **RQt** is for creating a graphical user interface (GUI) in ROS.
- **Plugins**: thanks to the plugin structure, the user can integrate his own graphical or human machine interface or integrate an existing one with minimal effort.

2.7.1. Visualization tools

Visualization tools represent complex and nuanced data in a digestible format and bring some much-needed sense and order to the multidimensional nature of robotics data. The most used ROS2 tools for data visualization are Rviz and Gazebo.

Rviz is used to visualize data in ROS in a 3D environment to help visualizing what the robot seeing and doing (2.14). Rviz can receive ROS messages and parameters of following type:

- **Paths** to visualize a series of points that have been visited over time
- **Visualization Messages** to display visual primitives
- **Markers** related to frame used to interpret their coordinates. Markers can be interactive to let user interact directly with the robot.



Global Options
Fixed Frame world
Background Color 48; 48; 48
Frame Rate 30
Global Status: Ok
Fixed Frame OK
Grid ✓
RobotModel ✓
Status: Ok

Waypoint: 1 173

Planning Group: arm
Start State: <current>
Goal State: <current>
Path Constraints: None

Planning Time (s): 5.0
Planning Attempts: 10
Velocity Scaling: 0.50
Accel. Scaling: 0.10

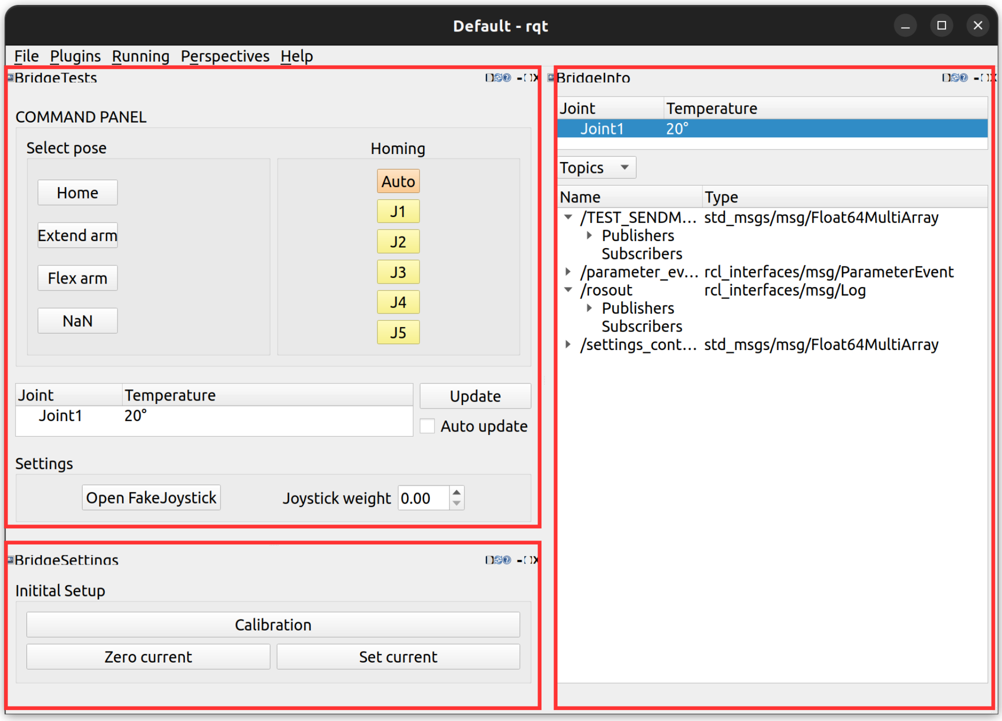
☐ Use Cartesian Path
☐ Collision-aware IK
☐ Approx IK Solutions
☐ External Comm.
☐ Replanning
☐ Sensor Positioning

Figure 2.14: Rviz exoskeleton visualization with end-effector marker to allow user interaction

Gazebo is a more complex set of tools to simulation with libraries and cloud services to build a realistic environment with advanced graphics engines. Compared to Rviz, it builds high-quality and more realistic virtual environments; for this reason it is preferable in situations where complex tasks have to be done.

2.7.2. GUI

Graphical interface is implemented through **RQt**, a Qt-based framework for GUI development for ROS; the implementation logic is similar to that of ROS plugins, in particular by means of dockable multiple widgets in a single window (fig. 2.15). Each widget is a separate and independent plugin with private settings, and threads.



Joint	Temperature
Joint1	20°

```
Name                Type
▾ /TEST_SENDM...    std_msgs/msg/Float64MultiArray
    ▸ Publishers
      Subscribers
▸ /parameter_ev...  rcl_interfaces/msg/ParameterEvent
▾ /rosout           rcl_interfaces/msg/Log
    ▸ Publishers
      Subscribers
▸ /settings_cont... std_msgs/msg/Float64MultiArray
```

Figure 2.15: Widgets in red; every widget is designed and loaded individually.

Users can create custom plugins for RQt with either Python or C++ using rqt metapackages or include one of the many developed by the community. They vary from simple topic/services administration tools to complex robot plugins to interact with them at runtime.

The advantage of using RQt from an architecture's perspective, is the **multi-platform** support (thanks to QT extended libraries) and the easier maintenance because of the use of common API. Moreover, a practical aspect is the possibility to completely integrate an existing Qt-based project into a RQt widget with minimal changes and additions: ROS2 and Qt-applications communicate each other through the use of ROS **Topics and**

Services so they can exchange robot models and processes information, such as operative nodes and control performances.

In a possible scenario, an RQt GUI can be designed, thanks to ROS2 topics, to access and control hardware through the ROS2 hardware layer (2.7).

2.8. Docker

ROS libraries are built for specific versions of ROS middleware and operating system. In order for an implemented package to work independently from the underlying system, a possible solution is containerization.

Containers are used to package and run an application, along with its dependencies, in an isolated, predictable and repeatable way. Containers require less system resources than traditional or hardware virtual machine environments because they don't include operating system, thus provide faster performances and increased adoption in mini-PCs.

Reproducibility To correctly reproduce a work, the most complicated way is provide a detailed README with all the necessary dependencies, installation steps, troubleshooting tips, etc. Alternatively, is possible to handle all those potentially tricky dependencies inside a Docker container that can be shared with much easier set of instructions to get things running.

OS abstraction Multiple projects can be compiled for specific frameworks or OS version. If a developer is working on projects that use different versions of ROS (or different versions of software in general), then switching between host machine's environment may be painful, if not impossible, to get right.

Process distribution Connected to OS abstraction, developers can run different processes on different containers; this is particularly helpful into an assistive scenario, when the wheelchair must be as lightweight as possible. For instance, a cloud workstation can perform 3D virtualization calculus while a single-board computer mounted on the back of the wheelchair handles the less heavy calculations. This is only possible if processes can abstract OS and run independently on the machine environment.

3 | Methods

The robotic platform and materials (2) previously described and selected offer a well assorted framework for the purposes of this work: demonstrating the possibility to build a software architecture, able to satisfy the requisites of assistive and rehabilitative robotics.

In order to do that, a custom architecture was implemented and explored on each of the main elements constituting it: the **robot description**, the **hardware abstraction**, the **control** systems, **planning** and **direct control** functionalities, **human-machine** interfaces and visualization. The main focus is to test if each of these components can be made customizable and adaptable, while still conserving the collective functionality of the system.

The proof of concept is mainly structured following the increasingly abstract levels of the overall architecture. Each layer of abstraction (in particular hardware abstraction layer, control layer, planning layer) is tested independently from the rest of the architecture, with modifications designed to test its modularity and the robustness of the remaining system ("horizontal" experiments).
After that, a final set of tests including the whole architecture ("vertical" experiments) is performed on an assistive exoskeleton.

3.1. Hardware setup

The main hardware consists in 3 different types of actuators, 2 types of drivers and one computer used to run the main software architecture. The "vertical" experiments are performed on the BRIDGE exoskeleton.

3.1.1. Hardware setup for "horizontal" experiments

To test independent functionalities of the different layers of the architecture, in particular the hardware-abstraction modularity, a testbench was constructed (fig: 3.1).

It consists mainly of 2 drivers connected to 3 effectors:

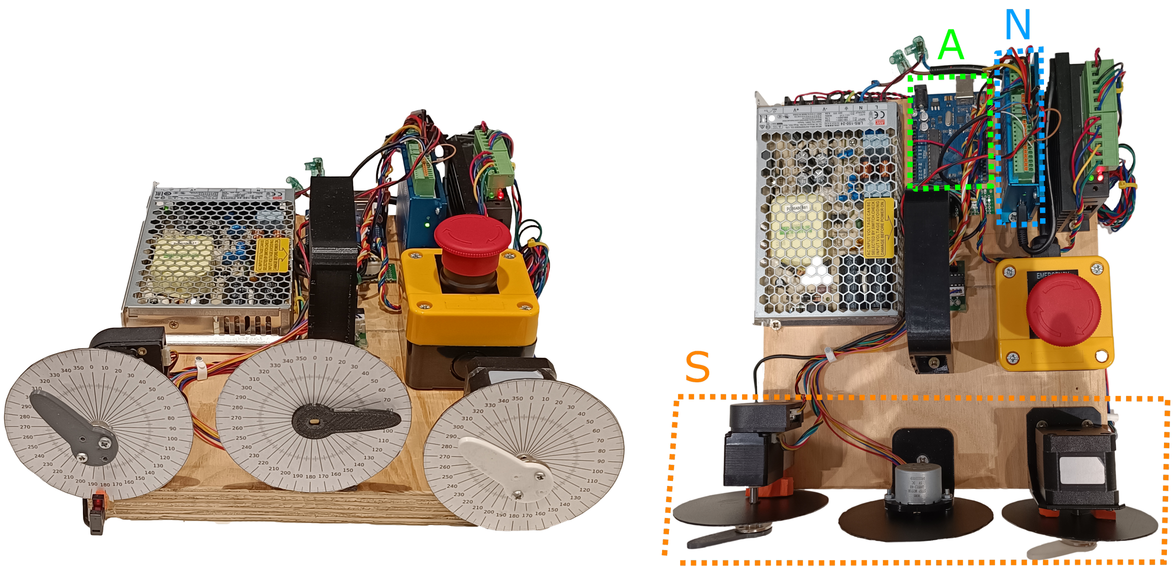


Figure 3.1: Test-bench setup - On the left, frontal view of the testbench with position indicators of the motors. On the right: top view of the testbench, with the components highlighted: A Arduino driver; N nanotec driver; S steppers

- A **Nanotec driver** (`SMCI33-1`) connected to a **Nanotec stepper** (`ST2818S1006-B`) with an integrated **encoder**. Nanotec driver and stepper were chosen for horizontal experiments as they constitute the same hardware implemented for the **BRIDGE exoskeleton** (3.1.2).

  The Nanotec driver is able to implement accurate functionalities for the actuators, such as low-level control loops, different transients for the control mechanisms, sensor reading, profile selection with configured parameters, calibration through external or internal inputs and more. Still, it is more expensive than the remaining hardware setup for the testbench put together.

  The driver allows for **velocity** and **position** control, returns a **position** feedback from encoders and dialogues with the computer through the USB serial protocol. The calibration function was exploited using a photoelectric sensor (`PM-25`) connected to an analog input. The only thing needed by the hardware abstraction to communicate with the Nanotec driver is knowledge about the **serial control string** (3.1)

- An **Arduino driver**, connected to the **28BYJ stepper**, and to the **Nema 17 stepper**. The Arduino driver consists in an Arduino UNO board, able to handle serial USB communication and simultaneous control of multiple actuators. It is

header	motor address	command	value	tail
`#`	`string`	`string`	`string`	`\r`

Table 3.1: Nanotec driver, serial control string. Length of the string is variable and ends with a carriage return byte. The motor address identifies the actuator to control. Commands include setting profiles, direction of movement, position and velocity, gathering and changing setup parameters of the driver, initiating a calibration procedure, getting position feedback and error information

effectively a more economic alternative than the Nanotec driver, depending on the application scope and requirements.

A `C++` firmware [5] was implemented on the UNO board, following the "modularity" philosophy: all instantiated actuators are derived from the same base abstract entity, and then specialized for each individual stepper. Thus, the functionalities of the firmware are independent from the stepper motor.
The firmware supports **velocity or position control** of actuators, and can return a **position state**. The position state is given by the algebraic sum of past commanded steps: it is not measured by an encoder and it may be prone to significant errors if the step is commanded but not executed, or vice-versa.

These information are exchanged through **USB serial** protocol with the computer running the ROS2 middleware. Again, the important part for the hardware abstraction is the **serial control string** (tab: 3.2).

header	motor ID	command	value MSB	value LSB	tail
`s`	8-bit	[`v`,`p`,`w`,`I`]	8-bit	8-bit	`e`

Table 3.2: Arduino driver, serial control string. The motor ID identifies the actuator to control. Commands `v`, `p`, `w` and `I` are used respectively for velocity control, position control, "whoami" identifier and position feedback

3.1.2. BRIDGE exoskeleton hardware

The BRIDGE exoskeleton is a four degrees-of-freedom active orthosis for upper-limb assistance, designed to be worn on the left upper limb (fig: 3.2). It consists of three shoulder joints and one elbow joint, each consisting of a rotation around the drive system axis. Joints' ranges of motion can be adapted to user's capabilities at mechanical level; moreover, the elbow module can be positioned according to the patient's anthropometric

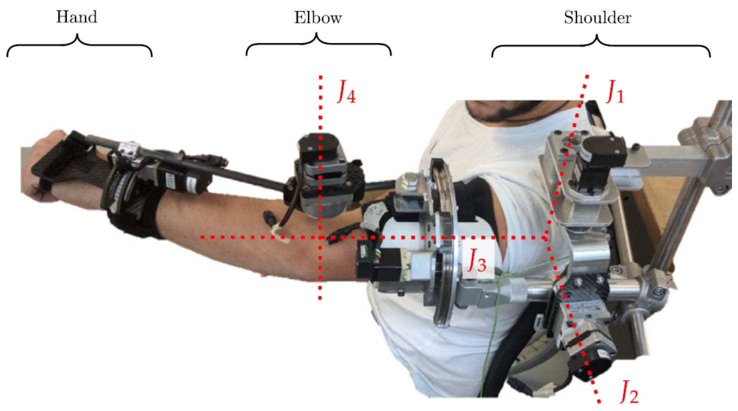


Figure 3.2: BRIDGE exoskeleton and rotational axis [17].

measures. The system includes a mounting frame to be secured to the personal user's wheelchair [17].

The actuating drivers and effectors are all **Nanotec** drivers and steppers, described in (3.1.1), but with different characteristics depending on the torque required by each joint. In particular, J2 shoulder joint must support arm lifting, therefore it is equipped with an anti-gravity system, consisting of an elastic element in parallel with the actuator. Each joint has also a gearbox connecting the structure to the actuators, with varying reduction ratios.

3.2. Robot description

Two different robot descriptions were implemented: a simple robot description for tests with individual actuators; a robot description for the BRIDGE exoskeleton.

3.2.1. Robot description for "horizontal" experiments

A simple 1-DoF URDF model was constructed in order to represent the actuators of the test-bench. It consists of a single continuous rotational joint connected between a fixed "base" link (the actuator case) and a movable "indicator" link, which is simply used to represent the motion in the virtual world of the RViz visualizer (fig: 3.3).

All three actuators of the testbench are portrayed through the same abstract description. To control and visualize them in parallel, one has to simply instantiate the same description multiple times, with a name indicating which actuator they represent.

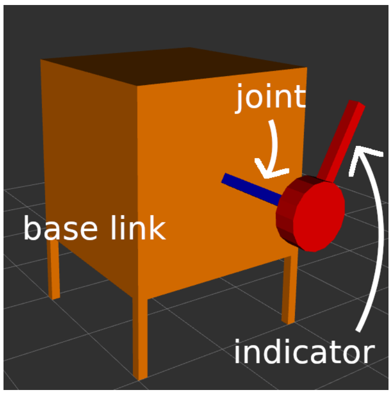


Figure 3.3: Robot description of a single actuator of the testbench

3.2.2. BRIDGE exoskeleton abstract description

The BRIDGE exoskeleton description in URDF/XACRO was implemented following the kinematic modelling described in [17] (fig: 3.4). It's composed by 4 revolute joints:

- J1: shoulder horizontal abduction/adduction
- J2: shoulder vertical flexion/extension
- J3: humeral rotation
- J4: elbow flexion/extension

The first three joints are used to replicate the shoulder glenohumeral joint (GH), approximated as a ball-and-socket joint. This is equivalently obtained through a kinematic chain of three rotational references whose axes intersect the GH joint center of rotation. The kinematic model follows the modified Denavit-Hartenberg parameters [6]. Figure (3.4) shows the reference frames of the implemented abstract joints. The rotation is always around the local z-axis. Figure (3.5) shows the implemented description of the exoskeleton visualized in RViz, including a fixed base-link mimicking the trunk and head of a person, for better understanding.

Each joint has also software customizable ranges of motion (position limits) and velocity limits, in order to be safe and user-specific. Collision and inertial information is added through simplified shapes, like cylinders for links and spheres for joints. Moreover, reduction ratios and offsets of the joints are added as parameters in the URDF, which will be interpreted and used by the hardware abstraction.

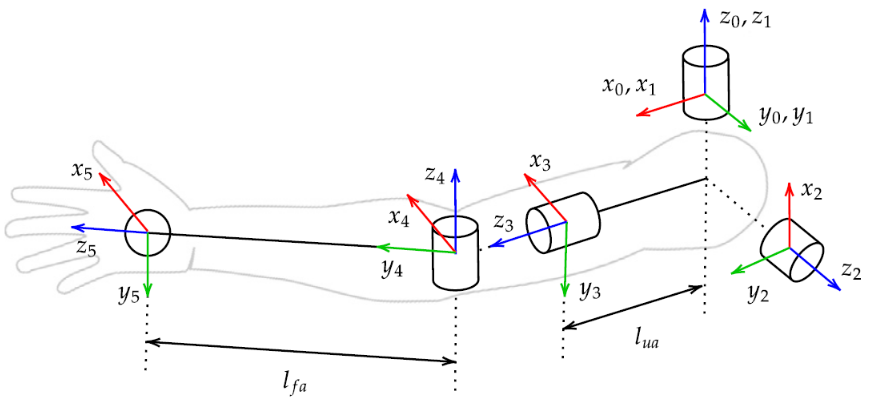


Figure 3.4: BRIDGE reference frames for joints and links, following the modified Denavit-Hartenberg convention [17]

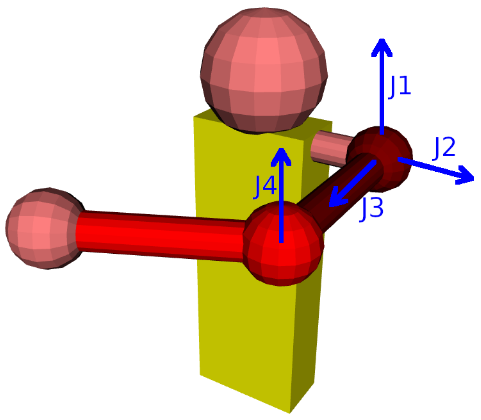


Figure 3.5: Robot description of the BRIDGE exoskeleton, rotational axes of the various joints are shown in blue

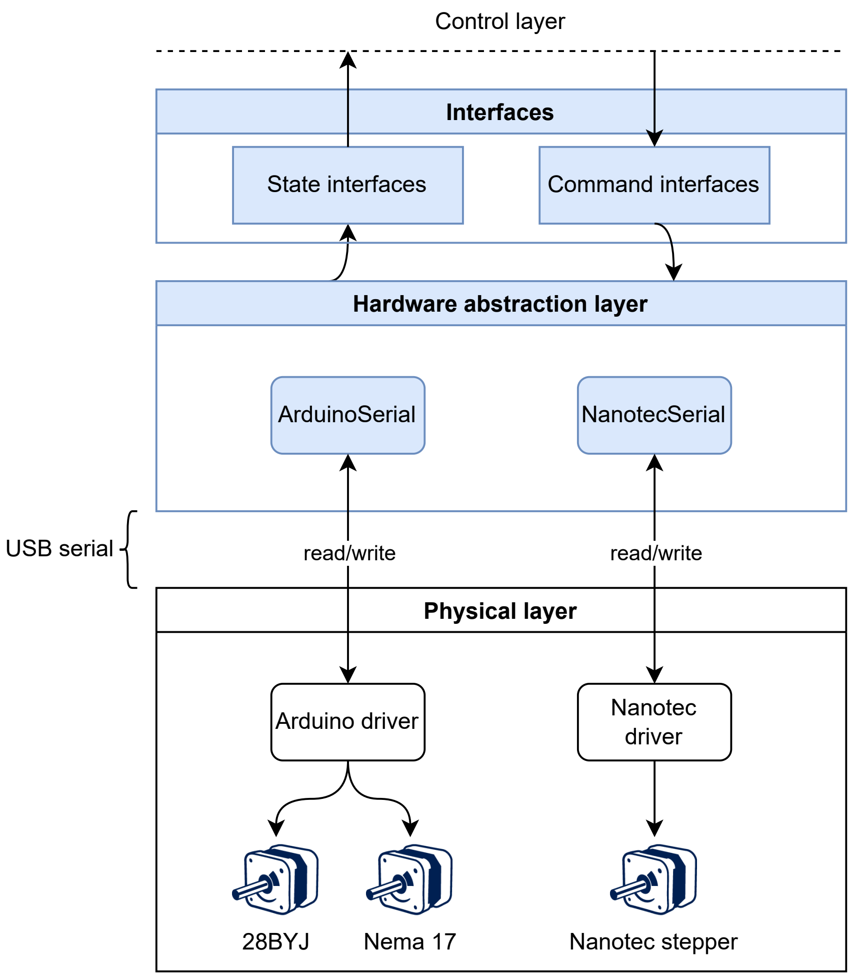


Figure 3.6: Implementation of hardware abstraction layer

3.3. Hardware abstraction implementation

As two different types of drivers were used to connect with the hardware, only two different component designs needed to be implemented in the hardware abstraction: `NanotecSerial`, able to communicate with the Nanotec driver, `ArduinoSerial`, able to communicate with Arduino driver (fig:).

Both hardware component designs use USB serial protocol to communicate with the corresponding drivers. To achieve that, the `serialib C++` library has been used [29].

The functionalities of the two component designs are enclosed in their life-cycle (2.4):

- **Initialization**: URDF parsing to extract logical components (joints and links), interface types (2.4.1) and parameters such as transmissions. As the Arduino driver is able to control multiple actuators simultaneously, 2 different joints are controlled in a single instance of `ArduinoSerial` component. The `NanotecSerial` component instead drives a single actuator and encoder for each instantiated component.

- **Configuration**: connection with the drivers is established automatically by scanning serial ports, as both types of driver can provide a driver ID as echo. Also the **state** and **command** interfaces (2.4.1) are exported: both component types support *position* and *velocity* command-interfaces and *position* state-interface. `NanotecSerial` component also has a *settings* interface, used to send on-demand commands to the hardware, such as calibration directives, and receive other information such as temperature and errors.
- **Activation**: all variables are initialized and a final check is performed for possible errors.
- **Execution**: the components follow the controllers' directives enabling actuation and perception functions. These mainly consist in step-to-radian conversions and transmissions' implementation.

3.3.1. Hardware abstraction for "horizontal" experiments

Concerning the testbench setup, both hardware component types just mentioned were used. The URDF implemented is that of the single actuator (3.2.1). Therefore, some parameters such as reduction-ratios are not necessary.

For testing the hardware abstraction functionality, the hardware composing the testbench is connected to the computer. The software architecture must be able to support all two drivers and three steppers, even in parallel, demonstrating hardware modularity.

A set of four tests were chosen for this purpose: all actuators at the same velocity; each actuator at a different velocity; all actuators at the same desired angular position; each actuator at a different desired position. This is done in order to test the different command modalities and hardware characteristics, while getting a position feedback to evaluate the results. The evaluation consists in assessing the outcome qualitatively, showing whether the functions were respected or not, without going to much in detail on the quantitative accuracy as this is not the scope of the experiments.

The `ros2_control` package also offers the possibility to use **Fake Hardware**: it consists of a standard hardware abstraction that accepts all types of interfaces and simply mirrors the commands back as states for the controllers (fig: 3.7). This is done in order to test the communication between controllers and hardware abstraction layer, without the need to connect to a physical hardware. The Fake Hardware functionality will be used in

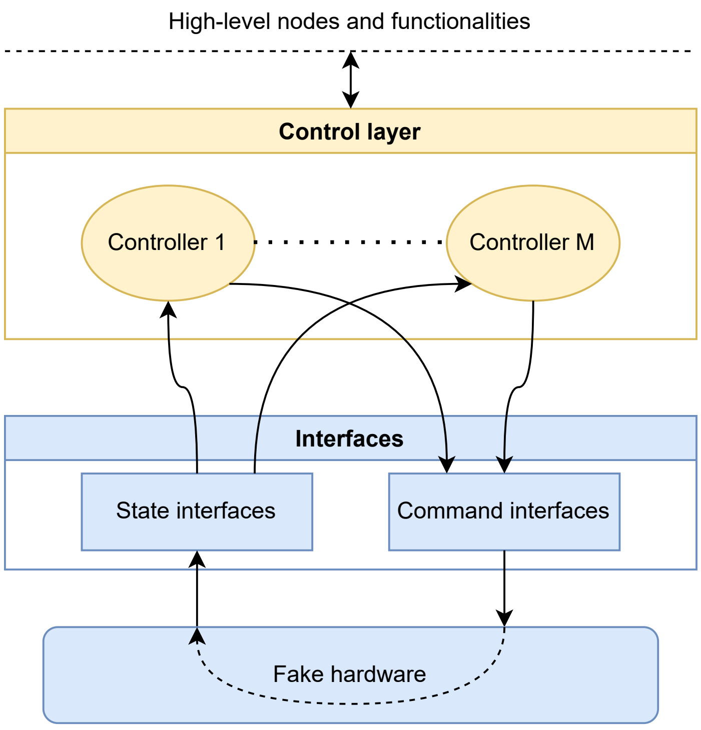


Figure 3.7: Fake hardware mirrors commands back to states

the horizontal experiments of high-level control modalities, such as planning and direct control of end-effector: in this way, they can be tested as if they were independent from the hardware abstraction.

3.3.2. BRIDGE exoskeleton hardware abstraction

In case of multiple drivers of the same type, as in the BRIDGE exoskeleton, the hardware component corresponding to that type can be simply instantiated multiple times. This is done by declaring multiple joints in the BRIDGE URDF description (3.2.2), each using the same `NanotecSerial` hardware component type (fig: 3.8). Thus, multiple processes will be initiated, each driving its own specific actuator driver, with specific parameters such as reduction-ratios, offsets and driver IDs for auto-connection. This implementation will be used for the "vertical" experiments comprising the whole architecture.

3.4. Control layer implementation

Regarding the control layer, the main goal is to verify if different control modalities and schemes can be implemented, without caring about the remaining architecture. Most controllers are already implemented by the ROS community, though with the limitations

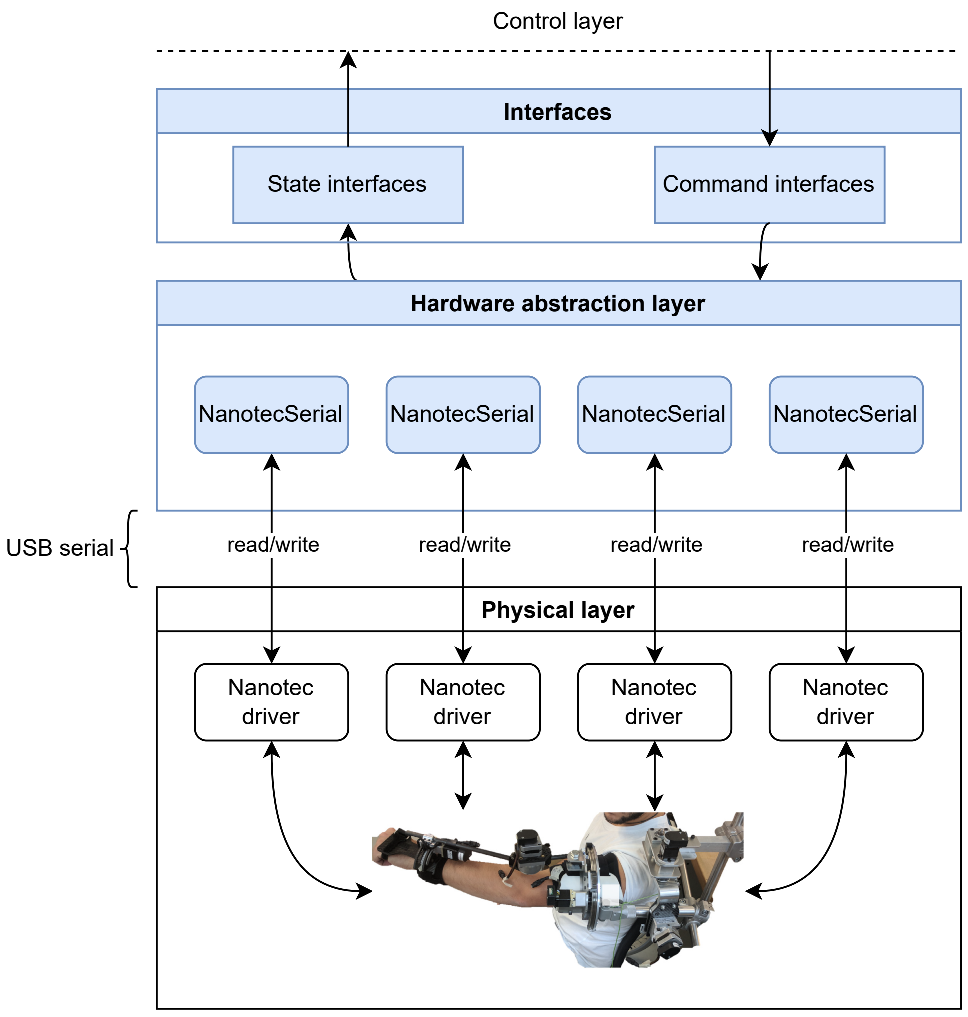


Figure 3.8: Hardware abstraction implementation for the bridge exoskeleton

introduced before (2.5). In particular, the following were chosen:

- **Forward controllers**: simple open-loop controllers with a proportional transfer function between input and output. They can control multiple joints simultaneously, with different output types, such as the *position* and *velocity* interfaces required by the implemented hardware abstraction.
- **Trajectory controllers**: Used to execute trajectories on a set of joints. A trajectory is specified as a set of way-points to be reached at specific time instants. Way-points may consist of position and optionally velocity or acceleration. The controller can work with different trajectory representations: by default a spline interpolator is provided. Trajectories are sent to the controller through an *action* interface (2.2.1). Also, it can write to multiple command and state interfaces, using a PID transfer function when required.
- **State broadcaster**: It's not really a controller in control-theory terms, as it doesn't send any command. Its function is to read all state interfaces of all joints and report them on a ROS topic that can be used by other nodes of the system (e.g. `rViz` and higher level controllers and planning functions). This functionality is used in all setups, at least to provide visual feedback.

Other controllers were implemented in this work, to exploit the customizability of the framework:

- A **PID controller** with position feedback and output in velocity, with the role of maintaining at each update a desired position:

  $$\dot{x}(t) = K_p\, e(t) + K_i \int_0^t e(\tau)\, d\tau + K_d \frac{de(t)}{dt} \tag{3.1}$$

  Where K_p, K_i and K_d are the PID coefficients, $\dot{x}(t)$ is the velocity command, $e(t) = x_r - x_m$ is the position error between desired reference position and measured position.

  The main structure is derived from the forward controllers, adding a position feedback interface, while the PID processing is obtained from the ROS `control_toolbox` package.
- **Settings controller**: a general purpose controller used to dispatch on-demand commands to the hardware abstraction. Differently from other controllers, it doesn't send commands at each update cycle, but only once, in correspondence to the callback of its input topic.

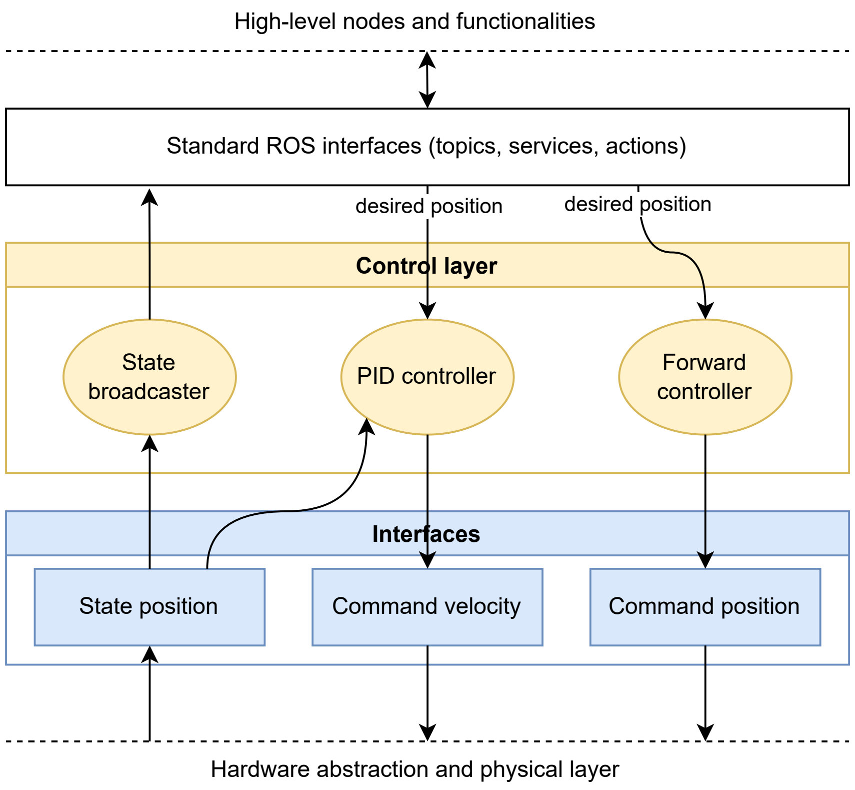


Figure 3.9: Implementation of two different control modalities: open-loop through forward controller, close-loop through PID controller

Controllers and **Controller Manager** node are declared and configured in a parametric file. The Controller Manager needs to know the update frequency of the control cycle, to which it synchronizes all its child controllers. Then it needs a list of all controllers it has to handle. Controllers are then specified with their own parameters.

3.4.1. Controllers for "horizontal" experiments

The control layer is tested against the architecture modularity with two different control modalities (fig:3.9):

- An **open-loop** control scheme, implemented through a **forward controller**;
- A **close-loop** control scheme, implemented through the custom **PID controller**.

The two modalities are tested keeping the hardware and hardware-abstraction fixed. The Nanotec driver was chosen for this purpose as it provides an integrated encoder for position feedback.

For horizontal experiments on higher layers, such as planning and direct control, the

Trajectory controller is used as it provides the most advanced functionalities for the purposes.

3.4.2. BRIDGE exoskeleton software controllers

As the exoskeleton must support all higher-level functions, the **Trajectory controller** was the preferred choice for all the main tests. At the same time, the custom **Settings controller** is working in background, listening to the user and system intentions of calibration and other setup functionalities.

3.5. MoveIt2 configuration

Since MoveIt2 [4] provides ready to use tools, a set of parameters have to be established to achieve the experiments goal.

Some of these features are not really used, such as **3D perception**, which is used to generate points cloud or depth map from sensor data. Perception file is indeed empty, since no external 3D sensor is used to perceive real world objects which are virtually crossed by trajectory. Tests' obstacles are virtually built measuring the real world object coordinates with respect to a reference frame, then placing a similar shaped object in the same virtual coordinates.

Joint limits are set to allows the dynamics properties specified in the URDF to be overwritten or augmented as needed. They include velocity and acceleration limits and scaling factors and are customized to limit maximum velocities/accelerations of URDF ones.

To configure **Kinematics**, a plugin has to be specified and its parameters must be configured. Plugin has to include also forward kinematics and finding jacobians. For testing purposes, **KDL** solver (from Orocos Kinematics and Dynamics, it integrate a joint-limit-constrained pseudoinverse Jacobian solver [2]) is used, because of its widespread adoption.

Moveit controller manager was a complete set of nodes and services to manage old ROS controllers but nowadays is implemented as a medium to connect ROS2 controllers and joint groups. This architecture will be soon deprecated, so its configuration are strictly related to define controller names, which are matched to the ones defined in the controller layer. Their correct coupling is essential to let MoveIt2 be conscious of used controllers.

3.5.1. Planning configuration

Planner configuration require set of files, one for each planner used. Different planners may requires specific parameters but, since in this work only **OMPL motion library** is integrated, only its specific parameters need to be configured. For how OMPL is built, the greater the number of planners allowed to be used, the greater the performances (2.6.1).

Planners included are SBL, EST, LBKPIECE, BKPIECE, KPIECE, RRT, RRTConnect, RRTstar, TRRT, PRM, PRMstar, FMT, BFMT, PDST, STRIDE, BiTRRT, LBTRRT, BiEST, ProjEST, LazyPRM, LazyPRMstar, SPARStwo, TrajOpt. For each of them, planning parameters are defined (tables A.1-A.2), otherwise defaults are used by OMPL. For some tests, it is forced to prefer RRTConnect because of its better performance but if it fails to find a trajectory, OMPL automatically try with another planner.

Upstream and downstream a planner activity, **adapters** used are:

- **Time Optimal Parameterization**: it time parameterize the motion plans converting paths to trajectories, applying velocity and acceleration constraints.
- **Resolve Constraint Frames**: resolves constraints that are defined in collision objects or sub-frames to robot links, because the former are not known to the planner.
- **Fix Workspace Bounds**: it specify a default workspace for planning (a cube of 10m x 10m x 10m) if planning request does not specify it.
- **Fix Start State Bounds**: it fix start state bounds adapter fixes the start state to be within the joint limits specified in the URDF, otherwise motion planner will be unable to plan. The fix is acceptable if the joint is slightly outside joint limits, so a parameter is set to define how much the the joint can be outside its limits for it to be “fixable”.
- **Fix Start State Collision**: The fix start state collision adapter try to sample a new collision-free configuration near a specified configuration (in collision) by perturbing the joint values by a small amount, defined as a percentage of the total range of motion for the joint. It is also specified how many random perturbations the adapter will sample before giving up.
- **Fix Start State Path Constraints**: if the start state for a motion plan does not obey the specified path constraints, an interim location where the path constraint is obeyed is included and will be used as the start state for planning.

3.5.2. SRDF configuration

Since SRDF require to semantically aggregate joints and links into groups, a chain from the trunk to the forearm is selected to identify the **arm**.

A single group pose is specified (referring to home position) for the arm with joint values `<0.1935,0.4363,0.2637,-0.9361,0.0854,0.0>`. This pose will be used in the test phase to have a reference to start from.

A fixed robot (like an industrial manipulator) should be attached to the world using a fixed virtual joint to represent the virtual motion of the robot base with respect to the ground which is an external frame of reference (considered fixed with respect to the robot); In this work, a virtual joint is defined to bind the exoskeleton to a fake wheelchair.

As the last configuration, a Self-Collision Matrix or Allowed Collision Matrix(ACM) have to be built. By default it is assumed that any link of the robot could potentially come into collision with any other link in the robot. ACM includes pairs of links that can safely be disabled from collision checking, decreasing motion planning processing time. Pairs are disabled when links are always/never in collision, adjacent to each other or if they are in collision in the initial pose.

3.6. Offline planning implementation

Offline planning rely on the same basic scheme of figure 1.10. Once all the parameters have been set, nodes, topic and services are generated (fig. 3.10): central node is move group (2.6.3) which integrates the core functionalities to handle planning requests and collision checking.

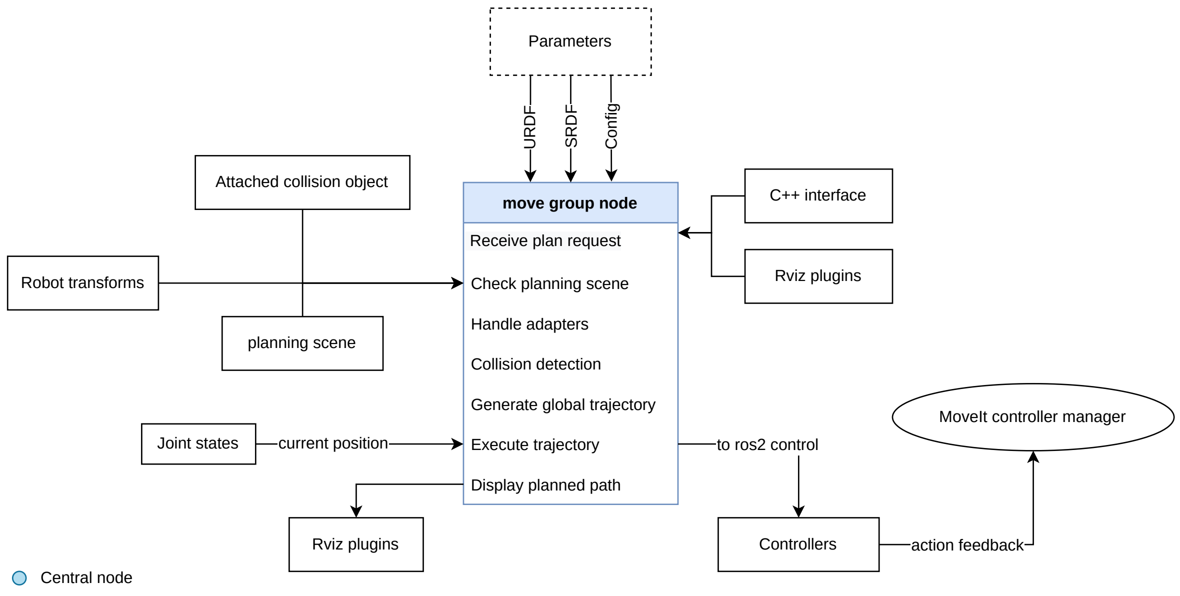


Figure 3.10: Moveit2 planning architecture

In particular, move group node (fig. 3.10) workflow is the follow:

1. reception of the **plan request** from Human machine interfaces, such as visualization tools (Rviz) or C++ interfaces. The latter include hardware inputs (keyboards, buttons etc.) or graphical interfaces.

2. Handling of plan request reading set parameters from URDF, SRDF and configuration files (3.5)

3. Check of the the **planning scene** using relative topic; scene is also updated according to attached objects and robot transformation.

4. Depending on the planner algorithm, **collision detection** is made before path generation (if a path is extracted, it is always possible to execute it) or after path generation (if path intersects an object, planner repeat N times the motion planning until a free path is found).

5. After path is found, **adapters** are applied, generating a trajectory, which is a time-parameterized and fixed path (3.6)

6. Trajectory is sent to **controllers** and an Action interface is generated, so a feedback signal is received by MoveIt controller manager for every point of the path.

7. Trajectory is also sent to GUI plugins which display the path in a three-dimensional space visualization.

In the **horizontal tests**, trajectories are sent to a fake controller, whose purpose is to copy commands into joints states. The test goal is the correspondence of goal state (fixed in advance) and joints final configuration. If they match, it is proved that the method is able to correctly reach a goal position from an initial state.

Another test is done to check move group collision detection, in particular an object is placed along the optimal path, then the executed trajectory is compared to a reference without the obstacle.

For **vertical tests** trajectories are sent to control layer, so using all the lower layers already tested.

As for horizontal tests, the experiment is considered passed if joints final configuration match the goal state.

3.7. Direct control implementation

Servo node, which is the central node used to direct control the exoskeleton, requires additional parameters to be set; they include command type, that could be `unitless` or `speed units`. The latter is expressed as m/s or rad/s, while unitless is in range of $[-1:1]$, an ideal input for joystick commands. Both of them are assigned because in the test phase, different inputs will be used (joystick and keyboard).

Collision-check rate is enabled at low frequency to not bog down a CPU if done too often, furthermore, a threshold under which decelerating if self-collision may occurs is set. Singularities are instead monitored through soft and hard thresholds, respectively conditions to start decelerating and completely stop the motion. As a safety margin, a parameter set a buffer to joint limits, if moving quickly, buffer becomes larger to let CPU computationally follow the real movement properly. Finally, commands smoothing is committed to a Butterworth filter plugin.

Twist and joint commands are the 2 type of message that MoveIt Servo can receive from whatever human machine interface. For demonstration purposes, a **Joystick** and a **keyboard** nodes are developed to send both message types. In particular, to correctly detect hardware button pressing, a driver has been made (fig. 3.11). It is composed by an hardware input detector, whose purpose is to convert **ASCII symbols** to encoded values, and by a topic which send the encoded messages to the succeeding node for further conversions. Before Servo node indeed, an intermediary node converts raw values which indicate what button has been pressed, to twist and joint messages.

Before converting twist and joint commands to states in the joints space, servo node need to check for obstacles, avoid singularities and respect joint limits by accessing the planning scene and joints current states. If everything checks out, the resulting joints configuration is sent to controllers which generates a feedback for every new goal they reach.

This process is equal for both joystick and keyboard used in tests phase.

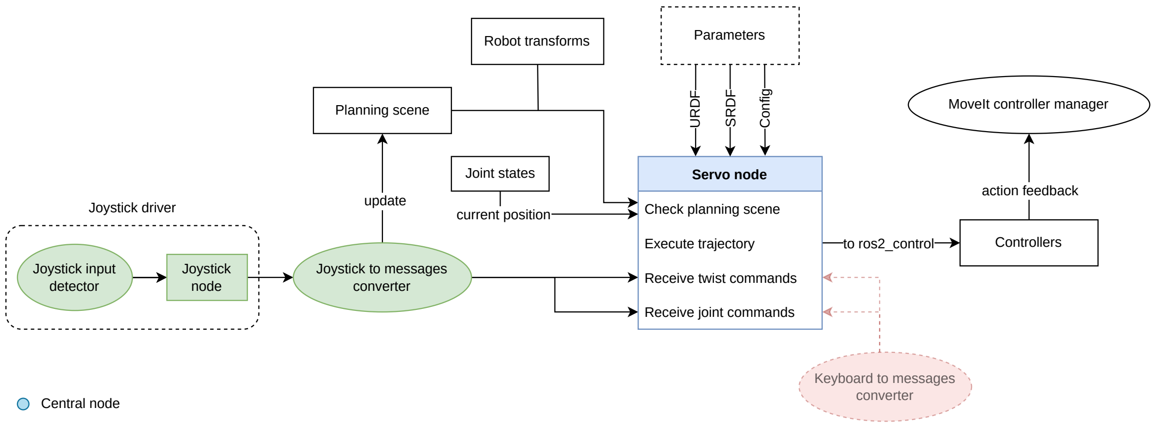


Figure 3.11: Servo planning architecture - green blocks belong to the servo variant with joystick, red block is the keyboard variant

The experiments are organized to demonstrate a subject can move with cartesian paths and in the joint space: as for offline planning tests, horizontal validation requires fake hardware, while horizontal experiments are done with real hardware and control layer that has been implemented in this work.

3.8. Hybrid planning implementation

Hybrid planning configuration parameters can be divided into three macro areas:

- **Local planner**: includes information about where to publish local solutions (controller's topic name and type), global solution topic information, working frequency and the plugin name that will solve **local constrain problem**.

  By default, Moveit2 uses a plugin whose main purpose is to ensure that no collision will happen in a short time window (2.6.4) but, no already-built algorithm is used since no external sensors are included in this work.

- **Global planner**: information about planning scene monitor defines what global planner has to consider as obstacles and let it know the initial joints positions. Planning pipeline is entrusted to OMPL planner (2.6.1) with the same configuration of offline planning.

- **Hybrid planning manager**: since it runs the planning logic and coordinates the planners, `planner logic` is an essential parameter to customize; planner logic refers to the support of mapping generic events to available actions provided by the architecture. In the basic implementation, logic simply ask global planner to re-plan if local constraint solver detect an obstacle on the path (2.6.4). Since there aren't external 3D sensors, default planner logic is replaced accordingly to the new local constraint solver.

The new local constraint solver behaviour is comparable to a "**fast switch**" system: Planning and direct control can be used alternatively without intervening in the architecture, because they are both active. Custom hybrid planning operation provides, specifically (fig.3.12), that the local planner is always listening from a joystick for incoming commands.

If a new input is received, an `event sent` flag is set to false; this flag allows to send a replan request once the joystick inputs are finished. Subsequently, joystick inputs are converted to joints space positions and sent to controllers as new way-point. In this scenario, controllers receives only the user input commands, therefore the control of the exoskeleton is totally entrusted to the joystick.

When no more joystick commands are received (fig.3.12), the `event sent` flag is checked: if it is true, it means there's no need to replan, because already replanned or the user never intervenes in the movement. Current way-point is then set as next point of global trajectory and sent to controllers, following the standard planning-execution process. If `event sent` is false, it means a replan is required, due to joystick intervention in the exoskeleton control. The exoskeleton is held still and a new global trajectory is calculated; in the next process iteration, the flag will be true and planning will resume working as usual.

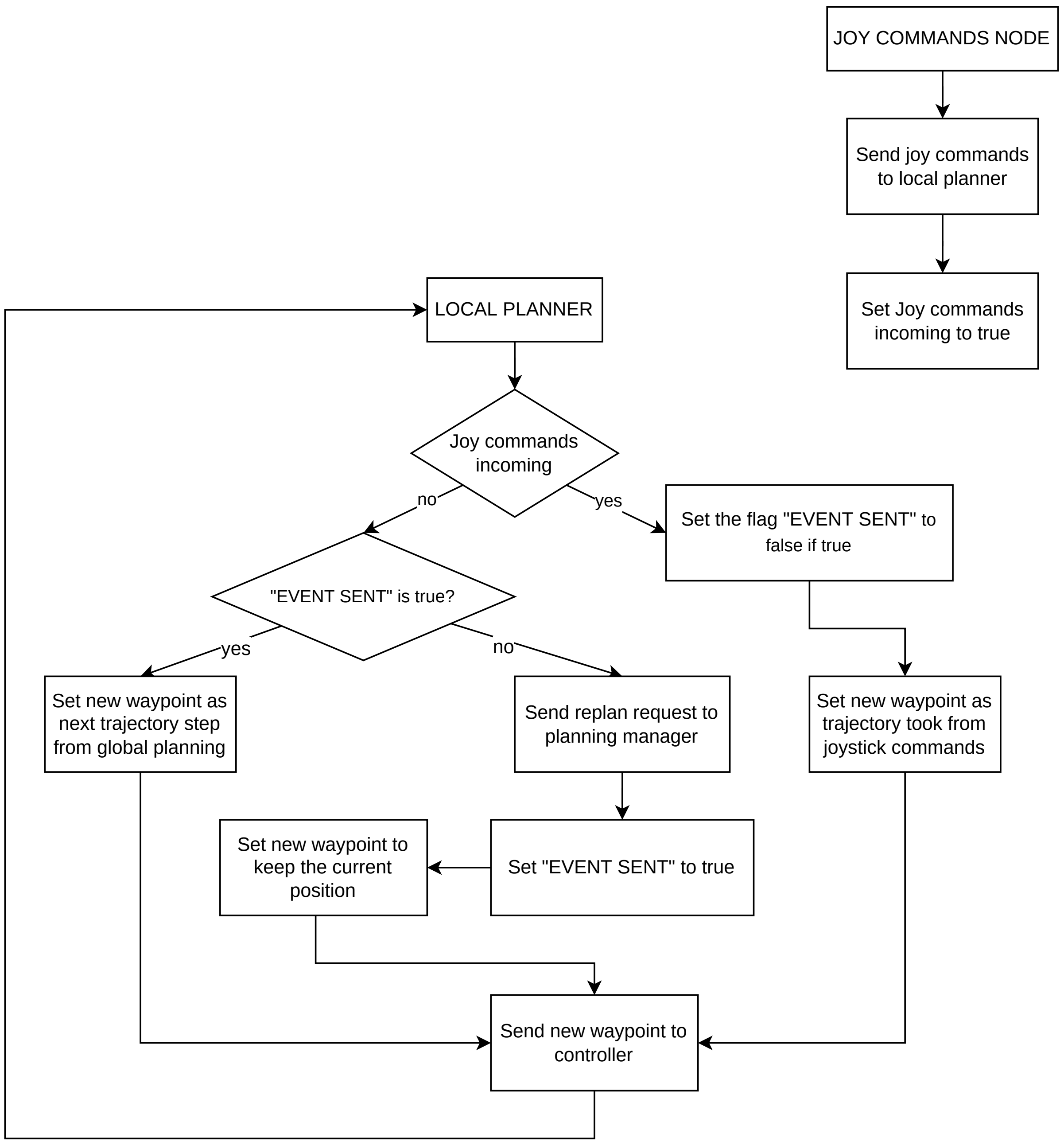

JOY COMMANDS NODE
Send joy commands to local planner
Set Joy commands incoming to true
LOCAL PLANNER
Joy commands incoming
no
yes
Set the flag "EVENT SENT" to false if true
"EVENT SENT" is true?
yes
no
Set new waypoint as next trajectory step from global planning
Send replan request to planning manager
Set new waypoint as trajectory took from joystick commands
Set new waypoint to keep the current position
Set "EVENT SENT" to true
Send new waypoint to controller

Figure 3.12: Fast Switch working diagram

Local planner and joystick node **frequencies** must be set properly, because if the local planner frequency is greater than that of the joystick, hybrid system will start replan at every loop iteration, making the control unusable.

Set of functionalities described are implemented (fig. 3.13) with three central nodes, two of them for motion planning (local and global planner nodes) and one for handling joystick commands. The upper section of the architecture (fig. 3.13) is essentially the same of direct control implementation (2.6.5). The only difference is that Servo node commands are not directly sent to controllers but to local planner, who will deal with planner logic described above.

Hybrid planning, beyond coordinating Global planner and local planner, has the task of handling motion requests that user send through a GUI. It also receives action feedback from controllers to make sure actuators are moving properly.

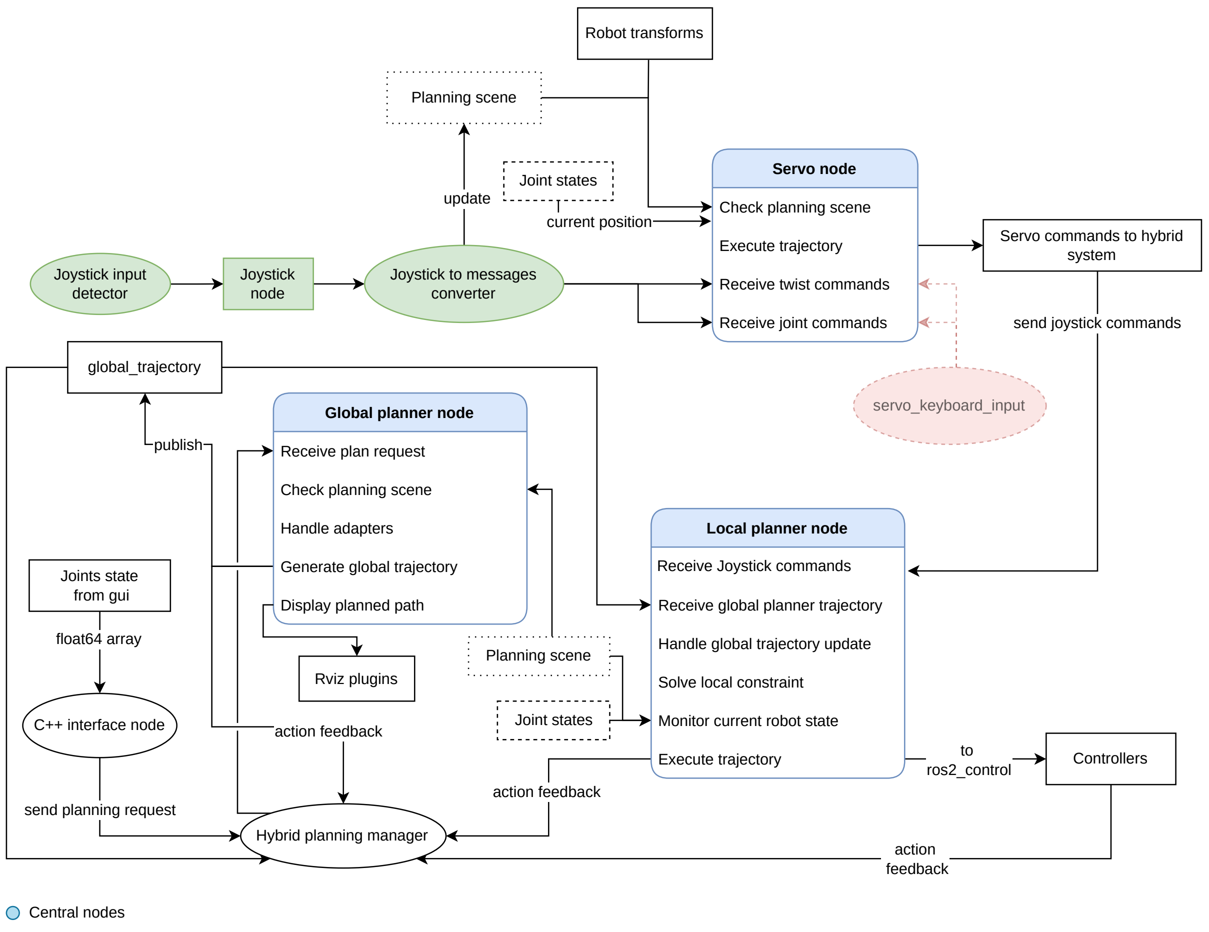


Figure 3.13: Hybrid planning architecture

Hybrid planning tests provide the demonstration of goal reaching despite user's input perturbation (like joystick): for this reason, joints final configuration is matched to a-priori goal state. If they are equal, the test is passed. As for offline planning and direct control, horizontal validation is done with fake hardware, while vertical experiments with real hardware and control layer that has been implemented in this work.

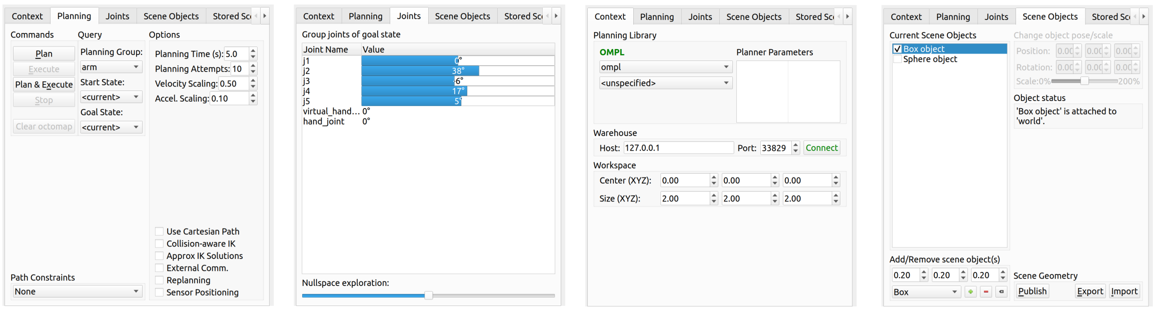
3.9. Human machine interfaces

Planning and direct control systems need interfaces to let user interact with them. Two approach are possible: physical input such as joystick, keyboard, pressure sensors or graphical interfaces thanks to RQt libraries (2.7.2).

3.9.1. Graphical interfaces

Some tools are already present in the MoveIt meta-package, they just need to be properly configured. One of them is selected to be used in this work thanks to its ease of use and completeness: **Motion planning plugin** allow to (fig. 3.14):

- Quickly switch planners which are used in the test phase
- Send a plan and execution request
- Set goal pose
- Include shaped objects in the 3D environment



Planning Group: arm
Start State: <current>
Goal State: <current>
Planning Time (s): 5.0
Planning Attempts: 10
Velocity Scaling: 0.50
Accel. Scaling: 0.10
Path Constraints: None

Group joints of goal state

Joint Name	Value
j1	0°
j2	38°
j3	6°
j4	17°
j5	5°
virtual_hand...	0°
hand_joint	0°

Planning Library
OMPL
ompl
<unspecified>
Planner Parameters

Warehouse
Host: 127.0.0.1 Port: 33829

Workspace
Center (XYZ): 0.00 0.00 0.00
Size (XYZ): 2.00 2.00 2.00

Current Scene Objects
☑ Box object
☐ Sphere object

Object status
'Box object' is attached to 'world'.

Add/Remove scene object(s)
0.20 0.20 0.20
Box

Scene Geometry

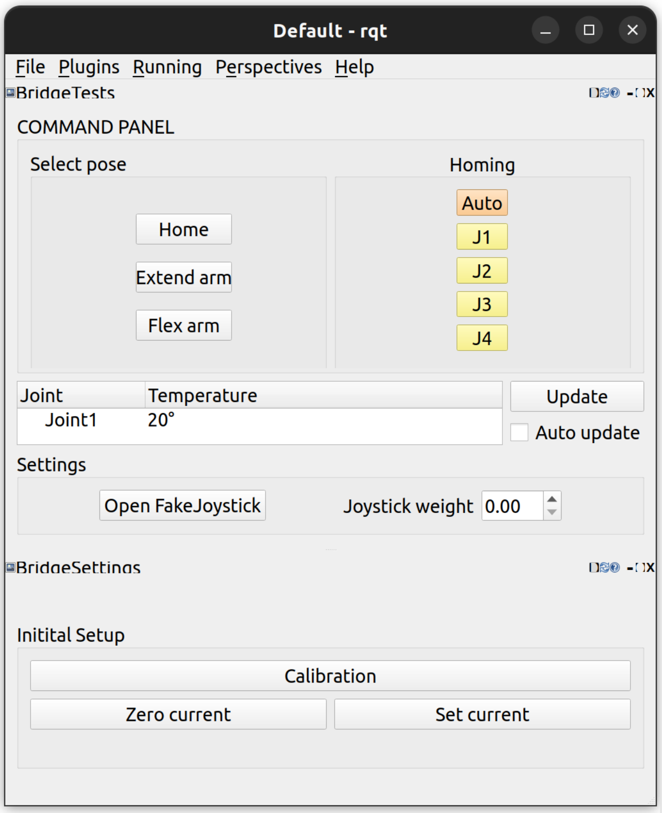
Figure 3.14: Planning plugin to handle planners and objects

For all other tests, custom graphical interfaces have been developed to adapt to specific purposes. To assist control layer validation (4.2) an adaptive GUI contains a variable number of sliders to test position and velocity control by selecting the relative modality. The control through graphical tools made it possible not to use the commands from the terminal and speed-up the test phase. Furthermore, a process monitor shows topics and services to check workflow and execution correctness.

User side, it has been necessary to integrate a **demo GUI** (fig. 3.15) to assist subject into vertical validation tests (4.5). Pose selection as a tab witch integrate the possibility of choose different test poses to send as motion plan requests to MoveIt central nodes.

Since the exoskeleton has 4 controllable joints, their homing is added as functionality with 4 different buttons. To set joints home positions, a calibration is required, thus a corresponding button is added. During calibration phase, user can set motor current to zero (transparent motion) or at working values.

Lastly, a dedicated section allow to monitor motor parameters such as temperature.



Joint	Temperature
Joint1	20°

Joystick weight 0.00

Figure 3.15: GUI to manage exoskeleton and its joints - functionalities include planning requests to predetermined poses, homing, stepper parameter control and calibration

3.9.2. Hardware interfaces

Despite the easiness of GUIs, hardware interfaces are always necessary when dealing with patients with reduced residual motor skills. Two type of hardware input have been tested: a joystick and a keyboard.

Joystick needs a software driver to convert hardware inputs to adimensional commands in the range [-1,1] for left/right analogs and on-off signals for the remaining buttons (figure 3.16).

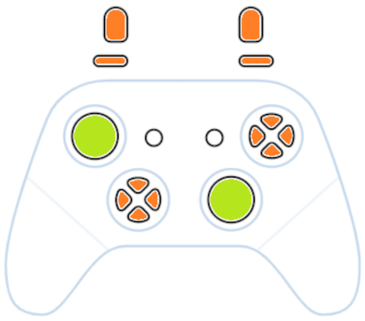


Figure 3.16: Joystick model - green analog lever gives input in [-1;1] range, while orange buttons are mapped as ON-OFF

Driver implementation is described into 3.7 and its purpose is to convert **ASCII symbols** to encoded values.

Keyboard input is an alternative to joystick: the functionality is the same, except for the input type, which is always pressed/not pressed and not in a continuous range as for analog levels. Keyboard drivers are implemented directly by kernel, so there's no node to interpret ASCII symbols.

4 | Results

As described in the methods (3), experiments were designed for independent layers of the architecture and then for the overall implementation.

Hardware components are equipped with a data logger, which is used to save the data transmitted through the interfaces of the hardware abstraction layer. This data is then processed with MATLAB: velocity profiles are derived from position data and both are displayed for low-level experiments; for high-level experiments, the BRIDGE exoskeleton robot description is loaded from URDF, forward kinematics is calculated to get the end-effector cartesian positions and visualize the motion trajectory and joint configuration data.

4.1. Hardware layer validation

To validate hardware modularity, it will be shown that it's possible to implement an architecture able to adapt to different hardware without requiring to change the rest of the system. This becomes possible thanks to the **hardware abstraction**.

The hardware used for validation consists in the 3 stepper motors and 2 drivers (described in 3.1), mounted on a test-bench.

The hardware abstraction must convert the inputs coming from controllers to the proper signal sent to the drivers. The three steppers have different step-to-radiant factors, different operating currents, the 28BYJ stepper has an integrated gearbox, therefore the actuator-control signal has to be correctly translated in order to transfer its semantic meaning down to the physical layer.

Also, hardware abstraction operates the very first stage of **perception** that is required for higher-level functionalities of the artificial agent. In fact, also the feedback coming from the drivers has to be correctly interpreted (for example encoder information).

The first test consists in controlling the different actuators at the same **velocity**. Thus, a signal carrying semantic value of "velocity" is sent from controllers; it is translated to a serial command in the hardware abstraction; it then reaches the drivers that decode the serial messages and initiate the proper electrical signals, driving the steppers to the correct velocity. The collected data shows that the 3 actuators were qualitatively able to successfully reach the same commanded velocity (fig: 4.1).

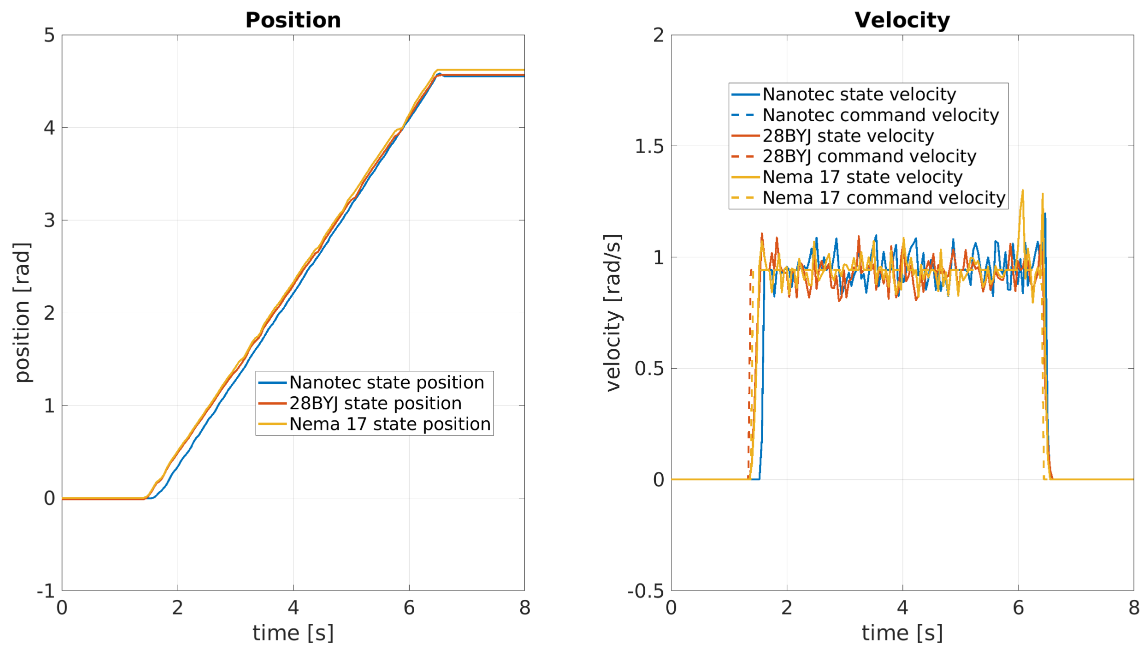


Figure 4.1: Control of different hardware, same commanded velocities (0.94 rad/s)

Modularity is also demonstrated by sending different velocity signals to the various steppers. This shows that the three components of the hardware abstraction were able, in parallel, to correctly translate the controllers' signals for the physical drivers (fig: 4.2).

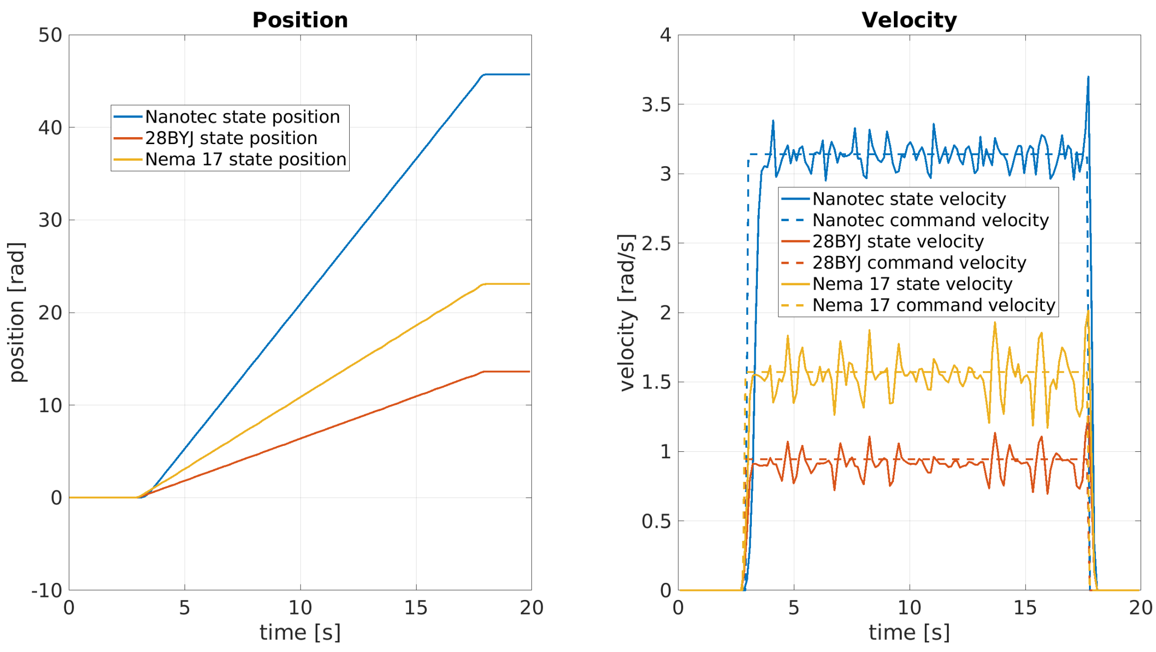


Figure 4.2: Control of different hardware, different commanded velocities (Nanotec stepper: 3.14 rad/s; 28BYJ stepper: 0.94 rad/s; Nema 17 stepper: 1.57 rad/s).

Both drivers support different controlling modalities, therefore other tests were made to validate if the hardware abstraction is able to understand and decode signals with a different semantic. In this case, the same **position** signal was sent from the controllers to the various abstract hardware components: results show that the 3 steppers reached the same correct commanded position, albeit at each one's own speed, confirming the hardware abstraction functionality (fig: 4.3).

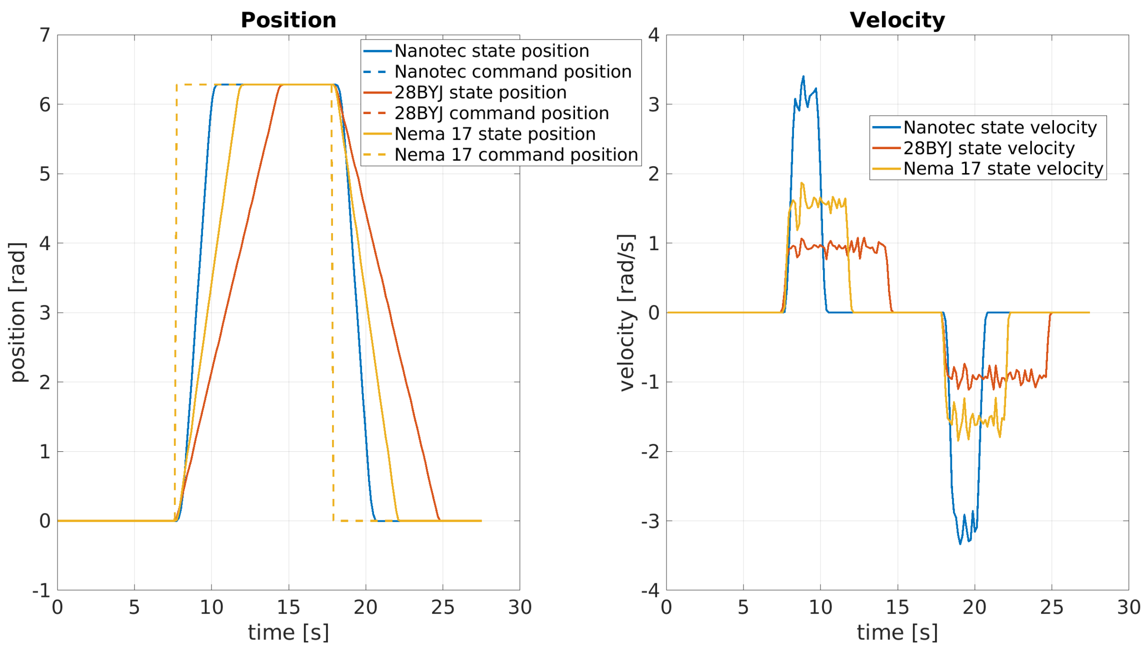


Figure 4.3: Control of different hardware, same commanded position (6.28 rad and back to 0 rad)

Finally, **position** control was used again, but with varying values across the different steppers, with results in accordance with expectations (fig: 4.4).

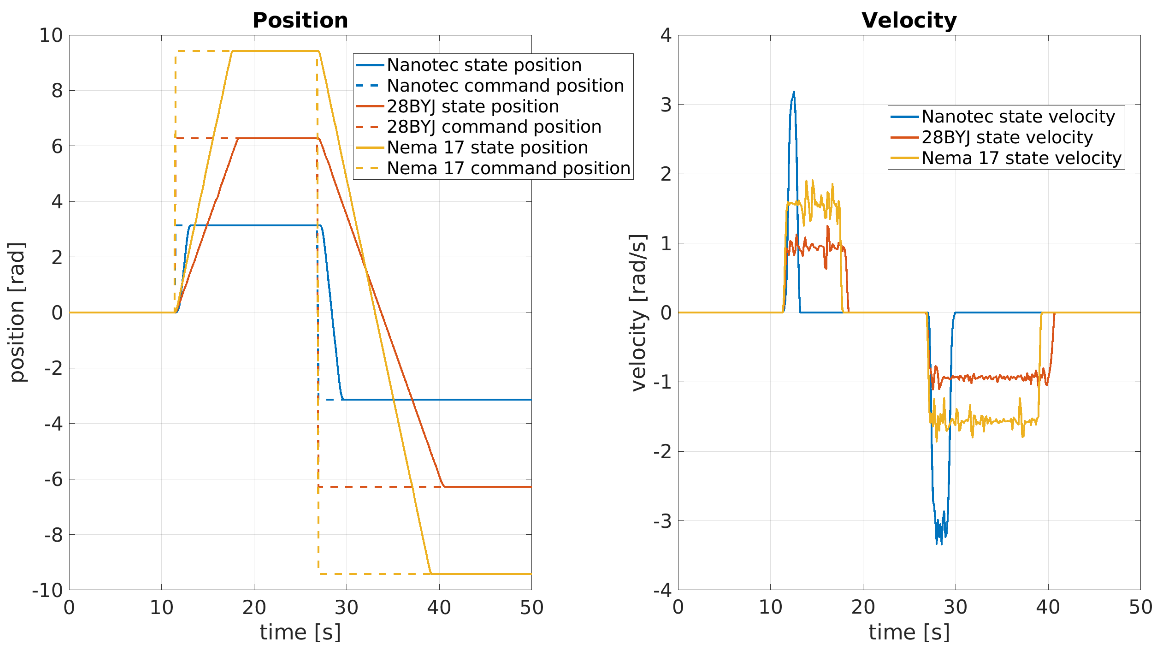


Figure 4.4: Control of different hardware, different commanded position (Nanotec stepper: 3.14 *rad*; 28BYJ stepper: 6.28 *rad*; Nema 17 stepper: 9.42 *rad*

Overall, results demonstrate how multiple hardware from different manufacturers can be connected and controlled in parallel by the same system, thanks to the implementation of an hardware abstraction. This allows the designers of an exoskeleton system to choose the preferred hardware implementation considering the different hardware costs and characteristics, while still maintaining a proper functionality.

4.2. Control layer validation

To demonstrate the modularity of **control** mechanisms, two different control modalities, one open-loop and one close-loop, will be tested against the architecture. In this case, the control layer will be isolated from the rest of the system and tested independently from it, as done before with the hardware abstraction. The bottom part of the architecture down to the actuators will be used in order to provide inputs and feedbacks from the effectors: only the **Nanotec driver** and its corresponding actuator and hardware abstraction will be employed and kept fixed.

The first test is done with a **feed-forward controller**, which is a simple controller directly sending position signals to the hardware abstraction. It doesn't access to any state interface so it doesn't receive feedbacks from the physical layer. The capabilities of the actuators to reach the correct position have already been established before. This time, a temporary obstacle is placed during the execution of the movement. As expected, the actuator couldn't reach the proper position because the controller didn't have any information about the failed execution.

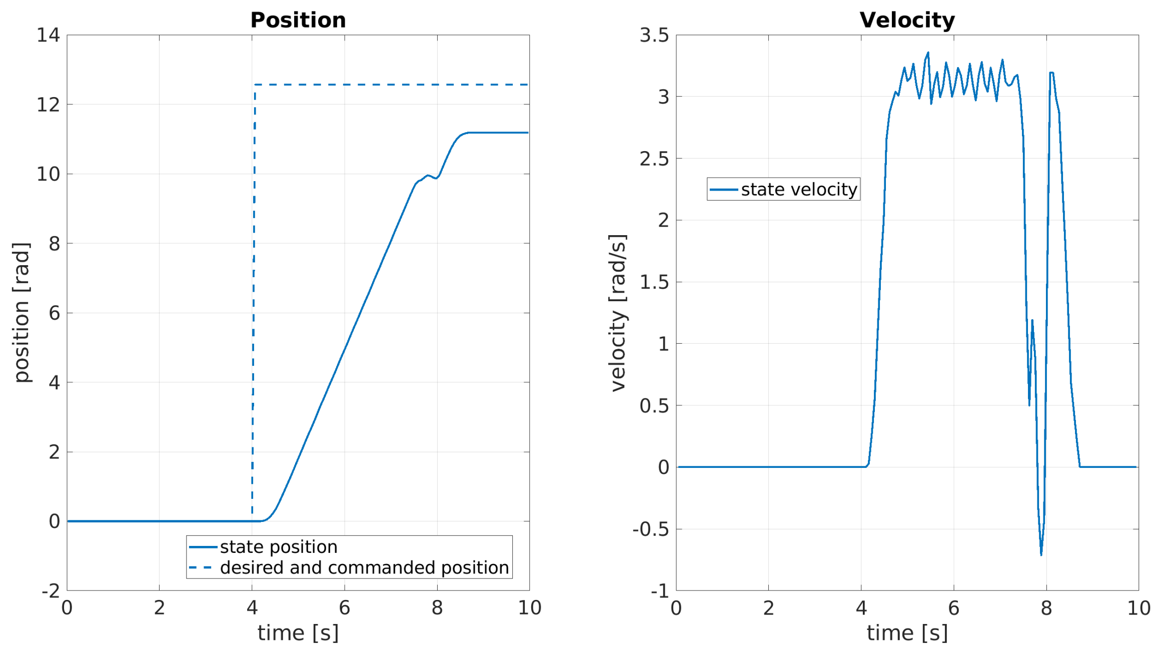


Figure 4.5: Open-loop controller with Nanotec stepper (target position: $12.56 rad$, obstacle around $10\,rad$)

In a different test, a PID **feed-back controller** is used, receiving a position feed-back, a position reference, and sending a velocity command proportional to the error between position and reference. Again, an obstacle is placed during the movement for a short time. However, this time the actuator correctly continued the execution up to the desired position, displaying the proper behavior.

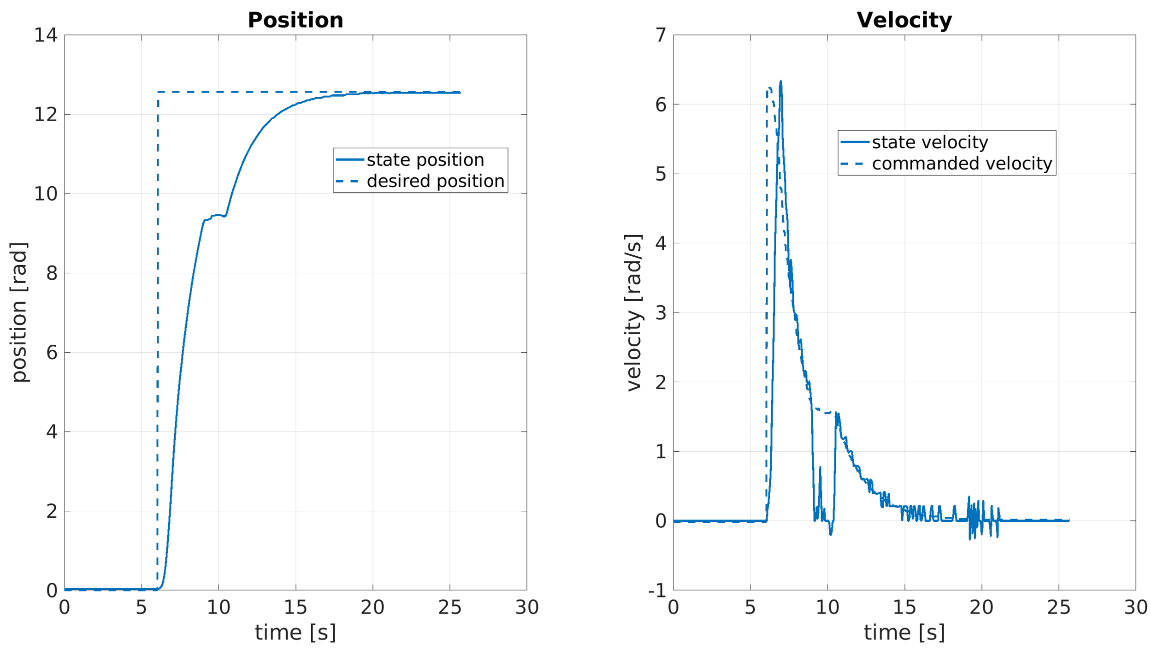


Figure 4.6: Close-loop controller with Nanotec stepper (target position: $12.56rad$, obstacle around $10\,rad$, PID coefficients: $[P = 0.5,\ I = 0,\ D = 0]$)

These qualitative results demonstrate the possibility to modulate the control strategy according to the choice and requirements of the application, independently from the rest of the implementation. As explained, this is crucial for both rehabilitative and assistive scenarios (1.2.2).

4.3. Planning layer validation

To test individually the planning layer, **fake hardware** (3.3.1) is used and joint limits are set as:

	J1	**J2**	**J3**	**J4**
Upper limit	170°	170°	170°	170°
Lower limit	-170°	-170°	-170°	-170°
Speed limit	30 deg/s	30 deg/s	30 deg/s	30 deg/s

Table 4.1: Joint limits for virtual planning and virtual direct control validation

Since fake hardware behaviour is that of an **ideal sensor and actuator** and considering the absence of motor friction, the expectation is a set of end effector coordinates whose variation will be mostly sharp. The first experiment wants to test the correct goal reaching: initial and goal states are defined a-priori as reference for successive comparison:

	J1	**J2**	**J3**	**J4**
Initial state	11°	25°	15°	-53°
Goal state	-30°	-30°	-25°	-50°

Table 4.2: Initial and goal state for virtual planning tests

Initially, no obstacle is placed along the path, so the result is an optimal solution found by planner (fig. 4.7). Z-axis, which has the higher range of motion in the test, shows an abrupt slope change due to perfect control behaviour (a fake hardware simulates the hardware ideally). Goal position is perfectly matched by end-effector.

In the second iteration, an obstacle is placed into `<0.31 -0.04 1.58>` to evaluate planning reaction to environmental change. The acquired data (fig. 4.7) demonstrate that the obstacle is properly avoided, probably without a optimal trajectory, but reaching the goal successfully, which is the aim of the test.

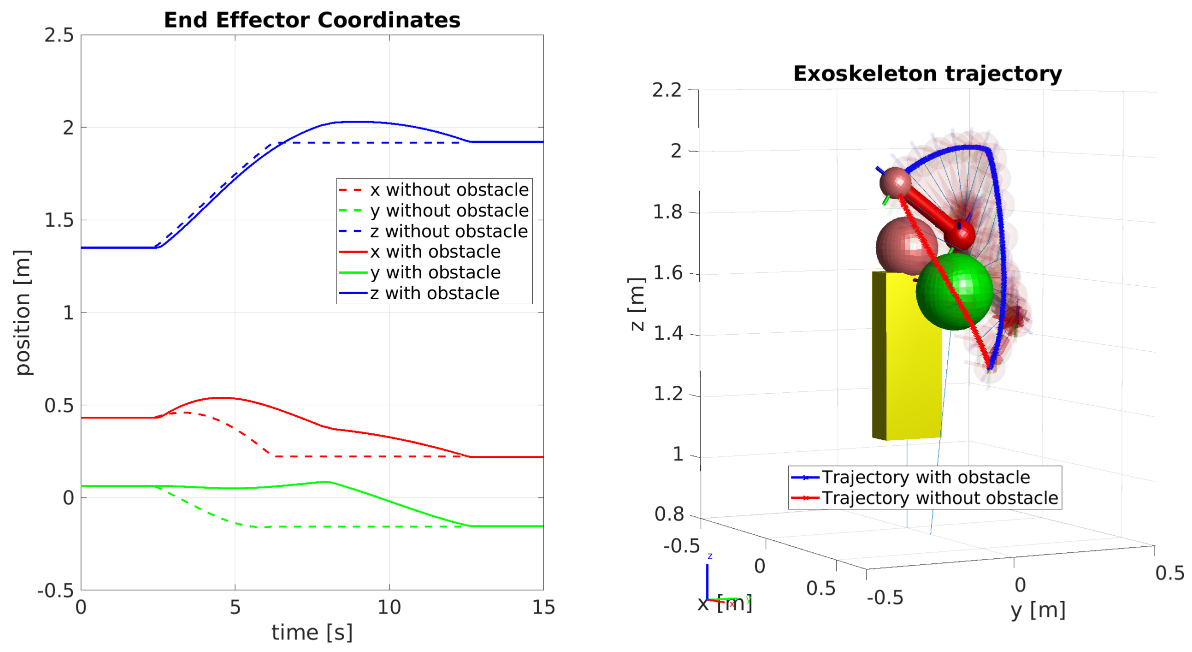


Figure 4.7: Virtual planning test - differences with and without collision detection with spherical obstacle

Hybrid functionalities are tested comparing a reference trajectory with another obtained by intervening with the joystick during movement execution. Initial and goal states are:

	J1	**J2**	**J3**	**J4**
Initial state	6°	8°	9°	-36°
Goal state	23°	31°	17°	8°

Table 4.3: Initial and goal state for hybrid planning tests

The experiment want to test the user's **unaware of switch** between two control modalities and the correctness into replanning to reach the goal state. Green area of figure 4.8 shows how X-Y-Z coordinates changes accordingly to the intervention of direct control by the user. Arm control is exclusive: as soon as joystick commands start to be sent, the framework must stop executing old trajectory and rapidly re-plan when joystick intervention ends. Goal state is correctly reached also in this test.

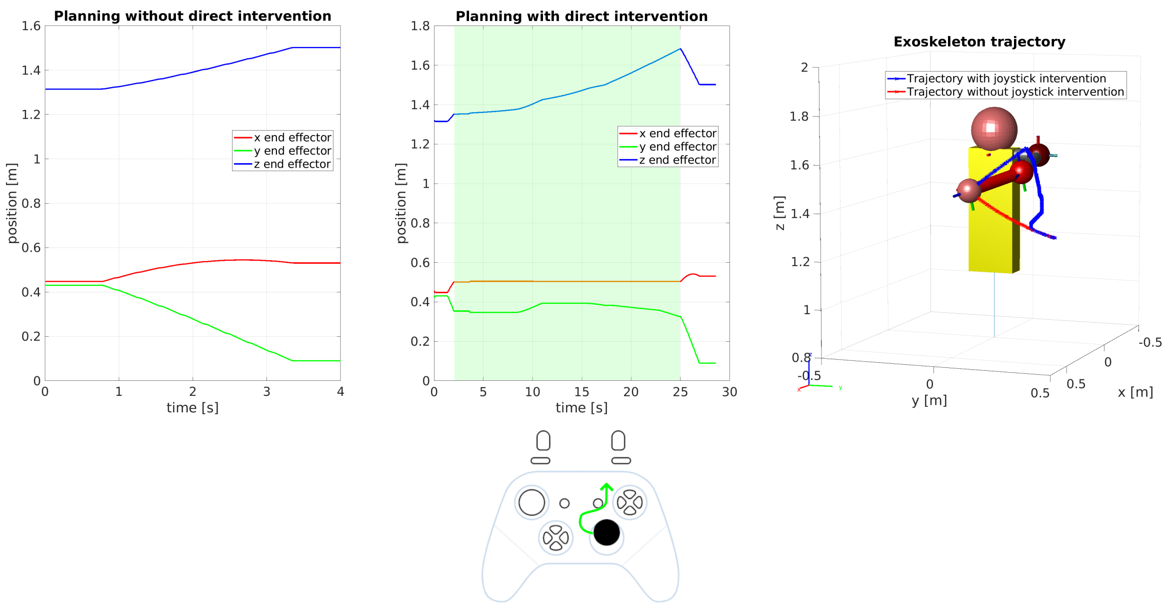


Figure 4.8: Virtual hybrid planning - Left plot: planning without joystick intervention. Middle plot: joystick intervention (green area). Right image: comparison between the two trajectories.

Even though simple planning and hybrid experiments were done with OMPL planner, it is always possible to switch with another one in accordance to necessities and patient's feedback: the calculated trajectories will follow a path based on the new planner construction goal.

4.4. Direct-control validation

In this experiment the end-effector is moved in a cartesian path with right analog lever, in particular in the plane Y-Z. Other planes are possible by using a different combination of joystick buttons. Results (fig. 4.9) point out the almost constancy of X-axis, even though its initial and goal coordinates does not coincide perfectly. This behaviour can occur due to singularities avoidance, indeed the system seems to try bending the elbow in some points of the trajectory to avoid outstretching the arm.

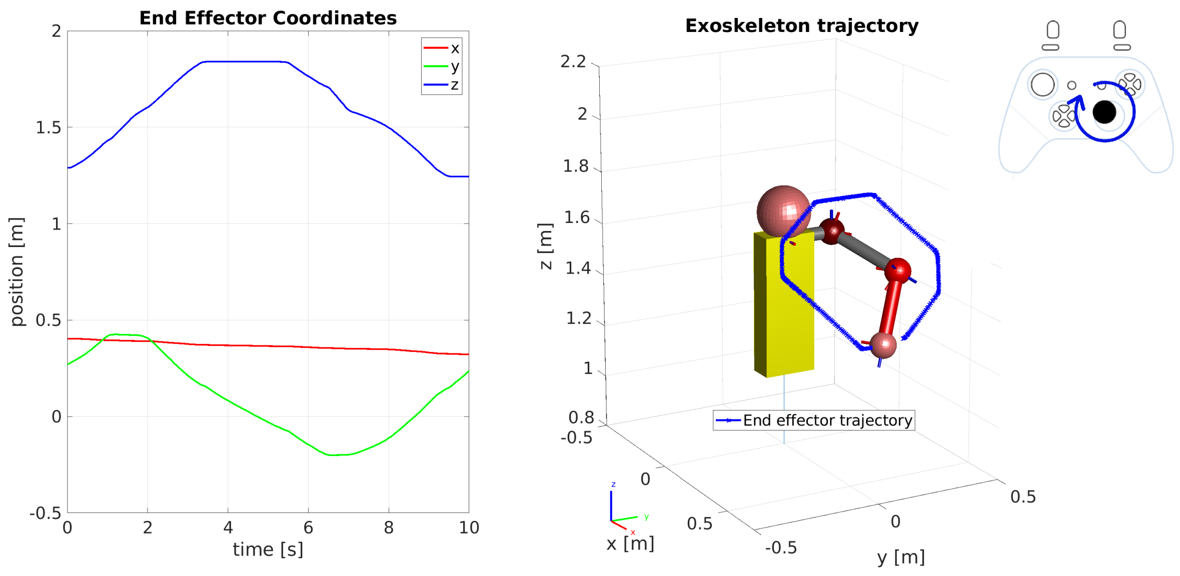


Figure 4.9: Joystick control of end-effector on the Y-Z plane, in a circular trajectory

The potential modularity of this layer arises from the possibility of switching the joystick with another input device, such as computer keyboard or voice control. The importance of this possibility has been well described for the assistive and rehabilitative fields (1.2.3). In this respect, a generic keyboard was tested with similar results and performances to those shown in figure 4.9.

4.5. Vertical validation: BRIDGE exoskeleton

Vertical validation involves the use of the whole architecture in the reference exoskeleton worn by a subject of about 80 Kg. The subject was requested to interact with a computer running virtual environment, and with a joystick to directly control the exoskeleton. Collected data represent joint angular positions measured by the encoders. A Matlab program was then used for further analysis.

A trajectory controller (3.4) with velocity commands and position states and OMPL as planner (2.6.1) are used for all the following experiments. Joint limits are set accordingly to hardware limits, they are considered after motor reductions:

	J1	**J2**	**J3**	**J4**
Upper limit	25°	25°	20°	-10°
Lower limit	-55°	-50°	-40°	-120°
Speed limit	15 deg/s	15 deg/s	15 deg/s	15 deg/s

Table 4.4: Joint limits for vertical validation

The experiments want to establish if:

- User can correctly see a virtualization of current robot configuration in space in a virtual environment.
- User can easily interact with planning functionalities with GUIs or hardware interfaces.
- Path planning is correctly computed in every scenario, with presence or not of obstacles. Cartesian paths must not influence the correctness of planning and have to follow a straight line in a bi-dimensional space.
- Adapters are correctly applied to planned paths, this include fixing start/end positions and time-parameterization.
- Time-parameterized trajectory is well-executed by a trajectory controller, this means that every final joint configuration read by encoders must be equal to reference final position;

For specific experiments such as joystick interface or hybrid control, further goals will be defined in the dedicated sections.

4.5.1. Planning

The experiment is conducted between two known points in the joints configuration. The initial state **P1** and the goal state **P2** are (joint space):

	J1	J2	J3	J4
Initial state	-10°	-30°	0°	-50°
Goal state	20°	20°	0°	-50°

Table 4.5: Initial and goal state for planning test with horizontal obstacle

In the first experiment the subject start from P1 and has to select P2, as planning goal, from a robot visualization widget. Initially, a preview of the movement is shown, then the subject has to accept the planning by pressing a button. Since the time-discrete path is accepted by the user, the trajectory controller start sending commands to actuator drivers. Results (figure 4.10) denote a perfect alignment of goal pose with goal reference P2 and an end-effector position in space which is consistent with goal and smooth. A stepped trajectory would be very unpleasant for the user, indeed a ramp is essential in everyday tasks.

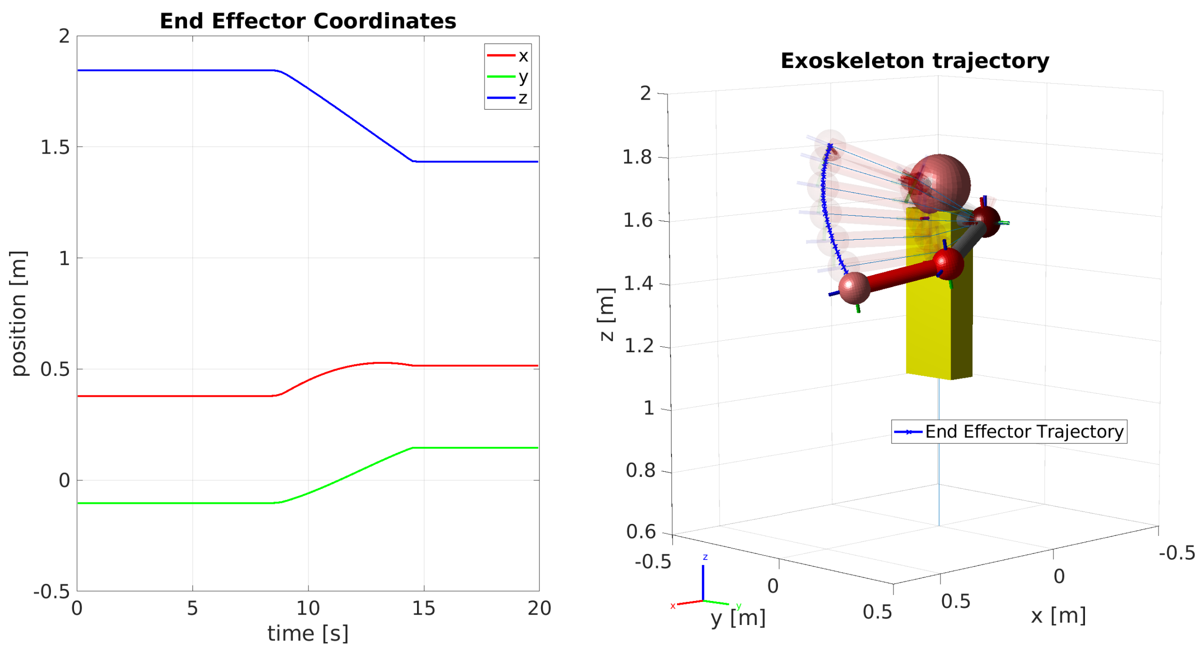


Figure 4.10: Planning without obstacles

An ulterior experiment want to test collision checking in an everyday routine, like moving the arm near an **horizontal table** or with a **water bottle** positioned along the trajectory.

For the table test, initial and final positions are the same P1 and P2 of the previous test. The table intercept the former optimal trajectory, so that an adjustment is necessary by the planner. Since no 3D perception is available, the position of real table is measured manually with a point in the ground as frame reference, then a virtual table is positioned accordingly. The coordinates in the workspace of table's center of gravity are `<0.3, 0, 1.58>`.

The subject is asked to repeat the previous task, checking for the planned trajectory preview and then executing the movement. Results (figure 4.11) show a noticeable deviation from the optimal trajectory of the previous task, this is particularly visible in figure 4.12. Comparing the x,y and z-axis of the end effector in both tasks underline this deviation, especially on y-axis, accordingly to expectations. Also in this case the movement is smooth and never jerky.

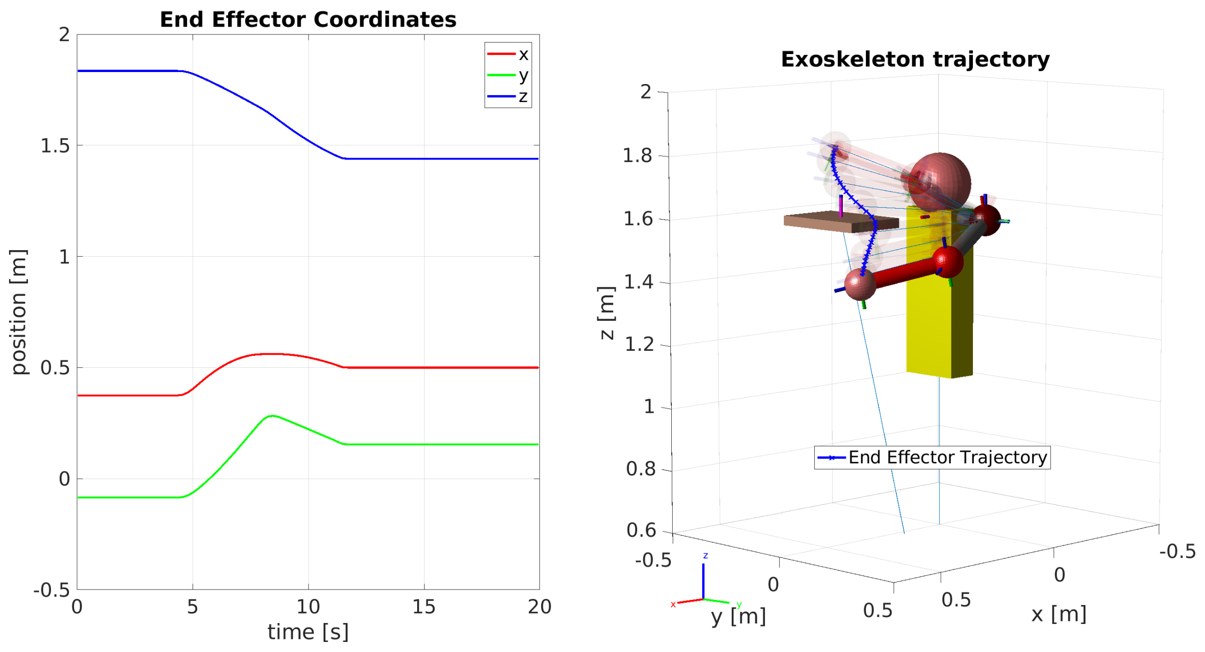


Figure 4.11: Planning with horizontal obstacle

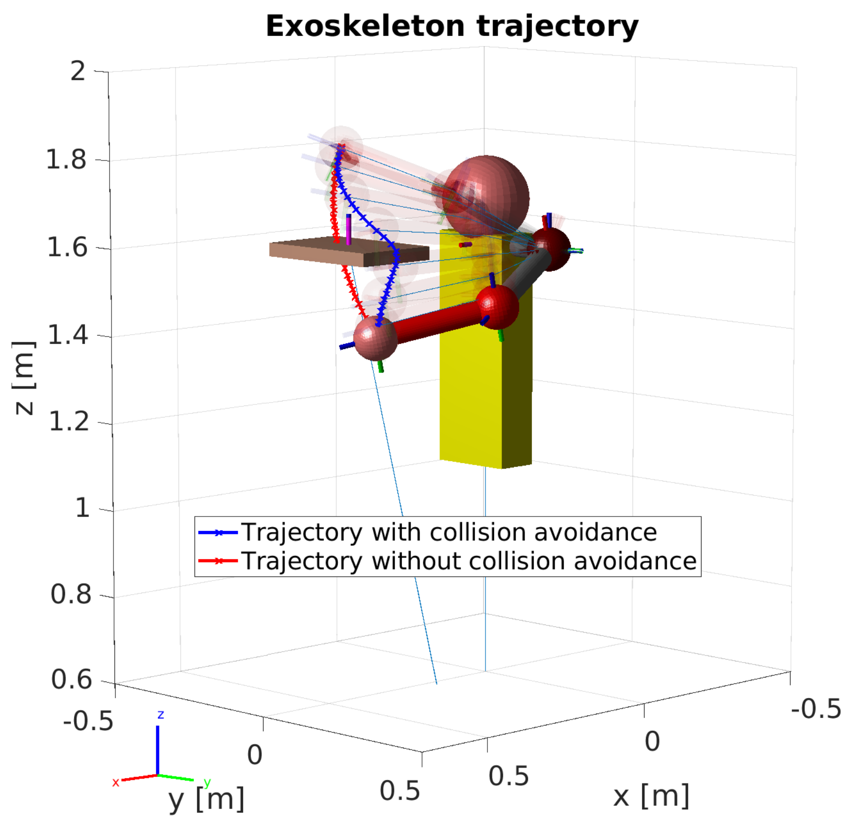


Figure 4.12: Planning - comparison between simple trajectory and collision avoidance

An horizontal movement is best suitable for testing with water bottle; the new initial and goal states are (joint space):

	J1	**J2**	**J3**	**J4**
Initial state	20°	20°	0°	-50°
Goal state	-15°	10°	0°	-30°

Table 4.6: Initial and goal state for planning test with vertical obstacle

As for the table, bottle position is mapped with respect to ground and then imported as a virtual shape to be visualized on the computer: center of gravity x-y-z coordinates are `<0.4 , 0, 1.4>`. After subject planning and execution, data collected confirm the perfect match with goal coordinates and the success in obstacle avoiding (figure 4.13).

Z-axis, indeed, shows a reasonable path to avoid water bottle.

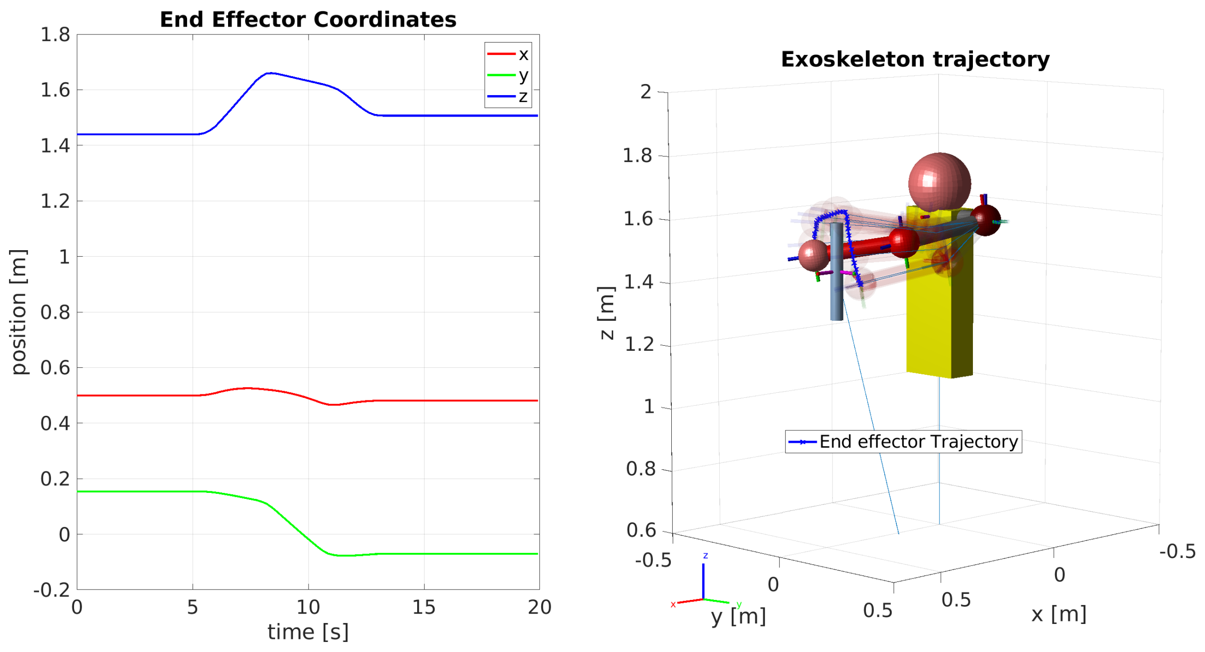


Figure 4.13: Planning with vertical obstacle

The robotic platform and the planner let also compute a Cartesian path in a 2D plane of choice. This results in end effector straight lines with 100% success in every test performed.

4.5.2. Direct control

Planning functionalities are not used if the intention is to move the exoskeleton with direct commands. Further **objectives**, beyond generic ones, are:

- Subject can interact one-to-one with end effector or in the joint configuration space with zero-lag or, at most, without a relevant delay when pressing joystick buttons
- Subject cannot hit an object that is near one of the exoskeleton links. Movements in other directions should be allowed but not ones that could lead to a collision
- System must not stuck into singularities
- For the entire test, subject should not perceive the movement as uncomfortable and imprecise

For this experiment are used the same joystick and software driver of direct control layer validation [4.4].

The tests consists of the 6 possible movements representing the end effector's degrees of freedom (3 translation and 3 rotation).

In the first experiment translational component is tested even though all 6 DoFs can be used together in a complex movement in the workspace. The subject is asked to draw a quadrilateral in the plane Y-Z by moving the right analog level (fig. 4.14). The task can be divided into 4 phases, complementary coupled:

- **Phase A-C**: shoulder's adduction (A) and abduction (C).
- **Phase B-D**: shoulder's flexion (B) and extension (D). Phase D is not linear because framework accounts for joint limits; shoulder's extension involves its abduction, but as defined in the configuration file, shoulder can't abduct more than a defined limit. Used framework compensates by slightly adducting the shoulder, as visible in phase D of figure 4.14. A further limit is reached for J2, this mean that user can't extend more the shoulder (at least not in current joint configuration and with a straight line).

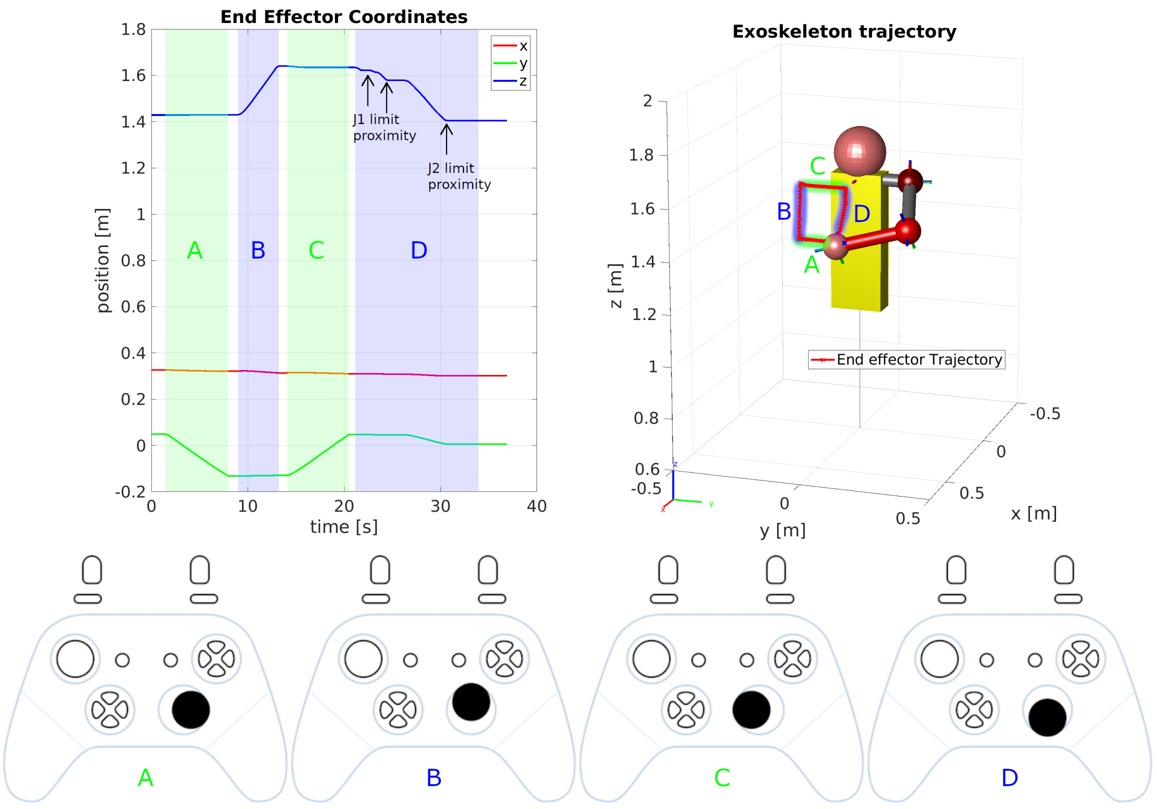


Figure 4.14: Direct Cartesian control through joystick, in the Y-Z plane. Path "D" shows softening of the vertical constraint due to closeness to J1 rotational limit (shoulder horizontal abduction). Also, this downward path stops when J2 limit is reached (shoulder extension)

The second experiment want to try the rotational component: arm's end effector can't move linearly, so for a rotation on the X-axis shoulder's flexion and abduction components are predominant (fig. 4.15). Results show that end effector is perfectly still on the 3 axis, the only active component is a torque around X-axis.

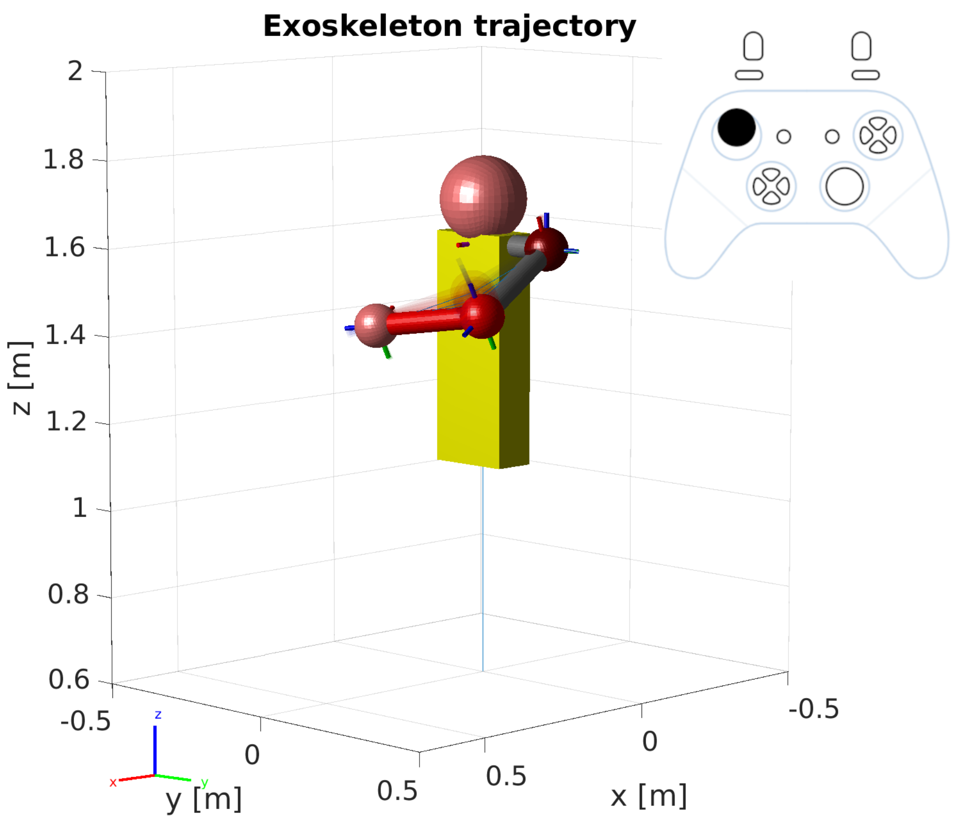


Figure 4.15: Direct control of twist angle (transparent trace of movement)

If the user want to control joints individually, joystick's buttons are mapped for this purpose. Since buttons are ON/OFF, respective joint will go at maximum speed possible, defined in the configuration file. Subject is asked to move J1 to bring the end effector near the trunk. As result (fig. 4.16), Z-axis doesn't change due to joint 1 construction, whose rotation is defined around Z-axis.

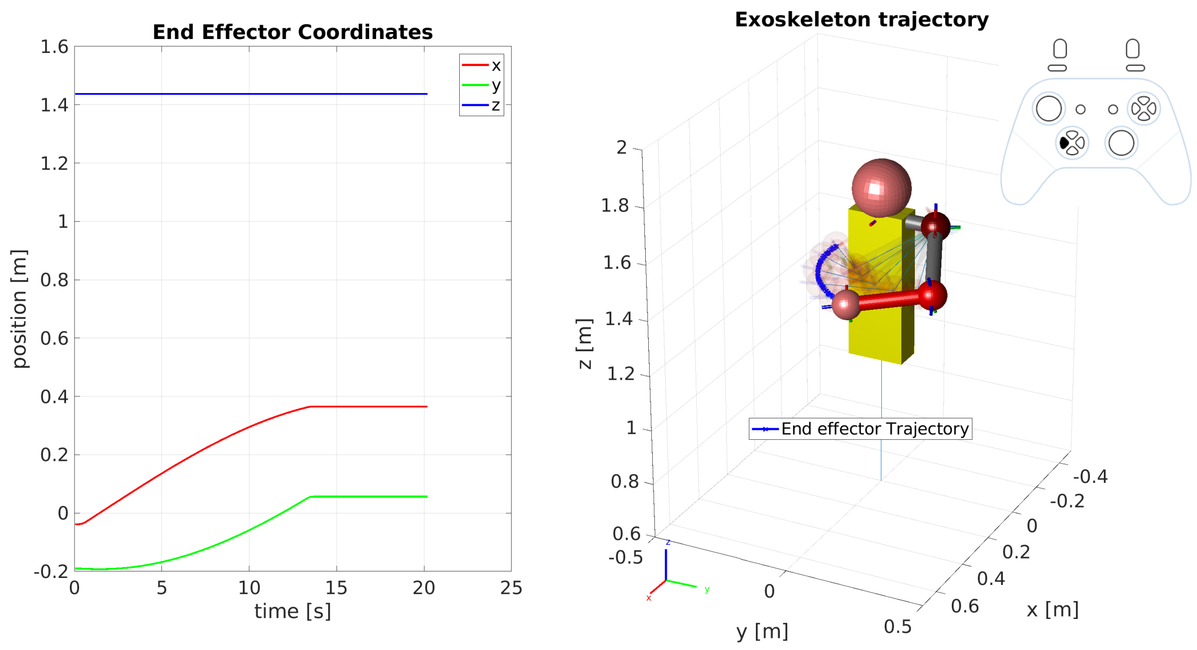


Figure 4.16: Direct control of the single joint j1

Further tests were conducted about the **collision avoidance** by placing a virtual obstacle and asking the subject to driving into. In none of the test it was possible for him to continue the movement in the object direction if it is near to exoskeleton links.

The last test about direct control concern the **singularity avoidance**: none of previous test led to singularity stuck because the framework always decrease the speed if near one of them or, as last countermeasure, stop the exoskeleton motion.

All previous direct-control tests were conducted using a generic joystick, but the same results have been obtained with a **computer keyboard**; the only difference is notable when trying to control the movement speed because keyboard buttons are ON/OFF and can't be mapped as analog levels to ranged inputs. Independently from using keyboard or joystick a ramp is associated to each trajectory but, if the user want to move the end effector at lower speed, with keyboard is impossible, the exoskeleton will move always at maximum speed the user set.

4.5.3. Hybrid planning

Hybrid planning tests want to investigate the possibility of subject to fast switch from planned path execution to direct control without delay or jerky movements. Such functionality can be helpful if, during a planned movement, user notices an obstacle on the path and, for security reasons, has to change the trajectory to avoid it. Since the software workflow is significantly different, new **objectives** are:

- Subject can plan and execute a movement as good as previous tests (4.5.1)
- Subject can control directly the exoskeleton with cartesian paths and every joint individually with the same results obtained in 4.5.2
- When moving along a planned trajectory to a goal pose, subject must be able to intervene with direct control
- As soon as joystick intervene, previous planned movement must **pause** and resume only when no more direct commands are sent by the subject
- When the initial trajectory **resume**, planner has to fast replan and correctly reaching the initial goal pose

Two new test positions are established: they are the same used for virtual hybrid validation (table 4.3). A reference trajectory is acquired (fig. 4.17), subject is able to move from initial state to goal state by pressing a button. Smoothness and precision is comparable to the ones of planning tests (4.5.1).

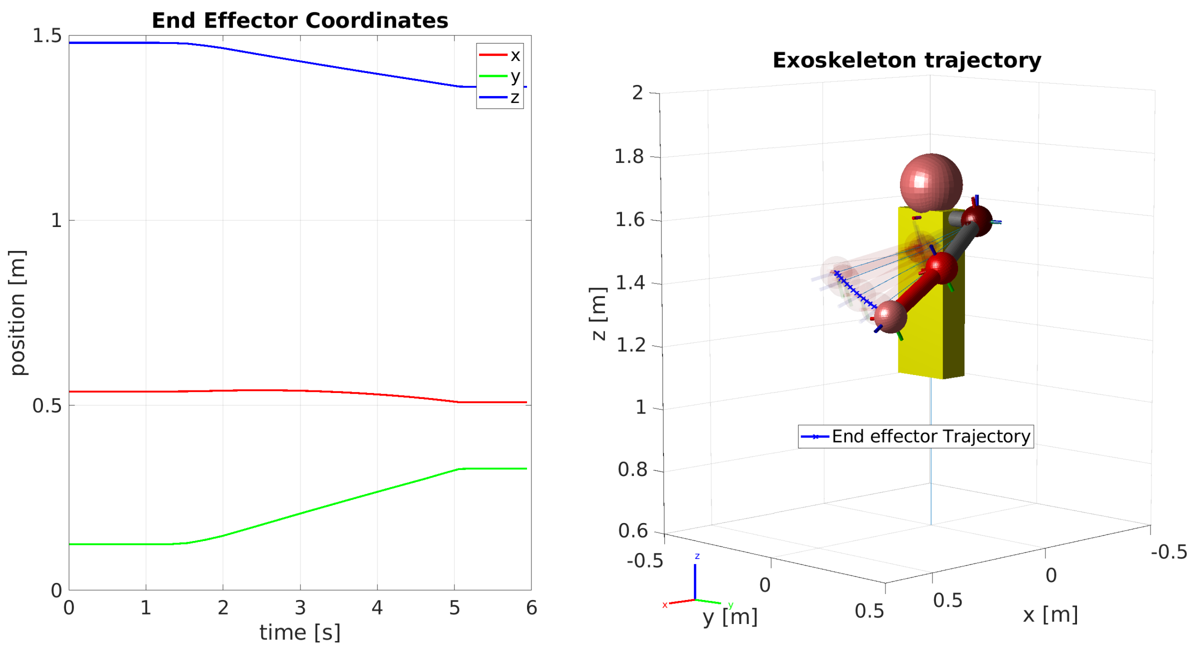


Figure 4.17: Hybrid planning without joystick intervention

The same exercise is repeated interfering with joystick during the trajectory execution (fig. 4.18) simulating an obstacle on the path. Three phases can be distinguished:

- Phase A: planner start executing trajectory by moving end effector in the direction of final pose
- Phase B: Subject use joystick to direct command the end effector in a cartesian path, pausing the execution of previous movement. From this point on, exoskeleton will execute only the joystick commands
- Phase C: Subject stop sending joystick commands, framework fast replan and set the new trajectory to reach defined goal pose

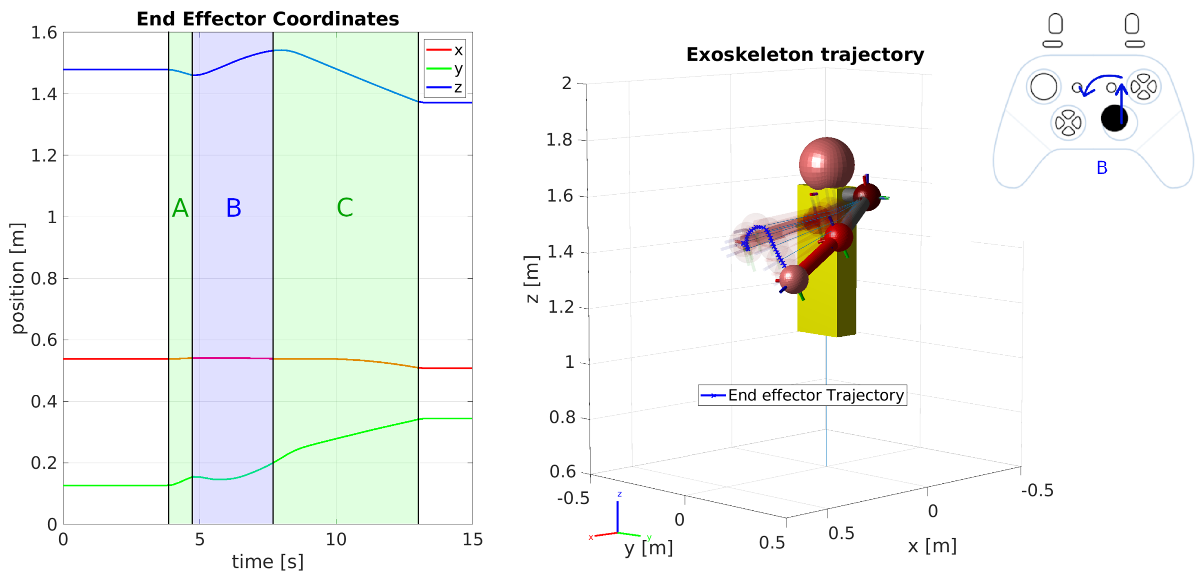


Figure 4.18: Hybrid planning with joystick intervention. A: start execution of planned trajectory; B: joystick direct control; C: continuation of planning towards end point

As effect of joystick cartesian commands, only Z-axis is affected by the trajectory change, anyway, user can perform end effector complex movements without any issue. In figure 4.19 is clearly visible the correspondence of initial and final positions of reference and with-interference exercises.

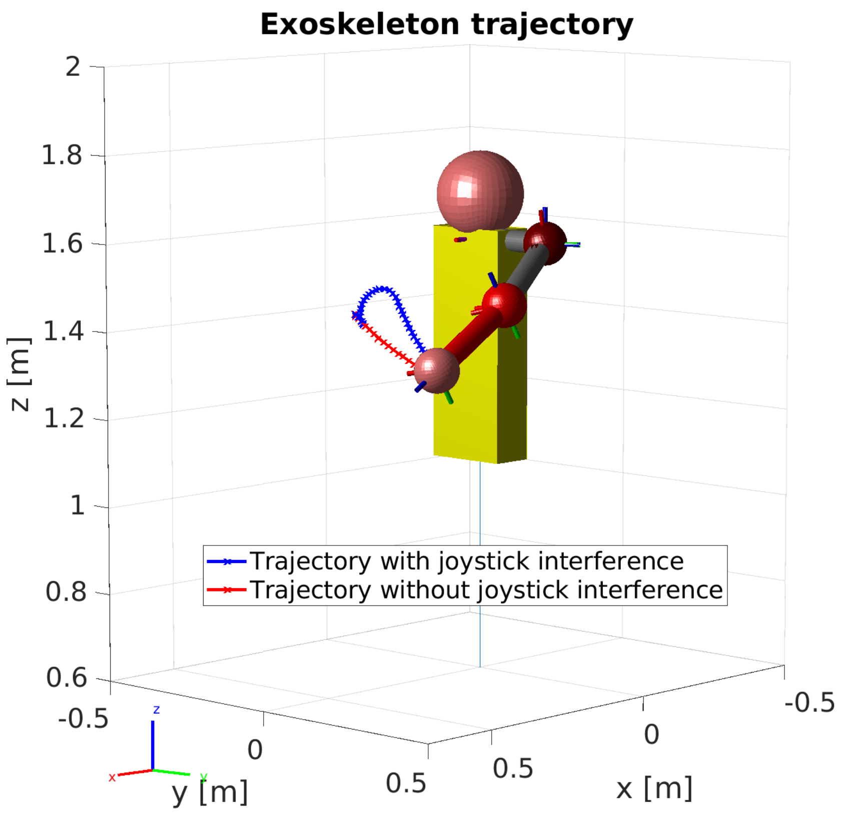


Figure 4.19: Hybrid planning: trajectories comparison

As for direct control tests (4.5.2) subject can intervene also with a computer keyboard with the same exact results shown.

5 | Conclusions

This project attempted to demonstrate the possibility to implement an architecture for exoskeletons, either rehabilitative or assistive, following recent robotic standards and an overall philosophy of system modularity. All concepts were followed while keeping the overall analysis tied to the biomedical field, in particular the topic of motor disability.

A software architecture was implemented using widespread tools, libraries and systems. The architecture was given a basic structure, still maintaining all the main functionalities required by the problem statements, but with a simplified implementation. All the different modular elements of the system can be specialized for the requirements that each scenario may have, and this work provides some concrete examples and a general line of thought on how to do that.

Specifically, this project focused on the customization and cooperation of the different elements that compose an exoskeleton or human-robot system, such as decision system, hardware, control system, high-level functionalities and planning, intention-detection mechanisms, human-machine interfaces and user-feedback. Qualitative results demonstrated that an overall customization is possible to obtain, still preserving a robust functionality for each of the modules of the system.

The adoption of a standardized software and architecture design philosophy in rehabilitation and assistance can help different realities and fields join together for a common objective. Partitioning a complex problem into simpler subsystems allows people with different backgrounds and knowledge to work separately and concentrate their expertise on tasks they are proficient in: the concept of a modular architecture is to fuse all these individual elements into a single system where they can be easily switched, composed together, customized and re-used, avoiding to re-design the whole system over and over, and to waste physical and mental resources.

Of course, a modular architecture is not the solution to all problems. Implementing from scratch a complex system made up of multiple simultaneous processes and mechanisms to make them dialogue can make one lose the primary purpose of the work; on the other hand, learning a standardized framework that implements these functions can be particularly

difficult and time consuming, especially for people whose background is more theoretical or broad-spectrum.
In this regard, it is useful to underline the importance of a well-documented system, at code-level, conceptual-level and use-case-level. A thorough documentation can help developers to construct and adapt the system faster and more easily; also, documentation can help users and caregivers understand better the main functionalities of the system and how it interacts with the environment. For this project, a documentation was written both for the software packages and for the conceptual framework that has been studied in order to implement them [9]. In addition, a system has been developed to automatically generate core packages and so, speeding up the development process [8].

Another aspect is that not all functions are better achieved if implemented using the same framework: that's why a middleware connecting and supporting different developing tools is important. Moreover, some low-level, reactive control systems are much more efficient if implemented closer to the hardware, for example on integrated circuit boards, rather than on an higher-level architecture. However, systems that are designed to stay fixed may display a greater connection between the system components, as they can be optimally designed for that particular instance. In many cases, an integrated design can also help lowering the prices, mostly for hardware, and this is one of the main factors in the assistive scenario. Therefore, some trade-offs are always present between the different design philosophies, and the choice on how to implement each functionality should depend on the context.

5.1. Future developments

In the future, this proof of concept and architecture design will be used by the Neuro-Engineering And medical Robotic Laboratory (NEARLab) for projects currently under development. In order to finalize a project built in this way, attention must be paid to several points.

First, the **robot description** must be more accurate, including meshes for both visualization and collision, as close as possible to the real geometries of the system, and correct dynamic properties. This is tricky to implement in a ROS environment, as the robot description is made in a markup language: other robotic platforms such as Orocos allow to define a 3D model of the robot directly into the system. Anyway, there are some possibilities to ease the robot description implementation: for example, a SolidWorks plugin allows to export the URDF from a 3D model made on a graphic interface, but it has to be constructed properly in kinematic terms. Also, alternatives to URDF description should be researched for soft structures, that are seldom used in the industrial field, but more and more studied by the scientific community.

Secondly, the **hardware abstraction** layer must be expanded to other communication protocols, such as EtherCAT, and other types of actuators like brushless and pneumatic effectors that are gaining interest in research for artificial muscles. Moreover, it's possible to construct more distributed architectures, where the main framework runs both on a main computer and on integrated boards: the latter may implement more reactive control mechanisms, using a lower level of communication with the hardware of the exoskeleton.

Furthermore, there's the need to transfer **controllers** already developed for rehabilitation into the system: as said, this is not immediate as the current version doesn't support nested control structures, but can be solved by wrapping controllers into a unique process/node of the system. Another feature under development is the asynchrony between controllers, that are currently all dependent on the Controller Manager's unique clock; anyway, it's possible to implement multiple Controller Managers each running independently their child controllers.

Another improvement concerns the **planning** layer, which currently uses planners designed for industrial robotics: a more physiological model to calculate trajectories such as "learning by demonstration" is necessary for medical applications, but currently there is no solution robust enough to be implemented. Also, other mechanisms of **intention detection** and user-feedback should be tested and assessed.

The final hope is that this work could be an useful starting point for a more outlined methodology on how to construct a modular human-robot system, for both clinical and assistive scenarios.

Bibliography

[1] T. M. B. A. Sensors and actuation technologies in exoskeletons: A review. *Sensors (Basel)*, 2022.

[2] P. Beeson and B. Ames. Trac-ik: An open-source library for improved solving of generic inverse kinematics. *International Conference on Humanoid Robots*, 2015.

[3] A. Cieza, K. Causey, K. Kamenov, S. W. Hanson, S. Chatterji, and T. Vos. Global estimates of the need for rehabilitation based on the global burden of disease study 2019: a systematic analysis for the global burden of disease study 2019. *Lancet*, 2020.

[4] D. Coleman, I. A. Șucan, S. Chitta, and N. Correll. Reducing the barrier to entry of complex robotic software: a moveit! case study. *Journal of Software Engineering for Robotics*, 2014.

[5] D. Comini and D. d'Arenzo. arduino_stepper_serial_driver, 2022. URL `https://github.com/Assistive-Exoskeleton/arduino_stepper_serial_driver`.

[6] J. J. Craig. *Introduction to Robotics: Mechanics and Control.* Pearson Higher Education, 3rd edition, 2014. ISBN 978-1-292-04004-2.

[7] S. Dalla Gasperina, L. Roveda, A. Pedrocchi, F. Braghin, and M. Gandolla. Review on patient-cooperative control strategies for upper-limb rehabilitation exoskeletons. *Front Robot AI*, 8:745018, Dec. 2021.

[8] D. d'Arenzo and D. Comini. Ros2_templates, 2022. URL `https://github.com/Assistive-Exoskeleton/Templates_ROS2`.

[9] D. d'Arenzo and D. Comini. Ros notes, 2022. URL `https://assistive-exoskeleton.github.io/ROS_notes/index.html`.

[10] L. A. V. der Heide, B. van Ninhuijs, A. Bergsma, G. J. Gelderblom, D. J. van der Pijl, and L. P. de Witte. An overview and categorization of dynamic arm supports for people with decreased arm function. *Prosthetics and orthotics international*, 2014.

[11] G. DH., V. M., and D. C. Satisfaction and perceptions of long-term manual

wheelchair users with a spinal cord injury upon completion of a locomotor training program with an overground robotic exoskeleton. *Disability and Rehabilitation: Assistive Technology*, 2019.

[12] S. A. Directorate General for Employment and E. Opportunities. Quality in and equality of access to healthcare services. Brussels, 2008.

[13] A. Elkady and T. Sobh. Robotics middleware: A comprehensive literature survey and attribute-based bibliography. *Journal of robotics*, 2012.

[14] L. G. et al. Development and psychometric properties of the family life interview. *Journal of Applied Research in Intellectual Disabilities*, 2010.

[15] S. W. et al. Use of the icf model as a clinical problem-solving tool in physical therapy and rehabilitation medicine. *Physical Therapy*, 2002.

[16] M. Gandolla, A. Costa, L. Aquilante, M. Gfoehler, M. Puchinger, F. Braghin, and A. Pedrocchi. Bridge - behavioural reaching interfaces during daily antigravity activities through upper limb exoskeleton: Preliminary results. *IEEE International conference on rehabilitation robotics*, 2017.

[17] M. Gandolla, S. Dalla Gasperina, V. Longatelli, A. Manti, L. Aquilante, M. G. D'Angelo, E. Biffi, E. Diella, F. Molteni, M. Rossini, M. Gföhler, M. Puchinger, M. Bocciolone, F. Braghin, and A. Pedrocchi. An assistive upper-limb exoskeleton controlled by multi-modal interfaces for severely impaired patients: development and experimental assessment. *Robotics and Autonomous Systems*, 143:103822, 2021. ISSN 0921-8890. doi: https://doi.org/10.1016/j.robot.2021.103822. URL `https://www.sciencedirect.com/science/article/pii/S092188902100107X`.

[18] R. Gopura, D. Bandara, K. Kiguchi, and G. Mann. Developments in hardware systems of active upper-limb exoskeleton robots: A review. *Robotics and Autonomous Systems*, 75:203–220, 2016. ISSN 0921-8890. doi: https://doi.org/10.1016/j.robot.2015.10.001. URL `https://www.sciencedirect.com/science/article/pii/S0921889015002274`.

[19] M. A. Gull, S. Bai, and T. Bak. A review on design of upper limb exoskeletons. *Robotics*, 2020.

[20] ISO 9241-210. Ergonomics of human-system interaction, part 210: Human-centred design for interactive systems, 2019.

[21] W. J., P. C., and B. J. A survey of stakeholder perspectives on exoskeleton technology. *Journal of NeuroEngineering and Rehabilitation*, 2014.

[22] M. M. H. P. Janssen, J. Lobo-Prat, A. Bergsma, E. Vroom, and workshop participants. 2nd workshop on upper-extremity assistive technology for people with duchenne: Effectiveness and usability of arm supports irvine, usa, 22nd-23rd january 2018. *NMD*, 2019.

[23] O. Just, F. Feedforward model based arm weight compensation with the rehabilitation robot armin. *International Conference on Rehabilitation Robotics*, 2017.

[24] V. Klamroth-Marganska. Stroke rehabilitation: Therapy robots and assistive devices. *Springer*, 2018.

[25] P. Langhorne, F. Coupar, and A. Pollock. Motor recovery after stroke: a systematic review. *The Lancet Neurology*, 2009.

[26] R. Lee. The demographic transition: three centuries of fundamental change. *The Journal of Economic Perspectives*, 2003.

[27] J. Li, S. Li, L. Zhang, C. Tao, and R. Ji. Position solution and kinematic interference analysis of a novel parallel hip-assistive mechanism. *Mech. Mach. Theory*, 2018.

[28] J. Lobo-Prat, P. N. Kooren, A. H. Stienen, J. L. Herder, B. F. Koopman, and P. H. Veltink. Non-invasive control interfaces for intention detection in active movement-assistive devices. *Journal of neuroengineering and rehabilitation*, 2014.

[29] F. Lulu. serialib, 2021. URL `https://github.com/imabot2/serialib`.

[30] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66): eabm6074, 2022. doi: 10.1126/scirobotics.abm6074. URL `https://www.science.org/doi/abs/10.1126/scirobotics.abm6074`.

[31] A. Mohebbi. Human-robot interaction in rehabilitation and assistance: a review. *Current Robotics Reports*, 2020.

[32] U. Nations. Article 26 – habilitation and rehabilitation. Convention on the Rights of Persons with Disabilities (CRPD).

[33] P. Neumann. Communication in industrial automation—what is going on? *Science Direct*, 2007.

[34] C. Nguiadem, M. Raison, and S. Achiche. Motion planning of upper-limb exoskeleton robots: A review. *Applied Sciences*, 2020.

[35] W. H. Organization. Disability and health. Geneva, 2001.

[36] W. H. Organization and W. Bank. World report on disability, 2011.

[37] F. Posteraro, S. Mazzoleni, S. Aliboni, B. Cesqui, A. Battaglia, P. Dario, and S. Micera. Robot-mediated therapy for paretic upper limb of chronic patients following neurological injury. *J Rehabil Med*, 2009.

[38] T. Proietti, V. Crocher, A. Roby-Brami, and N. Jarrassé. Upper-limb robotic exoskeletons for neurorehabilitation: A review on control strategies. *IEEE Reviews in Biomedical Engineering*, 9:4–14, 2016.

[39] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson College Div, 3rd edition, 2010. ISBN 978-0136042594.

[40] A. Sen and L. Sen. *The Idea of Justice*. Harvard University Press, 2009. ISBN 9780674036130. URL `https://books.google.it/books?id=enqMd_ze6RMC`.

[41] K. Suzumori and A. A. Faudzi. Trends in hydraulic actuators and components in legged and tough robots: a review. *Advanced Robotics*, 2018.

[42] J. B. West and A. M. Luks. *West's Pulmonary Pathophysiology: The Essentials*. Lippincott Williams and Wilkins, 9th edition, 2017.

[43] B. H. Çorak and F. Y. Okay. Comparative analysis of iot communication protocols. *Computers and Communications*, 2018.

[44] I. A. Şucan, M. Moll, and L. E. Kavraki. The open motion planning library. *IEEE Robotics and Automation Magazine*, 2012.

A | Appendix

Motion planner	Parameters	Description
EST	range: 0.0	Max motion added to tree
	goal_bias: 0.05	When close to goal select goal, with this probability
LBKPIECE	range: 0.0	Max motion added to tree
	border_fraction: 0.9	Fraction of time focused on boarder
	min_valid_path_fraction: 0.5	Accept partially valid moves above fraction
BKPIECE	range: 0.0	Max motion added to tree
	border_fraction: 0.9	Fraction of time focused on boarder
	min_valid_path_fraction: 0.5	Accept partially valid moves above fraction
	failed_expansion_score_factor: 0.5	When extending motion fails, scale score by factor
KPIECE	range: 0.0	Max motion added to tree
	border_fraction: 0.9	Fraction of time focused on boarder
	min_valid_path_fraction: 0.5	Accept partially valid moves above fraction
	failed_expansion_score_factor: 0.5	When extending motion fails, scale score by factor
	goal_bias: 0.05	When close to goal select goal, with this probability
RRT	range: 0.0	Max motion added to tree
	goal_bias: 0.05	When close to goal select goal, with this probability
RRTConnect	range: 0.0	Max motion added to tree
RRTstar	range: 0.0	Max motion added to tree
	goal_bias: 0.05	When close to goal select goal, with this probability
	delay_collision_checking: 1	Stop collision checking as soon as C-free parent found
TRRT	range: 0.0	Max motion added to tree
	goal_bias: 0.05	When close to goal select goal, with this probability
	max_states_failed: 10	hen to start increasing temp
	temp_change_factor: 2.0	how much to increase or decrease temp
	min_temperature: 10e-10	lower limit of temp change
	init_temperature: 10e-6	initial temperature
	frountier_threshold: 0.0	dist new state to nearest neighbor to disqualify as frontier
	frountierNodeRatio: 0.1	1/10, or 1 nonfrontier for every 10 frontier
	k_constant: 0.0	value used to normalize expresssion
FMT	num_samples: 1000	number of states that the planner should sample
	radius_multiplier: 1.1	multiplier used for the nearest neighbors search radius
	nearest_k: 1	use Knearest strategy
	cache_cc: 1	use collision checking cache
	heuristics: 0	activate cost to go heuristics
	extended_fmt: 1	activate the extended FMT*: adding new samples if planner does not finish successfully
PRM	max_nearest_neighbors: 10	Use k nearest neighbors

Table A.1: OMPL implemented planners and respective parameters

Motion planner	Parameters	Description
BFMT	num_samples: 1000	number of states that the planner should sample
	radius_multiplier: 1.0	multiplier used for the nearest neighbors search radius
	nearest_k: 1	use Knearest strategy
	balanced: 0	Exploration strategy: balanced true expands one tree every iteration. False will select the tree with lowest maximum cost to go
	optimality: 1	Termination strategy: optimality true finishes when the best possible path is found. Otherwise, the algorithm will finish when the first feasible path is found
	heuristics: 1	activate cost to go heuristics
	cache_cc: 1	use collision checking cache
	extended_fmt: 1	activate the extended FMT*: adding new samples if planner does not finish successfully
STRIDE	range: 0.0	Max motion added to tree
	goal_bias: 0.05	When close to goal select goal, with this probability
	use_projected_distance: 0	whether nearest neighbors are computed based on distances in a projection of the state rather distances in the state space itself
	degree: 16	desired degree of a node in the Geometric Near-neightbor Access Tree (GNAT)
	max_degree: 18	max degree of a node in the GNAT
	min_degree: 12	min degree of a node in the GNAT
	max_pts_per_leaf: 6	max points per leaf in the GNAT
	estimated_dimension: 0.0	estimated dimension of the free space
	min_valid_path_fraction: 0.2	Accept partially valid moves above fraction
BiTRRT	range: 0.0	Max motion added to tree
	temp_change_factor: 0.1	how much to increase or decrease temp
	init_temperature: 100	initial temperature
	frountier_threshold: 0.0	dist new state to nearest neighbor to disqualify as frontier
	frountier_node_ratio: 0.1	1/10, or 1 nonfrontier for every 10 frontier
	cost_threshold: 1e300	the cost threshold. Any motion cost that is not better will not be expanded
LBTRRT	range: 0.0	Max motion added to tree
	goal_bias: 0.05	When close to goal select goal, with this probability
	epsilon: 0.4	optimality approximation factor
BiEST	range: 0.0	Max motion added to tree
ProjEST	range: 0.0	Max motion added to tree
	goal_bias: 0.05	When close to goal select goal, with this probability
LazyPRM	range: 0.0	Max motion added to tree
SPARS	stretch_factor: 3.0	roadmap spanner stretch factor. multiplicative upper bound on path quality. It does not make sense to make this parameter more than 3
	sparse_delta_fraction: 0.25	delta fraction for connection distance. This value represents the visibility range of sparse samples
	dense_delta_fraction: 0.001	delta fraction for interface detection
	max_failures: 1000	maximum consecutive failure limit
SPARStwo	stretch_factor: 3.0	roadmap spanner stretch factor. Multiplicative upper bound on path quality. It does not make sense to make this parameter more than 3
	sparse_delta_fraction: 0.25	delta fraction for connection distance. This value represents the visibility range of sparse samples
	dense_delta_fraction: 0.001	delta fraction for interface detection
	max_failures: 5000	maximum consecutive failure limit

Table A.2: OMPL implemented planners and respective parameters

List of Figures

List of Tables

Acknowledgements

Ringraziamo la professoressa Alessandra Pedrocchi per l'opportunità offertaci nello svolgimento di questo lavoro e la professoressa Marta Gandolla per la cura e la supervisione del progetto. Un ringraziamento particolare a Stefano e Mattia per averci accompagnato e seguito in questo percorso con grande professionalità, dedizione e pazienza e per essere stati essenziali per la riuscita di questo progetto di tesi. Ringraziamo anche tutti i ragazzi e le ragazze del NearLab per la loro dedizione alla ricerca e per la compagnia durante questo percorso.