

**UNIVERSITÀ DEGLI STUDI DI ROMA
TOR VERGATA**



FACOLTÀ DI INGEGNERIA

**CORSO DI LAUREA MAGISTRALE IN INGEGNERIA
DELL'AUTOMAZIONE**

Tesi di Laurea Magistrale

**Framework di controllo open-source per robot
intelligenti**

RELATORE

Prof. Daniele Carnevale

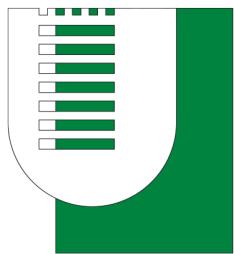
CANDIDATO

Roberto Masocco

CORRELATORE

Ing. Fabrizio Romanelli

A.A. 2020/2021



TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

*DIPARTIMENTO DI INGEGNERIA CIVILE E
INGEGNERIA INFORMATICA*

Framework di controllo open-source per robot intelligenti

Tesi di Laurea Magistrale in Ingegneria dell'Automazione

Relatore:

Prof. Daniele Carnevale

Firma:_____

Correlatore:

Ing. Fabrizio Romanelli

Firma:_____

Candidato:

Roberto Masocco

Firma:_____

Anno Accademico 2020/2021

Università degli Studi di Roma "Tor Vergata", Facoltà d'Ingegneria
Via del Politecnico 1, Roma, Italia

*Alla mia famiglia,
per aver sopportato tutte le mie stranezze
ed aver sempre creduto in me.
A tutti i compagni conosciuti in questo viaggio,
per averlo compiuto insieme.*

Ringraziamenti

Arrivato alla fine di questo percorso, un primo pensiero va naturalmente a tutti quelli con cui l'ho condiviso, in special modo a coloro che mi hanno affiancato e da cui molto ho imparato negli ultimi mesi. Senza un ordine particolare che non sia quello imposto dalla carta, vorrei ringraziare sentitamente il mio collega Alessandro Tenaglia, per aver qui implementato un formidabile algoritmo di navigazione e path planning ma più in generale per aver trascorso con me gli ultimi anni di studi, creando un rapporto di amicizia e collaborazione che ne è stata parte fondamentale. Uno speciale ringraziamento va al nostro Project Manager (*dottorando*) Simone Mattogno per l'ottimo lavoro di coordinamento logistico e le innumerevoli ore passate ad indagare il funzionamento dell'hardware e ricostruire parti, al nostro magnifico pilota e sviluppatore Lorenzo Bianchi cui va il merito di più di qualche recupero d'emergenza eseguito con pura maestria (quando anche il drone era d'accordo), e al Dott. Marco Passeri per il suo aiuto fattivo e le consulenze preziose nelle prime fasi di questo progetto. Niente di tutto questo sarebbe stato possibile senza il supporto del mio speciale correlatore, l'Ing. Fabrizio Romanelli, i cui spunti mi hanno portato ad esplorare nell'arco di pochi mesi ambiti stimolanti per me del tutto nuovi. Non basterebbero poi tutti i fogli qui raccolti per dire quanto io debba al mio relatore, il Prof. Daniele Carnevale: senza il suo supporto e incoraggiamento, nonché tutte le sfide che abbiamo dovuto affrontare, non sarei mai potuto arrivare dove sono ora. È inoltre a tutto il corpo docente di Ing. dell'Automazione, da cui ho potuto apprendere moltissimo negli ultimi anni, che rivolgo in quest'occasione un ringraziamento particolare. Infine, vorrei dedicare un caloroso saluto e ringraziamento di cuore a tutti gli amici incontrati fino al raggiungimento di questo traguardo: vi conosco solo da pochi anni, siete tantissimi e la pagina sta finendo, ma abbiamo condiviso settimane, lezioni, esami, feste e lockdown: nessuna giornata era mai uguale e nessuna meta sembrava irraggiungibile; spero di avervi dato e potervi dare tanto quanto voi avete dato a me.

Roberto Masocco, 12 ottobre 2021

Indice

Ringraziamenti	iii
Lista delle abbreviazioni	vi
1 Introduzione	1
2 Nuovi problemi e nuove soluzioni	6
Premessa	6
Fasi e criticità progettuali	7
I task	7
L'hardware	8
Il software	9
I dati e le misure	10
Debugging, testing e deployment	11
Middleware	12
Data Distribution Service	15
Discovery Module	19
Platform Specific Model	20
Caso di studio: ROS 2	21
Build system	23
Interfacce	24
Uso del DDS: context e nodi	27
Programmazione a eventi e concorrente	30
Launch System e Launch Files	31
Data recording e playback	32
Visualizzazione e simulazione	32
3 Caso di studio: drone autonomo	34

Hardware	37
Frame e motori	37
Controllore di volo	38
Fotocamere	42
Computer di bordo	45
Operating System	46
Software	49
Flight Control	51
RealSense Node	60
ORB_SLAM2	60
Aruco Scanner	70
Navigator	73
Path Finder	75
Ground Control Station	77
Precision Landing	77
FSM	80
Realizzazione	84
4 Controllori e middleware	86
Premessa	86
Problema di controllo	86
Identificazione del sistema da controllare	87
Design del controllore	88
Implementazione in ROS 2	98
5 Conclusioni	100
Sviluppi futuri	101
Elenco delle figure	103
Listings	105
Bibliografia	106

Liste delle abbreviazioni

Simbolo	Descrizione
API	Application Programming Interface
CPU	Central Processing Unit
DDS	Data Distribution Service
EDP	Endpoint Discovery Protocol
ESC	Electronic Speed Controller
GPU	Graphic Processing Unit
IDL	Interface Description Language
IMU	Inertial Measurement Unit
IoT	Internet of Things
IP	Internet Protocol
NED	North-East-Down
OMG	Object Management Group
OS	Operating System
PDP	Participant Discovery Protocol
PID	Proportional-Integral-Derivative
QoS	Quality of Service
RAM	Random Access Memory
ROS	Robot Operating System
RTPS	Real-Time Publish Subscribe
SEDP	Simple Endpoint Discovery Protocol
SoC	System on Chip
SPDP	Simple Participant Discovery Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VSLAM	Visual Simultaneous Localization And Mapping

Capitolo 1

Introduzione

Pervasività. Una parola che, come quasi il suo significato suggerisce, si sta facendo pian piano largo nelle cronache della quotidianità, per descrivere efficacemente l'impatto che le nuove tecnologie stanno avendo sulla vita di tutti i giorni. È immediato constatare quanti siano oggi i compiti e le sfide per affrontare le quali ci affidiamo, in tutto o in parte, alle macchine. Si tratta di un numero crescente e destinato a rimanere tale, e mentre aumentano le mansioni che possiamo svolgere con l'aiuto di dispositivi automatici, allo stesso modo compaiono nuovi scenari e possibilità che questi ci offrono. Ricerca, medicina, logistica, svago: sono solo alcuni degli innumerevoli ambiti in cui l'introduzione di dispositivi robotici e autonomi, via via sempre più sofisticati, ha portato negli ultimi decenni a delle innovazioni con un ritmo senza precedenti. Questa diffusione sempre più capillare porta però con sé un incremento della loro complessità, sia di progettazione che di realizzazione, cosa di cui spesso ci si fregia ma che più sommessamente si riconosce come fonte di nuove sfide e difficoltà. Ciò è, prima di tutto, per via della moltitudine dei compiti che questi apparati sono oggi chiamati a svolgere e della loro varietà.

Una seconda causa delle crescenti difficoltà con cui progettisti e sviluppatori devono misurarsi viene dai dati necessari per operare. Conseguenza dell'enorme diffusione delle nuove tecnologie è anche una capacità senza pari nella storia moderna di raccogliere informazioni sul mondo che ci circonda e sulle persone che lo abitano. Questa mole di dati, varia e diversificata, costituisce l'input degli algoritmi di Intelligenza Artificiale

CAPITOLO 1. INTRODUZIONE

che sono oggi alla base di sempre più sistemi di calcolo e automazione. La loro validità piuttosto generale e la possibilità di raccogliere virtualmente ogni genere di informazione necessaria per il loro funzionamento sono solo due delle ragioni per cui essi stanno prendendo piede in moltissimi ambiti della vita contemporanea. Ciò nonostante, tali *big data* sono complicatissimi da gestire. Progettare un sistema in grado di processare questo genere di input e operare di conseguenza porta chiaramente a confrontarsi con nuovi problemi e richiede strumenti innovativi. Acquisire tutte queste informazioni richiede inoltre un'infrastruttura di comunicazione robusta, efficiente e di grande portata. Se da un lato la rete Internet ha recentemente permesso di trasmettere dati senza fatica e a velocità elevatissime, dall'altro ogni dispositivo che la utilizza deve poterlo fare nel modo migliore per i suoi scopi. Questo implica che sia l'hardware che realizza la comunicazione, sia i protocolli che la regolano, devono essere adeguati e supportare funzionalità appropriate. Tipici esempi di situazioni in cui è necessaria una connessione tra dispositivi vanno dalla ricezione di comandi remoti all'aggiornamento del software in esecuzione sugli stessi, quest'ultimo uno scenario critico che va gestito con la massima cura per non rischiare di pregiudicare l'operatività o causare danni. È poi sempre più attraverso la Rete che si realizza l'interfacciamento tra tali apparati ed i loro operatori umani, per finalità che vanno dalla supervisione al vero e proprio controllo, senza più alcuna restrizione alla distanza tra i luoghi in cui essi sono fisicamente posti. Considerati tutti i fattori elencati fino a questo punto, ci si deve aspettare che anche le interazioni con l'uomo siano destinate ad aumentare e diventare parte sempre maggiore delle mansioni di questi sistemi. Per raccogliere queste nuove sfide e far sì che la comunità di progettisti e sviluppatori riesca a soddisfare le necessità dell'industria e del mercato, occorre riflettere su cosa si ha già a disposizione e quali novità è invece necessario integrare. In primo luogo, un sistema autonomo moderno ha bisogno di una potenza di calcolo adeguata ad acquisire misure, ricevere informazioni, processare dati ed azionare attuatori, il tutto in tempo reale. Questo requisito va naturalmente di pari passo con la complessità del sistema e dei task per cui esso è programmato, sulla base di quanto già discusso. Si può osservare negli ultimi dieci anni come l'industria si stia già muovendo in tal senso: dacché la maggior parte dei sistemi robotici e automatici erano mossi da semplici microcontrollori numerici, demandando

CAPITOLO 1. INTRODUZIONE

operazioni di gestione ad apparati di supervisione o agli operatori umani, oggi non è raro trovare pressappoco ovunque sistemi piccoli ma performanti, altamente integrati, in grado di svolgere operazioni sia di alto che di basso livello e di interfacciarsi con altri componenti più o meno complessi. È ormai comune lo scenario in cui un unico sistema è controllato da un'elettronica di bordo composta da più SoC diversi e interconnessi che si dividono i compiti a vari livelli: dal microcontrollore veloce che acquisisce le misure dai sensori e aziona gli attuatori, al computer di bordo che comunica con l'esterno e applica onerosi algoritmi per decidere di volta in volta cosa fare e come reagire agli stimoli. L'efficacia di queste soluzioni è innegabile, tanto che ha portato all'esplosione di un mercato rimasto per molto tempo di nicchia, ossia quello dei microcomputer, microcontrollori, SoC e sistemi embedded, che al giorno d'oggi sono venduti a prezzi competitivi anche al di fuori dell'ambito industriale ed utilizzati da hobbisti e amatori (e.g. i *maker*) per la realizzazione di piccoli progetti "fai-da-te". La disponibilità di una varietà così ampia di sistemi digitali ha poi reso la sperimentazione in ambito accademico molto più semplice, consentendo di dare forma rapidamente e con poco sforzo a nuove idee.

Progettando un nuovo apparato, scegliere l'hardware che lo comporrà può risultare comunque complicato. Riguardo la sola elettronica di bordo ad esempio, nonostante la gamma di componenti disponibili *off-the-shelf* sia più che mai ampia resta sempre il doveroso confronto di ogni soluzione preesistente con una sviluppata ad hoc, che ci si può aspettare risulti meglio ottimizzata ma in generale più costosa. In ogni caso, sempre per far fronte alle nuove criticità, usare l'hardware deve essere semplice e la ragione di ciò è palese: se il futuro sarà popolato da un gran numero di dispositivi tutti in grado di comunicare, interagire, cooperare tra loro e con l'uomo per risolvere problemi complicati, il software che li muove dovrà essere comparabilmente sofisticato, efficiente, sicuro e mantenibile. Gli strumenti coi quali è possibile scrivere agevolmente software con queste caratteristiche dovranno semplificare problematiche classiche e consentire di definire intuitivamente la struttura del lavoro. Ciò consentirebbe infatti di concentrarsi sui nuovi problemi, incrementando la qualità dei prodotti sviluppati ed evitando di perdere tempo a riflettere su come risolvere in modi diversi questioni già affrontate, concetto questo riassunto dal celebre adagio "non reinventare la ruota".

CAPITOLO 1. INTRODUZIONE

Considerando infine anche l’evoluzione del mondo dello sviluppo del software, sempre più comunitario e aperto al contributo del pubblico, è naturale ritenere che le nuove soluzioni debbano essere, salvo casi specifici, *libere* ed *open-source*. Ciò consentirebbe di velocizzare il progresso sotto ogni punto di vista poiché, se si hanno a disposizione pacchetti di codice pronto da riutilizzare o modificare per meglio adattarlo al proprio caso d’uso, ci si può dedicare a scrivere quello che ancora non esiste poggiando sempre su basi solide e collaudate. Inoltre condividere il software su cui si lavora ha molti altri vantaggi, tra cui un’attiva collaborazione ai fini del *testing* e del *debugging* ed una gestione collegiale delle problematiche di sicurezza, alla quale può partecipare l’intera comunità scientifica: sono infatti svariati i casi¹ di tecnologie proprietarie mantenute intenzionalmente sconosciute al pubblico contando sulla *security-by-obscurity*, rivelatesi poi affette da gravi problemi che la collettività degli addetti ai lavori ha scoperto ed evidenziato mediante *reverse engineering*. Anche questo è un punto che l’industria ha riconosciuto abbastanza recentemente come fondamentale per favorire l’innovazione: basti pensare alla fioritura e diffusione nell’ultimo decennio di servizi di hosting di repository remoti basati su sistemi di controllo versione e che integrano funzionalità di discussione, revisione e condivisione, nei quali molto del software su cui poggiano i servizi ormai d’uso comune è mantenuto e sviluppato.

Appare chiaro dunque come la progettazione e realizzazione di apparati robotici debba avvalersi di nuovi strumenti e metodologie multilivello, per realizzare un ulteriore avanzamento e rispondere alle necessità presenti e future.

L’obiettivo di questo lavoro è presentare una breve panoramica delle più recenti soluzioni ai problemi elencati, disponibili nell’ambito della robotica e che soddisfano i requisiti descritti, corredata da prove sperimentali della loro applicabilità ed efficacia. Come caso di studio si presenta un drone autonomo per volo automatico, la cui progettazione e realizzazione verranno descritte nel seguito, sviluppato nell’ambito dell’edizione 2021 del Leonardo Drone Contest e testato presso la Divisione Velivoli di Leonardo S.p.A. a Torino, Italia. Con tale prototipo, il team dell’Università di Roma "Tor Vergata" si è aggiudicato il secondo posto al termine del suddetto Contest,

¹Si veda ad esempio quanto accaduto con gli algoritmi di autenticazione COMP128 per reti GSM (2G) tra il 1997 ed il 1998.

CAPITOLO 1. INTRODUZIONE

posizionandosi tra il Politecnico di Milano e quello di Torino e sorprendendo la giuria per le performance in volo migliori tra quelle conseguite dai vari team in gara.

Il resto di questo documento è organizzato come segue: nel Capitolo 2 verranno discussi i più recenti strumenti software rivolti alla realizzazione di sistemi autonomi complessi e distribuiti; nel Capitolo 3 verrà introdotto il caso di studio del drone, discutendone le componenti hardware, il software ed i risultati ottenuti; nel Capitolo 4 verrà presentata l'implementazione di un controllore veloce su tale architettura; infine nel Capitolo 5 verranno riassunte le novità presentate e proposti alcuni sviluppi successivi.

Capitolo 2

Nuovi problemi e nuove soluzioni

Premessa

Lo sviluppo di un sistema robotico autonomo è un processo che attraversa varie fasi. Ciascuna di esse è solitamente caratterizzata dalla definizione e soluzione di problematiche inerenti diversi aspetti dell'apparato, che possono andare dalle operazioni che questo sarà chiamato a svolgere, alla sua costruzione e programmazione, fino al modo in cui i dati che esso raccoglie e che condizionano il suo funzionamento debbano essere presentati agli eventuali operatori. Volendo generalizzare dallo specifico caso applicativo, la maggior parte delle problematiche si possono ripartire tra poche tipologie di alto livello. Nonostante possa risultare controiduitivo dalla suddivisione che ci si appresta a delineare, queste non sempre si rivelano affrontabili in sequenza, né tanto meno completamente indipendenti. È anche per risolvere situazioni simili che negli ultimi tempi sono state ideate svariate tecniche di project management, volte anche a minimizzare il rischio che un errore commesso in una fase precedente possa pregiudicare quelle future, compromettendo il lavoro svolto ed allungando i tempi di sviluppo. In virtù di quanto osservato nel capitolo precedente, è chiaro che con l'aumentare della complessità dei sistemi automatici odierni e dei compiti che essi devono svolgere, queste tematiche siano quanto mai attuali. In particolare, nuovi strumenti messi a disposizione di sviluppatori e progettisti dovranno riflettere queste specificità,

semplificare la gestione di situazioni tipiche dove possibile, ed evidenziare eventuali criticità per aiutare nella soluzione dei problemi. Nonostante la generalizzazione che si sta facendo possa sembrare eccessiva, nella realtà pratica si riscontra facilmente che a dispetto della varietà di casi di specie e applicazioni, le complicazioni da affrontare sono quasi sempre se non le stesse, molto simili, e lo stesso si può rilevare circa le strategie da adottare.

L'obiettivo di questo capitolo è presentare brevemente le problematiche che solitamente si devono affrontare quando si progetta un sistema robotico, evidenziandone le peculiarità seppur senza concentrarsi su un particolare ambito applicativo, e passare poi ad esaminare alcune recenti soluzioni hardware e software che consentono di ottimizzare il lavoro di progettazione, sviluppo e testing in loro funzione. La discussione parte ora da un livello abbastanza alto di astrazione ma diventerà via via sempre più specifica, costituendo di fatto la linea guida seguita durante il progetto che costituisce il caso di studio di questo lavoro e dunque un filo che lega tutti i capitoli successivi.

Fasi e criticità progettuali

I task

Le prime decisioni da prendere all'inizio del progetto di un sistema automatico o robotico riguardano naturalmente le specifiche delle operazioni che questo dovrà svolgere. Esse vanno completamente delineate fin da subito nei minimi particolari, in quanto è da loro che dipenderanno tutte le scelte successive, da quelle inerenti la costruzione e la scelta dell'hardware fino alla sua programmazione. Durante questa decomposizione dei requisiti operativi in *task*, intesi come singole unità di lavoro da eseguire, compariranno diversi tipi di questi ultimi. Senza scendere, per ora, nei particolari di uno specifico caso, tipologie comuni sono:

- task **continuativi**, ossia iniziati all'accensione del sistema e mai interrotti fino al suo spegnimento o malfunzionamento;
- task **periodici**, da eseguire a intervalli regolari;

- task **asincroni**, svolti solo quando un particolare evento si verifica.

È facile rendersi conto di come gran parte delle funzionalità presenti nei sistemi robotici odierni possa essere decomposta in task appartenenti a ciascuna di queste categorie. Ne fanno evidentemente parte operazioni che vanno dal controllo degli attuatori, alla raccolta di dati e misure necessari agli algoritmi di controllo, fino alla comunicazione con operatori ed utenti. Le specificità di questi task avranno risvolti sia sull'hardware che dovrà realizzarli che sul software che li codificherà, i quali dovranno essere organizzati opportunamente in loro funzione, anche in modo da facilitarne lo sviluppo e la manutenzione.

L'hardware

Le decisioni da prendere successivamente riguardano l'hardware da utilizzare per realizzare i primi prototipi dell'apparato. Probabilmente non si tratterà di quello che comporrà il prodotto finito, per via di eventuali errori di valutazione, imprevisti o altre problematiche sorte successivamente. Ciò nonostante esso deve essere selezionato con cura in base ai requisiti operativi, strutturali e costruttivi definiti fin qui. Le decisioni prese durante il passo precedente rivestono in questa fase un'importanza cruciale: stabilendo infatti quale sia la complessità computazionale delle operazioni che il sistema dovrà svolgere, sarà richiesto un hardware di alto livello più o meno potente, equipaggiato eventualmente con coprocessori discreti¹ necessari per trattare particolari tipi di dati, o suddiviso in più sottosistemi interconnessi. Al giorno d'oggi, ricordando la discussione fatta nel capitolo precedente circa la disponibilità di un'ampia gamma di soluzioni off-the-shelf, è d'obbligo un'indagine di ciò che offre il mercato per capire se sia possibile sfruttare componenti o SoC forniti già testati e pronti per essere utilizzati, piuttosto che ricorrere ad una soluzione proprietaria che porta generalmente con sé considerevoli costi di sviluppo, testing e validazione. Per i motivi sopracitati è molto difficile, se non impossibile, delineare fin da questo punto una struttura complessiva e definitiva dell'hardware: ciò che importa è capire di cosa c'è bisogno e se esistono

¹Con il termine *discreto* si intende hardware aggiuntivo montato su di un calcolatore, non in grado di operare da solo ma fatto per essere pilotato da quest'ultimo, offrendogli supporto specifico per particolari operazioni; classici esempi sono i coprocessori matematici, le GPU e le schede audio.

sul mercato offerte compatibili con i propri requisiti, cercando di prevedere anche successive richieste aggiuntive. Seguendo questa linea, eventuali variazioni potranno essere gestite similmente con poco sforzo.

Il resto delle scelte riguardano l'hardware di più basso livello, formato in generale da:

- microcontrollori veloci;
- attuatori;
- sensori;
- interfacce e dispositivi di comunicazione;
- hardware per il controllo e la supervisione;
- sistemi di immagazzinamento di dati.

Niente di più si può dire su di esse prescindendo dal caso di specie, tuttavia nel seguito di questo lavoro saranno descritte situazioni simili relative all'esempio che si discuterà. Per ora ci si può limitare a sottolineare che le osservazioni fatte precedentemente valgono anche per questi componenti.

Il software

Eccezion fatta per quei componenti che svolgono funzioni cruciali e molto specifiche, come circuiti elettrici/elettronici, la quasi totalità delle operazioni svolte da un sistema robotico è oggi codificata in un calcolatore ad esso interno, che traduce tali istruzioni in comandi per gli altri componenti, sottosistemi, sensori o attuatori. Non è questa la sede per la discussione di procedure standard di progettazione e realizzazione del software, ma basta soffermarsi a riflettere su un punto: anche il software che governa un robot, data la complessità di quest'ultimo, avrà un'organizzazione stratificata e fortemente gerarchica. In essa, i livelli inferiori saranno occupati da quei moduli che dovranno interagire con, se non muovere, i sottosistemi più piccoli e veloci (e.g. il *firmware*), e che andranno sviluppati ponendo particolare attenzione alle caratteristiche di essi ed alle loro finalità. Man mano che si sale nella scala gerarchica ci si muove verso moduli a più

ampio spettro, che codificano operazioni di supervisione, pianificazione e comunicazione con gli utenti, e per questo più vicini alla nozione di software applicativo o di sistema, con cui magari si è più abituati ad avere a che fare essendo quello "maggiormente visibile" ad un utente finale. Supponendo di dover operare in questo contesto, si deve poter contare su degli strumenti che semplifichino il più possibile il design e lo sviluppo a tutti i livelli senza distinzione, aiutando nella gestione di problematiche inerenti l'organizzazione dei moduli, la comunicazione, il processamento e la memorizzazione di diversi tipi di dati.

Rientra in questo contesto anche una decisione circa i linguaggi di programmazione da adottare. Com'è noto, ciascuno possiede le sue caratteristiche peculiari, che ne evidenziano rispetto alle alternative punti di forza ma anche debolezze. Pochi sono i linguaggi di programmazione adatti per molteplici scopi, intendendo in questo senso il grado di facilità con cui consentono di codificare diversi tipi di task, e nessuno è da considerarsi universale. Nel seguito di questo lavoro verranno presentate, a titolo di esempio, due diverse alternative, mettendone in risalto le specificità e motivando un uso preponderante di una delle due con dei requisiti progettuali del caso di studio.

I dati e le misure

Un robot, per portare a termine i propri task con successo, necessita più o meno costantemente di informazioni circa quello che gli sta accadendo intorno. Esse possono essere ricevute entro pacchetti dati, gestiti da opportune interfacce di comunicazione, oppure acquisite direttamente mediante degli organi di misurazione, comunemente detti *sensori*. Per entrambe queste soluzioni, problematiche classiche che si incontrano da subito sono l'integrazione dei relativi dispositivi nel resto dell'apparato e il processamento dei dati che queste restituiscono. Esse sono peraltro cruciali, in quanto il funzionamento dell'intero apparato è quasi sempre condizionato alla possibilità di disporre di informazioni circa l'ambiente circostante e lo stato del robot stesso². E' dunque lecito aspettarsi che un robot sia dotato di una gran quantità di interfacce di comunicazione e sensori, questi ultimi composti da hardware specifico atto a misurare

²Si pensi agli algoritmi di controllo in *feedback*, basati sulla possibilità di acquisire misure delle azioni di controllo impartite ad un sistema e di grandezze che ne esplicitano gli effetti.

particolari grandezze fisiche d'interesse. Le due problematiche introdotte poco fa riguardano pertanto l'integrazione di tale hardware con il resto del sistema e la scrittura di moduli software capaci di gestirlo, e ricevere e processare i dati che produce. Sia le piattaforme hardware che gli strumenti software dovranno essere sviluppati o scelti, per quanto possibile, tenendo conto di queste eventualità in modo da minimizzarne l'impatto e facilitare il lavoro.

Debugging, testing e deployment

Non appena i primi prototipi vengono finalizzati occorre iniziare a testarli per verificarne il funzionamento e confermare la validità delle decisioni progettuali prese fino a quel punto. Tenendo presente che non si può definire una procedura di testing generica e universale per qualunque tipo di sistema robotico, ci si può aspettare che non appena si è constatato il corretto funzionamento dell'hardware, si debba passare alla validazione del software. Il debugging del software in esecuzione su un robot è in generale più difficile rispetto a quello di un comune software applicativo, e ciò a causa di diverse specificità:

- è più difficile tracciare l'esecuzione di programmi su un dispositivo autonomo, potenzialmente in movimento: non è detto che si abbia accesso diretto al sistema e tipicamente gli strumenti a disposizione sono limitati o non molto sofisticati;
- in caso di problemi, oltre al software va ricontrollato anche l'hardware in uso, in quanto il problema potrebbe anche essere causato da quest'ultimo o da un suo uso non corretto;
- eseguire test non è semplice quanto lanciare un programma e controllarne gli esiti: va assicurata anche la sicurezza dell'apparato e dei suoi operatori durante l'intera procedura, motivo per cui già la sola organizzazione di un test può risultare impegnativa.

Date la difficoltà e l'importanza di questa fase, è ragionevole voler impiegare strumenti e metodologie che possibilmente facilitino le cose anche a questo livello, che magari siano già state adeguatamente validate e certificate.

Infine si può passare al *deployment*, ossia la costruzione di un apparato che si possa considerare un prodotto finito da mettere in uso. È questo il momento in cui si manifestano le conseguenze di alcune scelte progettuali che in questa sede si è scelto di non trattare, relative ad esempio alla produzione e realizzazione delle varie parti. Ciò non toglie come una ragionevole oculatezza nelle fasi precedenti possa risultare conveniente anche qui.

Alla luce della discussione appena conclusa, si rileva come le problematiche da affrontare nella realizzazione di un sistema robotico siano molteplici e variegate, come pure le scelte che si è chiamati a fare in diversi punti dello sviluppo. Le ripercussioni che queste possono avere sono pure diversificate, e rischiano di manifestarsi anche in stadi più avanzati. In tempi recenti le difficoltà delineate si sono poi acute, proprio a causa della maggiore complessità ed autonomia demandate a tali apparati. Ciò nonostante, lo sviluppo tecnologico è proseguito, rendendo anche piccoli SoC dotati di CPU multicore performanti, RAM capiente e GPU, e capaci di integrarsi facilmente con microcontrollori numerici più veloci e sottosistemi real-time più piccoli, eseguendo task abbastanza veloci oppure più lenti, ad esempio di supervisione, coordinamento o comunicazione. Quello che resta sono le difficoltà relative alla programmazione e lo sviluppo del software dedicato.

Il resto di questo capitolo è dedicato alla discussione di nuove soluzioni, recentemente divenute interessanti in ambiti quali IoT, mobile devices e telecomunicazioni, ma che stanno trovando anche nella robotica enormi risvolti applicativi. È anche grazie ad esse che il drone autonomo descritto successivamente, esempio pratico di questo studio, è stato realizzato con relativa semplicità.

Middleware

Nel 1968 a Garmisch, in Germania, si tenne una conferenza sponsorizzata dal Comitato Scientifico della NATO sul tema dello sviluppo del software. Nonostante non se ne parli molto, si trattò di un evento storico molto importante in questo ambito: venne fatto il punto sulle pratiche e metodologie di design seguite dalle maggiori industrie

attive nel campo dell'informatica, allora in forte crescita e competizione, nel tentativo di individuare strategie standard per affrontare problemi comuni.

Si ricorda infatti che all'epoca, e per altri quindici anni almeno, non sarebbe esistito quasi nessuno standard da seguire nella produzione e programmazione dei calcolatori: ogni azienda proponeva il suo prodotto sviluppato da zero, nell'hardware e nel software. Solo quando le informazioni hanno iniziato ad essere scambiate tra diversi calcolatori, prima su supporti di memorizzazione e poi attraverso Internet contemporaneamente ad una domanda sempre crescente di nuovi tipi di dispositivi, è iniziata un'opera di standardizzazione e collaborazione tra i diversi produttori.

Si può trovare in [1] una trascrizione abbastanza completa della conferenza. In essa si parla per la prima volta ufficialmente di alcuni concetti oggi ben noti, come *component design* o *unit test*, ma ciò che interessa ai fini di questo lavoro è la rappresentazione della struttura gerarchica del software presentata da uno dei relatori, valida ancora oggi a meno di dovute estensioni, con lo schema a piramide invertita riportato in Figura 2.1. L'idea è che un sistema complesso, così come una piramide invertita, non può restare operativo senza un adeguato supporto offerto da strumenti secondari, magari totalmente invisibili all'utente finale, ma che sono d'importanza vitale per il corretto funzionamento del tutto: *compilatori* e *assemblatori*, ossia programmi che traducono codice scritto in un qualche linguaggio prima nell'assembly dell'architettura target e poi in linguaggio macchina binario, tipicamente generando alla fine della pipeline di processamento un file eseguibile. Questi devono essere affidabili ed efficienti, poiché da essi dipende l'implementazione di tutto il resto.

Successivamente troviamo gli strati dei *Control programs* e *Service Routines*: si tratta di quelli che al giorno d'oggi siamo abituati a chiamare *Sistema Operativo* e *drivers*: questi vengono sviluppati solitamente a diretto contatto con la macchina, tenendone presenti le specificità, e dipendono fortemente dagli strumenti di compilazione impiegati. Il loro ruolo è quello di fornire un'astrazione dell'hardware che ne renda semplice l'utilizzo sia da parte dell'utente finale, che usa le applicazioni, sia da parte degli stessi programmatori di applicazioni. Ed il software applicativo è proprio quello che ritroviamo nell'ultimo livello: programmi che usano l'hardware più o meno direttamente per offrire agli utenti svariati servizi.

C'è però uno strato, intermedio, che solitamente non viene considerato poiché in passato non sempre presente o difficile da caratterizzare: il *middleware*. Posizionato a metà tra il software di sistema e quello applicativo, il middleware trova la sua prima definizione e caratterizzazione proprio in occasione della suddetta conferenza. Inizialmente pensato solo come uno strato connettivo tra software corrente e *legacy*, è un software che fornisce servizi e funzionalità comuni alle applicazioni al di fuori di quanto offerto dal sistema operativo. Dal punto di vista di quest'ultimo, si presenta come un normale software applicativo, composto ad esempio da librerie condivise o dinamiche. La gestione dei dati, i servizi applicativi, la messaggistica, l'autenticazione e la gestione delle API³ sono tutti compiti comunemente assolti dal middleware, il quale dunque aiuta gli sviluppatori a creare applicazioni in modo più efficiente, agendo come tessuto connettivo tra queste ultime, i dati e gli utenti.

In ambienti distribuiti o in presenza di container⁴, il middleware può rendere facile e conveniente sviluppare ed eseguire applicazioni su larga scala. Un servizio generalmente offerto da un middleware è un modo per scambiare dati tra applicazioni, anche tramite dispositivi diversi, secondo un formato comune denominato *interfaccia* definito al bisogno.

In generale, i servizi offerti sono specificati solo nei termini della loro dinamica ma possono essere portati su diverse architetture e dispositivi, funzionandovi allo stesso modo pur mantenendo la loro implementazione nascosta all'utente finale: ad esempio, quando un middleware si occupa di realizzare la comunicazione tra applicazioni, in uno stack di rete si porrebbe tra il livello di trasporto e quello di applicazione.

Oggi, il middleware si può dividere in due tipologie: quello che opera in *tempo umano*, e dunque è pensato per essere usato direttamente da utenti umani (e.g. servizi web), e quello che opera in *tempo macchina*, ed è dunque composto principalmente da API che i programmatori di applicazioni possono usare per svolgere i loro compiti con più

³Con *Application Programming Interface* si intende un insieme di funzioni e procedure proprie di un sistema, un software o un linguaggio di programmazione, atte all'espletamento di un determinato compito. Sono generalmente implementate come librerie rese disponibili al programmatore.

⁴Un *container* è un pacchetto di software autosufficiente, comprendente sia il proprio codice sorgente che tutte le dipendenze, dunque pronto per essere direttamente eseguito su qualunque sistema o macchina. Sono spesso usati in ambienti virtualizzati, assegnandogli in parallelo porzioni delle risorse della macchina host e isolandoli tra loro, garantendo prestazioni uniformi e predibili e il rispetto di standard di sicurezza.

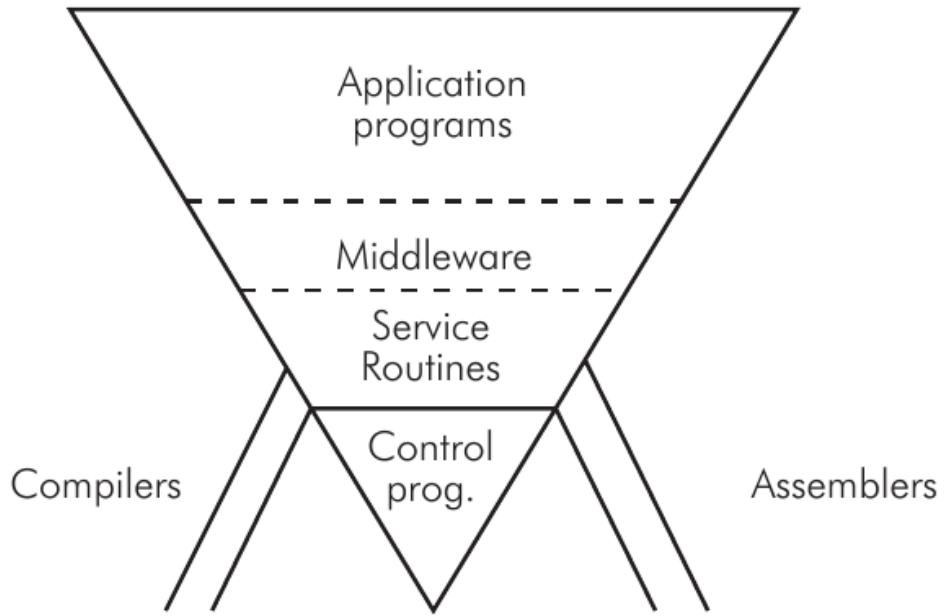


Figura 2.1: Organizzazione del software in esecuzione su un calcolatore generico.

facilità e beneficiando di una maggiore portabilità verso un'ampia varietà di dispositivi. Alla luce di tutto ciò, ricordando le difficoltà che comunemente si incontrano durante il design del software di un sistema robotico, ci si può rendere conto di come l'adozione di un middleware nei sottosistemi di più alto livello possa rendere le cose più semplici: molti dei moduli che andrebbero realizzati ex novo sarebbero invece implementati dal middleware stesso e offerti sotto forma di servizi da includere nei programmi da sviluppare.

Data Distribution Service

Nel 1989, undici compagnie tra cui Hewlett-Packard, IBM, Sun Microsystems, Apple Computer e American Airlines fondarono l'*Object Management Group*, un consorzio tuttora attivo nel campo dell'informatica con il fine di definire standard per il software a scopo industriale, favorendo l'integrazione di soluzioni di vendors diversi in unici ecosistemi, relativi a differenti tipi di applicazioni e tecnologie. L'obiettivo dell'OMG è stabilire, per ogni nuova tipologia di software in esame, un modello comune da seguire

nella sua implementazione, che renda possibile e favorisca il suo sviluppo ed utilizzo su potenzialmente qualunque genere di piattaforma che sia per essa rilevante.

Il Gruppo fornisce all'industria solo specifiche e non implementazioni, anche se un team che ne sottomette una nuova, prima che questa sia accettata, deve fornirne una possibile implementazione: ciò è, intuitivamente, per evitare di definire standard inutili o irrealizzabili. Tra le compagnie membri dell'OMG è favorito lo sviluppo di soluzioni che possano interoperate da subito e senza alcuna limitazione.

Uno standard definito dall'OMG è quello denominato *DDS - Data Distribution Service* [2]. Esso definisce un particolare tipo di middleware, incaricato di gestire comunicazioni dirette tra sistemi real-time specificando come i dati debbano essere serializzati, deserializzati, trasmessi e instradati, e come debbano essere fatte le API che consentono ai programmati di invocare queste operazioni. Ma c'è molto di più: l'obiettivo di un DDS è consentire a tecnologie, piattaforme e dispositivi di differenti vendors, operanti in ambiti e luoghi diversi, di comunicare da subito scambiandosi dati di cui sia noto a priori solo il formato e poco altro, consentendo di gestire criticità, prestazioni e scalabilità dei vari apparati e della rete che li collega. Di fatto, un DDS semplifica notevolmente ad un programmatore quello che sarebbe un complesso lavoro di configurazione di rete ed organizzazione delle trasmissioni, che come si è visto è una delle problematiche classiche incontrate durante la realizzazione di un sistema robotico.

Il modello del middleware è di tipo *publish-subscribe*: le entità coinvolte nella comunicazione, denominate *nodi*, dichiarano alle altre la propria volontà di trasmettere o ricevere pacchetti di informazioni, codificati secondo opportune interfacce. La ragione della denominazione sta nell'organizzazione delle diverse trasmissioni, dal punto di vista del programmatore, in *topic*, ossia astrazioni che rappresentano canali di comunicazione definiti da:

- un **nome** che identifichi il topic, non necessariamente in modo univoco, generalmente codificato come una stringa di testo in modo che per i programmati sia facile ricordarlo e utilizzarlo;
- un'**interfaccia**, descritta secondo un qualche linguaggio (e.g. IDL⁵), che specifici quale sia il contenuto di un singolo pacchetto di dati e come sia codificato;

⁵Con *Interface Description Language* si intende un linguaggio con cui si possano descrivere elementi

- una policy di **Quality of Service**, specificata dal programmatore per impartire al middleware precise indicazioni su come debbano essere gestite le comunicazioni.

Su uno stesso topic, un nodo può essere sia publisher che subscriber, e non sono vietate a priori istanze multiple di stessi topic. Riguardo il QoS, dal punto di vista dell'organizzazione delle trasmissioni sulla Rete questo riveste un ruolo importante, essendo uno strumento a disposizione del programmatore per invocare uno specifico comportamento del DDS.

Una policy QoS di un DDS è generalmente composta da:

- **history**, impostazione che indica se vadano bufferizzati tutti i pacchetti trasmessi su un determinato topic oppure un loro numero massimo prefissato;
- **depth**, profondità della coda finita eventualmente impostata al punto precedente;
- **reliability**, che sta ad indicare se le comunicazioni vadano gestite dal middleware in modo da assicurare o meno la ricezione da parte di tutti i subscribers di un topic dei pacchetti trasmessi dai publishers;
- **durability**, che consente di conservare i pacchetti trasmessi da un publisher in modo da recapitarli anche ai subscribers che dovessero connettersi successivamente;
- **deadline**, **lifespan**, **liveliness** e **lease duration**, tempi massimi caratteristici che consentono di capire quando un pacchetto è diventato vecchio o un nodo può considerarsi non più attivo.

Chiaramente esistono delle impostazioni di default, che si cerca di mantenere il più possibile uguali tra diverse implementazioni in modo da garantire la massima compatibilità.

È evidente già a questo livello la notevole semplificazione introdotta da un middleware DDS: con una configurazione relativamente semplice e rapida si possono risolvere immediatamente problemi quali la scelta di un protocollo di trasporto (e.g. TCP o

operativi che siano intelligibili da linguaggi diversi. Ne esistono vari, e in questo contesto un DDS ne fa un uso volto a rendere semplice al programmatore la specifica del formato di un pacchetto dati, nonostante nella specifica dell'OMG non sia indicato un IDL in particolare.

UDP) e la costruzione di moduli software che lo usino per trasmettere informazioni codificate secondo specifiche regole.

Il DDS si occupa di tutti i compiti necessari ad organizzare le comunicazioni: dai più basilari quali la trasmissione dei messaggi a quelli organizzativi come l'advertising dei publisher e la discovery dei nodi nella rete, al controllo di flusso e delle ritrasmissioni, gestisce la ridondanza dei publisher e altre situazioni patologiche, il tutto in modo trasparente al programmatore. Un DDS di fatto rende molto semplice costruire applicazioni distribuite, o reti di esse, in quanto nasconde e semplifica allo sviluppatore moltissimi compiti di gestione, consentendogli di prescindere totalmente da fattori che invece potrebbero risultare complessi da gestire (si pensi ai problemi che può creare la sola dislocazione geografica dei nodi).

I DDS oggi commercialmente disponibili offrono librerie di API in diversi linguaggi quali Ada, C, C++, C#, Java, Python, Scala, Lua, Pharo e Ruby, ed essendo costruiti tutti sullo stesso standard sono completamente compatibili ed interoperabili. Talvolta vengono organizzate dimostrazioni che coinvolgono diversi vendors volte a presentare e dimostrare:

- funzioni di connettività di base su strato di rete con protocollo IP;
- discovery di publishers e subscribers;
- compatibilità delle comunicazioni soggette a requisiti di QoS uguali o simili;
- gestione dei ritardi di trasmissione;
- gestione di situazioni patologiche quali multiple istanze di stessi topic;

Esistono poi specifiche riguardanti tipologie di DDS di più basso livello, pensati per essere inclusi in sistemi a risorse limitate, e.g. *embedded*, restando comunque compatibili con le più complete implementazioni di alto livello.

Le modalità di comunicazione, serializzazione e deserializzazione dei dati, sono definite nello standard OMG denominato *Wire Protocol*, di cui una recentissima versione è disponibile in [3]. Nonostante l'ampiezza del contenuto, ai fini di questa trattazione è necessario evidenziarne due parti salienti.

Discovery Module

Questo modulo del DDS comprende dei protocolli che consentono a diverse istanze di DDS in esecuzione su sistemi collegati in Rete di trovarsi ed organizzare le eventuali comunicazioni da quel momento in poi. Essi sono generalmente due:

- **Participant Discovery Protocol (PDP)**, che consente alle istanze dei DDS, ciascuna relativa ad una qualche applicazione in esecuzione su una determinata macchina e per questo denominate *participants*, di annunciare alle altre la propria esistenza ed allo stesso tempo ottenere informazioni circa eventuali altre raggiungibili;
- **Endpoint Discovery Protocol (EDP)**, chiamato in causa subito dopo il precedente e relativo alla trasmissione di informazioni che riguardano gli *endpoints* che i vari participants possiedono, intesi come publishers o subscribers che risultano compatibili in quanto relativi ad uno stesso topic, una stessa interfaccia e governati da policy QoS compatibili.

Le implementazioni possono scegliere di includere, oltre ai protocolli PDP ed EDP definiti dallo standard⁶, delle versioni proprietarie. Se due istanze che iniziano una comunicazione hanno in comune almeno un PDP e un EDP, possono scambiarsi informazioni e completare la discovery.

Si tratta di protocolli che in uno stack di rete si troverebbero al livello di IP, di tipo *unicast* oppure *multicast* a seconda che i participants coinvolti siano tutti noti a priori o meno: nel primo caso è necessaria una configurazione di rete preliminare del DDS in modo che gli indirizzi ed i range di porte di tutte le macchine coinvolte siano preimpostati, mentre nel secondo i protocolli faranno tutto automaticamente non appena le istanze dei DDS vengono avviate. La potenzialmente vastissima scala operativa di queste soluzioni ha reso i DDS appetibili per ambiti mission-critical quali quello militare e logistico, nonostante non sia per ora consigliabile appoggiarsi interamente alla rete Internet per questo tipo di comunicazioni in quanto, a meno di configurazioni predefinite, è necessario che ogni apparato di Rete presente nel percorso tra due participants supporti il multicast e non ostacoli le trasmissioni.

⁶Essi sono noti come **SPDP** ed **SEDP** rispettivamente, dove la "S" sta per "Simple".

Platform Specific Model

Questo modello stabilisce come dati di formati e codifiche diversi debbano essere serializzati, trasmessi e deserializzati, in modo univoco per tutte le possibili implementazioni e piattaforme, tra i diversi topic.

Tutti i pacchetti dati sono trasportati su UDP. Indipendentemente dal fatto che l'associazione degli endpoints sia stata eseguita in multicast o unicast, il protocollo del Discovery Module avrà aperto per essi *socket* UDP con numeri di porta ben precisi, dipendenti da:

- **Domain ID**, costante numerica che seleziona nell'implementazione un range di porte UDP e dunque consente alle applicazioni che usano il DDS di comunicare solo specificando stessi domini; più domini possono coesistere anche entro una stessa macchina;
- base del range di porte UDP utilizzabili nel sistema;
- **Participant ID**, identificativo univoco di un'istanza entro una stessa macchina;
- eventuali configurazioni manuali.

Ci si potrebbe a questo punto chiedere perché usare UDP per il trasporto dei dati. Le ragioni di questa scelta sono le stesse di altri contesti simili: UDP è un protocollo basico, best-effort e privo di qualunque funzionalità di gestione della comunicazione che non sia il semplice trasporto tra endpoints, per questo leggero e dunque supportato da un'amplissima varietà di dispositivi e piattaforme; tutto ciò conferisce alle sue implementazioni una predicitività totale, fatta ovviamente eccezione per quello che accade nella Rete, e una naturale scalabilità. Per questo è la scelta migliore sia per i sistemi real-time, sia per lo sviluppo di questo tipo di software.

Caso di studio: ROS 2

Si è discussa finora una classe di moduli software i cui obiettivi coincidono con quelli che emergono durante la progettazione di un sistema robotico, e si è passati a descriverne una tipologia specifica atta ad offrire particolari servizi di comunicazione real-time. Con tutti questi elementi a disposizione è ora possibile introdurre e discutere il middleware che costituisce il caso di studio di questo lavoro, essendo stato pensato per applicazioni alla robotica di qualunque tipo ed utilizzato in questa sede per realizzare e programmare il drone autonomo descritto nei prossimi capitoli.

ROS 2 è la seconda versione del middleware *Robot Operating System* sviluppato dal 2007 da Willow Garage, uno spin-off dello Stanford Artificial Intelligence Laboratory. Si tratta, fin dalle prime versioni, di un middleware volto ad offrire a chi programma sistemi robotici e automatici API e servizi quali astrazione dell'hardware, controllo di dispositivi a basso livello, scambio di dati e comunicazione interprocesso, supporto alla realizzazione di moduli di tipo client-server e gestione dei pacchetti software. L'intero progetto è sempre stato open-source, con un modello comunitario di sviluppo e attenta revisione: il codice sorgente di ogni versione, pronto per la compilazione e l'installazione, è raggruppato in diversi repository su GitHub, e la documentazione è pubblicata online assieme a svariati *design documents* che spiegano e commentano le scelte progettuali fatte e le nuove funzionalità introdotte. Fix e novità vengono testati in una *Rolling Release* e successivamente introdotti nelle release stabili, le quali seguono un naming scheme alfabetico simile a quello di Canonical per la distribuzione di Linux *Ubuntu* e per questo sono spesso chiamate "distribuzioni".

I sistemi operativi supportati ufficialmente sono:

- macOS;
- Windows;
- Ubuntu Linux, con possibilità di installare i pacchetti binari o compilare tutto dal codice sorgente.

A partire dal codice sorgente è comunque possibile sulla carta installare tutti i pacchetti di una distribuzione di ROS su un qualunque sistema operativo, e infatti la comunità

mantiene istruzioni di installazione aggiornate per diverse altre distribuzioni di Linux non ufficialmente supportate.

Si può consultare [4] per un riassunto della storia di ROS e una sintesi dei suoi obiettivi progettuali. Da essa si evince come il progetto si sia ben presto dimostrato potenzialmente adatto a casi d'uso complessi, quali ad esempio la gestione distribuita e real-time di gruppi di robot collaborativi o l'integrazione omogenea di diversi tipi di hardware ed interfacce di comunicazione. Si può poi trovare in [5] una descrizione più tecnica e dettagliata delle novità della versione 2, che ne introduce molte delle features salienti.

La differenza sicuramente più importante di ROS 2 rispetto alla prima versione, e che rappresenta attualmente il suo maggior punto di forza, è il supporto ai DDS [6]: l'intero middleware è infatti costruito su un DDS, la cui implementazione specifica è lasciata decidere all'utente ed è totalmente intercambiabile, per realizzare i servizi di base di comunicazione interprocesso e scambio di dati. Ciò significa che tutti i benefici dei DDS elencati precedentemente, quali facilità d'uso e definizione delle interfacce, Quality of Service, configurazione delle trasmissioni tramite discovery e serializzazione sono resi immediatamente disponibili agli sviluppatori di sistemi robotici risolvendo non pochi problemi. Va detto però che questa ulteriore astrazione rende un po' più difficile l'interazione col DDS sottostante, creando qualche problema in contesti ad oggi problematici come la trasmissione di messaggi di grandi dimensioni su reti soggette a perdita; si trovano comunque online testimonianze dell'attività della comunità di sviluppo di ROS e dei DDS per migliorarli anche in tal senso.

Le librerie e i pacchetti software di ROS sono disponibili in diversi linguaggi ma i due ufficialmente supportati sono C++ e Python. Ad esempio, nella versione 2, l'intera architettura poggia su una libreria base scritta in C denominata *rcl - ROS Client Library*⁷, che implementa le funzionalità di più basso livello e da cui derivano quella in C++ e quella in Python, denominate *rclc++*⁸ ed *rclpy*⁹ rispettivamente. Risulta dunque molto semplice, in questi termini, sviluppare software per costruire e programmare apparati, usare nuovo hardware o implementare nuove funzionalità: si può beneficiare

⁷<https://github.com/ros2/rcl>

⁸<https://github.com/ros2/rclc++>

⁹<https://github.com/ros2/rclpy>

delle librerie standard di ROS e del vastissimo ecosistema di pacchetti open-source basati su quest'ultimo che sono disponibili pubblicamente, e contribuire col proprio lavoro al progresso della comunità di ingegneri e sviluppatori che ne fanno uso.

Per dare un'idea dei principali servizi offerti da tale middleware ai fini di questa trattazione, verranno ora descritte le caratteristiche salienti di ROS 2 secondo un approccio top-down, passando infine in rassegna alcuni strumenti ausiliari ma molto utili; si escludono così molti comandi e utilità minori relative alla creazione e gestione dei pacchetti, per i quali però si rimanda alla documentazione tecnica ed ai tutorial di base.

Build system

Nell'ecosistema di ROS, il software è diviso naturalmente in unità denominate *pacchetti*. L'idea è che ogni pacchetto costituisca un modulo software a sé stante, atto a realizzare una specifica funzione nell'economia del sistema, eventualmente dialogando a vari livelli con altri e facendo uso di determinati servizi. Ciò nonostante, come si è più volte evidenziato finora, un robot è uno scenario diverso da quelli tipici dello sviluppo del software, dunque non è raro che uno sviluppatore possa trovarsi a lavorare su più pacchetti contemporaneamente. A tale complessità si aggiunge anche il fatto che se diversi moduli devono dialogare, i corrispondenti pacchetti risulteranno interdipendenti. Occorre dunque un sistema che renda semplice la costruzione di uno *workspace* in cui sviluppare ed eseguire il software e la compilazione dei pacchetti al suo interno. La soluzione offerta da ROS 2 è costituita da un build system universale per ogni contesto in cui questo middleware viene impiegato, diviso in due livelli e le cui ragioni sono descritte in [7].

In principio vi è *colcon*¹⁰. Doveva trattarsi di un software che automatizzasse e semplificasse la compilazione e l'installazione di pacchetti di codice Python o C++ tramite CMake¹¹ ad uso di ROS 2, ma ben presto divenne un progetto personale

¹⁰<https://colcon.readthedocs.io/en/released/>

¹¹CMake (cmake.org) è un software libero multipiattaforma per l'automazione dello sviluppo, il cui nome è un'abbreviazione di "cross platform make". Nasce per semplificare il procedimento di build fornendo un linguaggio di scripting con cui automatizzarlo semplicemente anche per progetti complessi.

del suo autore di valenza più generale: di fatto oggi ROS 2 si appoggia a colcon che però può essere utilizzato del tutto indipendentemente per altri scopi. In essenza, colcon crea un workspace definendo alcune directories di base nel file system in cui andrà a costruire le applicazioni, e procede poi alla loro compilazione e installazione secondo regole automatiche o specificate, risolvendo automaticamente le dipendenze. Ogni pacchetto, per risultare intelligibile da colcon, deve essere munito di un *package manifest* nella forma di un file XML¹² in cui tutte le caratteristiche del modulo, come ad esempio il nome o le dipendenze, siano riportate. L'installazione può risolversi nella generazione di un file eseguibile o di un link simbolico verso una posizione del workspace in cui si effettuano le build: ciò semplifica notevolmente le cose quando si sta testando il software ed eseguendo più volte compilazioni dello stesso, evitando di mettere ogni volta mano ai punti in cui si installa rimandando sempre all'eseguibile più recente. Le molte opzioni da riga di comando di colcon consentono infine di interagire, se necessario, con i tool di più basso livello usati ai vari stadi della compilazione.

Oltre a normali problematiche inerenti la costruzione di pacchetti interdipendenti, bisogna risolvere quelle legate all'integrazione con le librerie e le interfacce di ROS 2. A ciò occorre il secondo modulo: *ament*, descritto in [8]. Si tratta di una collezione di script Python, invocati automaticamente da colcon, che terminano il processo di build risolvendo le dipendenze verso altri packages ROS 2, generando infine degli script che verranno eseguiti automaticamente prima dell'avvio dell'eseguibile generato, per assicurarsi che gli altri moduli di ROS 2 necessari siano operativi in un environment corretto.

Questo sistema risulta efficace: avendo vari pacchetti in un workspace, possono essere costruiti tutti insieme con un unico comando.

Interfacce

Di base, in ROS 2 come in ogni DDS, è possibile specificare il formato dei pacchetti dati scambiati in particolari file di testo denominati *interface files* [9]. In essi, si specifica

¹²eXtensible Markup Language, basato su un meccanismo sintattico che consente di definire e controllare il significato degli elementi contenuti in un documento o in un testo. Viene usato nelle pagine web o per collezionare dati in un file di testo in modo che siano anche interpretabili da un programma: il *parsing* può essere infatti eseguito facilmente grazie ad apposite librerie.

la struttura del pacchetto definendo in ogni riga un campo dello stesso, identificato da nome e tipo di dato; questi ultimi vanno dai numeri interi o in virgola mobile alle stringhe, fino agli arrays di lunghezza variabile. È buona prassi raggruppare questi file in pacchetti costituiti da sole interfacce, da costruire e installare mediante lo stesso build system: il risultato in questo caso sarà una collezione di classi Python, header files e librerie condivise C++ che specificheranno i tipi dei pacchetti dati e le operazioni di serializzazione e deserializzazione degli stessi, e che potranno essere inclusi negli altri moduli che dovranno scambiarsi dati secondo tali formati. Di fatto, lo sviluppatore ROS 2 non deve mai preoccuparsi di tutto ciò che riguarda specifica del formato, serializzazione, trasmissione, ricezione e deserializzazione: è tutto gestito automaticamente da ROS 2 mediante il DDS.

A differenza di un normale DDS però, ROS 2 consente anche di dare una semantica alle interfacce, che definisce le dinamiche in cui la comunicazione deve avvenire quando si segue tale specifico formato.

Indipendentemente dal topic creato e dal QoS impostato, si può distinguere tra:

- **messaggi:** normali pacchetti dati contenenti campi di tipi diversi, di trasmissione semplice ed immediata, specificati nei file *.msg*;
- **servizi:** si tratta di comunicazioni di tipo client-server divise in due tempi: dapprima un client invia una *richiesta* e in un secondo momento il server a cui l'ha inviata fornirà *risposta*, e il formato di entrambe è specificato in due diverse sezioni di un file *.srv*;
- **azioni:** sono dei servizi particolarmente indicati quando il processamento della richiesta può richiedere molto tempo, ed è bene che il client non stia sempre fermo ad aspettare ma interroghi quando può il server per controllare se c'è una risposta pronta; un file *.action* è diviso in tre sezioni, di cui quella in più rispetto al *.srv* serve per definire un feedback da dare finché la risposta non è ancora stata prodotta;

A patto di risolvere le dipendenze tra i pacchetti, interfacce definite in un file possono essere usate come campi di interfacce definite in altri file.

A titolo di esempio, per riassumere i concetti introdotti e dimostrare la semplicità di questo meccanismo, si riporta il listato del message file *Image.msg* della libreria *sensor_msgs*, relativo alla trasmissione di immagini.

```

# This message contains an uncompressed image
# (0, 0) is at top-left corner of image
#
Header header          # Header timestamp should be acquisition time of image
                      # Header frame_id should be optical frame of camera
                      # origin of frame should be optical center of camera
                      # +x should point to the right in the image
                      # +y should point down in the image
                      # +z should point into to plane of the image
                      # If the frame_id here and the frame_id of the CameraInfo
                      # message associated with the image conflict
                      # the behavior is undefined

uint32 height           # image height, that is, number of rows
uint32 width            # image width, that is, number of columns

# The legal values for encoding are in file src/image_encodings.cpp
# If you want to standardize a new string format, join
# ros-users@lists.sourceforge.net and send an email proposing a new encoding.

string encoding         # Encoding of pixels -- channel meaning, ordering, size
                      # taken from the list of strings in
                      # include/sensor_msgs/image_encodings.h

uint8 is_bigendian      # is this data big endian?
uint32 step              # Full row length in bytes
uint8[] data             # actual matrix data, size is (step * rows)

```

Listing 2.1: Definizione del messaggio *sensor_msgs/Image*.

Uso del DDS: context e nodi

Affinché un programma, scritto in Python o C++, possa usare i servizi offerti da ROS 2 e dal DDS ad esso sottostante, è necessario creare ed impiegare alcuni oggetti ad essi relativi. Anche alla luce della precedente discussione circa i DDS, appare chiaro che una delle prime azioni da compiere debba essere la creazione di un DDS participant, inteso sempre come un’istanza del DDS relativa all’applicazione corrente. In ROS 2, l’inizializzazione dell’istanza del middleware e del DDS participant viene eseguita creando un oggetto denominato *context*. È possibile farlo sia manualmente, magari specificando delle opzioni particolari, sia usando delle routine di default. Ad esempio, in C++ con rclcpp, all’inizio del programma solitamente si scrive la chiamata a funzione

```
rclcpp::init(argc, argv)
```

che esegue proprio dei task di inizializzazione della libreria stessa e del DDS, agendo in questo caso su un context generico globale nell’address space del programma ed usando eventuali argomenti specifici di ROS 2 passati da riga di comando. Esso, così come l’istanza del DDS, può essere terminato con una chiamata a

```
rclcpp::shutdown()
```

quando opportuno.

Il middleware e i servizi offerti da esso e dal DDS possono essere invocati ed usati solo fin quando c’è almeno un context attivo, facendovi riferimento. Il punto d’ingresso verso lo strato del middleware è rappresentato da un secondo oggetto chiamato *nodo*, la cui denominazione richiama volutamente il concetto di nodo in un DDS.

Concettualmente, nell’ecosistema di ROS, un nodo rappresenta un’entità in grado di svolgere dei compiti nel sistema. È infatti possibile definire un nodo ad-hoc estendendo una classe base offerta dalla libreria, aggiungendo i membri e i metodi necessari e facendo riferimento a un particolare context o a quello globale. In questo senso una complessità eccessiva è scoraggiata, in quanto è sempre preferibile che un nodo si occupi di una e una sola cosa.

Può sembrare che un nodo ROS sia anche praticamente un’unità operativa, ma in

realtà si tratta solamente di un costrutto che materializza un'applicazione nello strato del middleware, anche in modo potenzialmente non univoco: un'applicazione può infatti contenere e gestire contemporaneamente più nodi. Essenzialmente un nodo è una struttura dati atta a contenere, principalmente, dei DDS endpoints, relativi a vari topic e che in ROS 2 possono essere usati come:

- publisher o subscriber di messaggi;
- client o server di servizi;
- client o server di azioni;

tutti potenzialmente entro lo stesso nodo. A questi ovviamente si possono aggiungere altri oggetti propri di ROS 2, come ad esempio dei timer. Le operazioni vere e proprie sono svolte dal codice dell'applicazione, che usa il nodo per interagire col middleware. A titolo di esempio si riporta il listato in C++ della definizione di due semplici nodi che fungano da server e client di un servizio che calcola la somma di due numeri interi.

```
/* AddTwoInts server */
class AddTwoIntsServer : public rclcpp::Node
{
public:
    AddTwoIntsServer();

private:
    void add_two_ints_clbk(const AddTwoInts::Request::SharedPtr request,
                           const AddTwoInts::Response::SharedPtr response);
    rclcpp::Service<AddTwoInts>::SharedPtr server_;
};

/* AddTwoInts client */
class AddTwoIntsClient : public rclcpp::Node
{
public:
    AddTwoIntsClient();
    void call_srv(int a, int b);

private:
    rclcpp::Client<AddTwoInts>::SharedPtr client_;
};
```

Listing 2.2: Definizione delle classi dei nodi server e client del servizio *AddTwoInts*.

Programmazione a eventi e concorrente

Una volta definito e creato un nodo, il software di cui è composto il modulo in questione può iniziare ad operare secondo la sua programmazione ed usando il nodo per accedere ai servizi offerti dal middleware, ad esempio per pubblicare messaggi. Ciò nonostante, ROS 2 stabilisce uno standard per la gestione di alcune operazioni:

- ricezione di messaggi;
- ricezione di richieste di servizi;
- ricezione di richieste di azioni;
- scatti di timer periodici, che possono essere creati ed attivati sempre attraverso un nodo.

Tutte queste operazioni sono implementate per essere gestite come degli eventi a cui rispondere con delle *callbacks*¹³, in modo totalmente alternativo rispetto al resto delle operazioni compiute dal modulo software.

Dal punto di vista del middleware, queste callback sono delle unità di lavoro da schedulare opportunamente, e la loro definizione viene fatta usando metodi offerti dall'oggetto nodo in modo che esso porti con sé informazioni circa il lavoro, periodico o meno, che richiede di svolgere. Tale lavoro può essere schedulato eseguendo un'operazione che nelle librerie standard di ROS 2 è denominata *spin*. L'entità che prende in carico la richiesta di spin, creando delle code di callback da attivare, è chiamata *executor*. Il modo più semplice di fare ciò in C++ è mediante la chiamata:

```
rclcpp::spin(nodo);
```

dove *nodo* è un'istanza della classe *rclcpp::Node* o sua derivata.

Agli executors possono essere associati anche più nodi, e ve ne sono di due tipi:

¹³Una callback è, in genere, una funzione che viene passata come parametro ad un'altra funzione, ed è richiamata ogni qual volta si verifica un particolare evento, in modo potenzialmente asincrono e casuale.

- **Single Threaded:** sono quelli standard, creati automaticamente durante chiamate come la precedente; ogni callback di eventi relativi ai nodi associati a tale executor sarà inserita in un'unica coda, eseguita da un solo thread¹⁴;
- **Multi Threaded:** rispetto ai precedenti, offrono la possibilità di suddividere le callback in diversi *callback groups*, a ciascuno dei quali corrisponderà una diversa coda d'esecuzione gestita da un diverso thread;

Il secondo tipo costituisce di fatto il supporto alla programmazione concorrente offerto da ROS 2, consentendo anche di specificare se callback di uno stesso gruppo possano essere eseguite in parallelo o siano mutuamente esclusive. La definizione dei gruppi può essere fatta fornendo particolari opzioni all'atto della creazione di publishers e subscribers; si rimanda alla documentazione tecnica per ulteriori dettagli.

Launch System e Launch Files

Un eseguibile ROS 2 va avviato usando particolari comandi del middleware, per assicurarsi di attivare tutti i demoni¹⁵ eventualmente necessari e i sottosistemi di logging. Il sottosistema del middleware incaricato di fare ciò è chiamato *launch system*. Esistono due modalità:

- avvio diretto tramite i comandi del launch system;
- avvio indiretto tramite i *launch files*;

Nel primo caso il comando ha la forma seguente:

```
ros2 run PACKAGE EXECUTABLE
```

a cui possono fare seguito gli argomenti di input per l'eseguibile stesso, quelli per la libreria con l'opzione *-ros-args*, più eventuali prefissi per l'impiego di debugger.

¹⁴Un thread o thread di esecuzione, in informatica, è una suddivisione di un processo in due o più filoni (istanze) o sottoprocessi che vengono eseguiti concorrentemente da un sistema di elaborazione monoprocesso (monothreading) o multiprocesso (multithreading) o multicore.

¹⁵Un demone, in informatica e più in generale nei sistemi operativi multitasking, è un programma eseguito in background, cioè senza che sia sotto il controllo diretto dell'utente, tipicamente per fornire un servizio.

Nel secondo caso invece si sostituisce al nome dell'eseguibile quello di uno script Python che fa parte del package. In esso si specifica, secondo un formato per cui si rimanda alla documentazione tecnica¹⁶, la configurazione con cui va avviato l'eseguibile, completa di dettagli che vanno dagli argomenti di input fino a come deve essere gestito il logging in console o su file, o entrambi. Eventuali log file generati dall'output del processo, a meno che non si specifichi il contrario, saranno salvati in un percorso standard, tipicamente una subdirectory della home directory (e.g. `./ros/` su Linux).

Data recording e playback

Per diverse finalità quali testing o esame a posteriori, può essere necessario acquisire tutti i messaggi che sono stati gestiti dal middleware in un dato intervallo di tempo e su specifici topic. ROS 2 offre questo servizio mediante il sistema del *bagging*¹⁷: attraverso specifici comandi è possibile avviare una registrazione dei messaggi inviati su determinati topic in un file, di cui si possono specificare formato e algoritmo di compressione. Tale file, accompagnato anche da informazioni circa le interfacce dei messaggi che contiene, può essere processato in un altro software per visionare i dati. Le bags possono anche essere riprodotte interamente daccapo con un altro comando del middleware, che crea dei publisher ad-hoc i quali pubblicheranno nuovamente, ad un rate uguale o impostato, i messaggi registrati.

Visualizzazione e simulazione

Un'installazione completa di ROS 2 porta con sé anche una serie di strumenti utili per il debugging, quali comandi per ispezionare i topic attivi, le loro interfacce, misurare il rate di pubblicazione e avviare dei publisher al bisogno. Vi sono poi dei software non necessari al funzionamento di un robot, e di fatto superflui in tal caso, ma molto utili in fase di sviluppo. È doveroso citare almeno i seguenti:

¹⁶<https://docs.ros.org/en/foxy/Tutorials/Launch-Files/Creating-Launch-Files.html>

¹⁷<https://docs.ros.org/en/foxy/Tutorials/Ros2bag/Recording-And-Playing-Back-Data.html>

- **Gazebo¹⁸:** si tratta di un simulatore open-source nativamente compatibile con ROS, in grado di interagire con i topic del middleware per acquisire e inviare dati. In esso è possibile costruire, mediante una sintassi basata su XML, modelli 3D di robot completi di sensori e attuatori e degli ambienti in cui essi si muovono, e testare gli algoritmi di controllo.
- **RViz2:** È la versione per ROS 2 del visualizzatore robotico RViz. Supporta tipiche interfacce legate ai sensori (e.g. immagini, pointclouds) e permette di leggere in tempo reale dai topic, offrendo una visualizzazione 3D delle informazioni raccolte. È particolarmente utile quando configurato per ascoltare dagli stessi topic su cui pubblicano i sensori montati su un robot, in quanto consente di "vedere" l'ambiente nello stesso modo in cui quest'ultimo lo percepisce.
- **rqt_graph ed rqt_plot:** Sono due visualizzatori di informazioni legate ai topic. rqt_graph consente di ottenere uno schema, più o meno dettagliato, dei nodi presenti nel sistema e nel dominio correnti, dei loro publishers e subscribers e più in generale di topic e servizi attivi, specificando quali siano le associazioni; un esempio con due nodi che si scambiano stringhe sul topic `/chatter` è riportato in Figura 2.2. rqt_plot consente invece di sottoscriversi ai topic su cui si trasmettono dati numerici e visualizzarli in tempo reale in grafici 2D.



Figura 2.2: Semplice esempio di schema generato da *rqt_graph*.

¹⁸<http://gazebosim.org/>

Capitolo 3

Caso di studio: drone autonomo

A seguito della discussione portata a termine nei capitoli precedenti circa le nuove soluzioni hardware e software da impiegare per la costruzione di apparati robotici, verrà ora descritto un caso di studio pratico con cui si dimostreranno la validità e l'efficacia di tali strumenti: un drone volante automatico.

I task che tale sistema deve svolgere, oltre naturalmente a quelli inerenti il volo in sé, sono specificati nelle regole dell'edizione 2021 del Drone Contest indetto da Leonardo S.p.A. e tenutosi presso la Divisione Velivoli di Leonardo in Corso Francia, Torino. Il prototipo è stato presentato in tale occasione come la proposta del team Asgard Flight Group dell'Università di Roma "Tor Vergata". Il progetto ha coinvolto in tutto cinque tra tesisti e dottorandi afferenti al Dipartimento di Ingegneria Civile e Ingegneria Informatica, i quali sono citati all'inizio di questa Tesi assieme ai loro ruoli e ad un sentito e personale ringraziamento, sotto la supervisione del Prof. Daniele Carnevale. Le regole dell'edizione 2021 del Contest, che costituiscono gli obiettivi operativi del drone, possono essere riassunti come segue:

- il drone deve essere in grado di decollare, volare e atterrare autonomamente, orientandosi all'interno di un ambiente indoor di cui è nota a priori la conformazione e la posizione delle piazze di decollo e atterraggio;
- inizialmente il drone deve esplorare l'ambiente, cercando ed individuando dei target costituiti da robot mobili che montano un landmark ArUco;

CAPITOLO 3. CASO DI STUDIO: DRONE AUTONOMO

- sono noti solo alcuni dei landmark montati sui robot, dunque quando il drone ne individuasse uno sconosciuto, dovrà scattare ed inviare delle fotografie di esso ad una *Ground Control Station* controllata da un operatore;
- sulla base di una stringa di punteggi segnata vicino al landmark del robot, dovrà essere decisa una sequenza di atterraggi sulle varie piazzole;
- una volta trasmessa la sequenza al drone, esso dovrà eseguire i vari atterraggi, riconoscendo le piazzole sapendone la posizione nella mappa e rilevando i landmark ArUco dipinti su di esse;
- durante tutta la durata del volo deve essere possibile inserire il controllo manuale, per ragioni di sicurezza;

Per brevità sono stati tralasciati dei dettagli del regolamento inerenti il solo funzionamento della gara. L'obiettivo finale del Contest è chiaramente la massimizzazione del punteggio ottenuto con gli atterraggi, resa non banale dalla presenza nella mappa di ostacoli, rialzi e zone a bassa visibilità.

Nel resto di questo capitolo si descriverà più nel dettaglio il robot, partendo dall'hardware e passando poi al software, da quello di più basso livello fino alle logiche di supervisione e comunicazione con GCS ed operatori.

Il prototipo realizzato è ritratto in Figura 3.1, mentre una mappa completa del campo di gara è rappresentata in Figura 3.2.

CAPITOLO 3. CASO DI STUDIO: DRONE AUTONOMO



Figura 3.1: Prototipo del drone (photo credit: Lorenzo Bianchi).

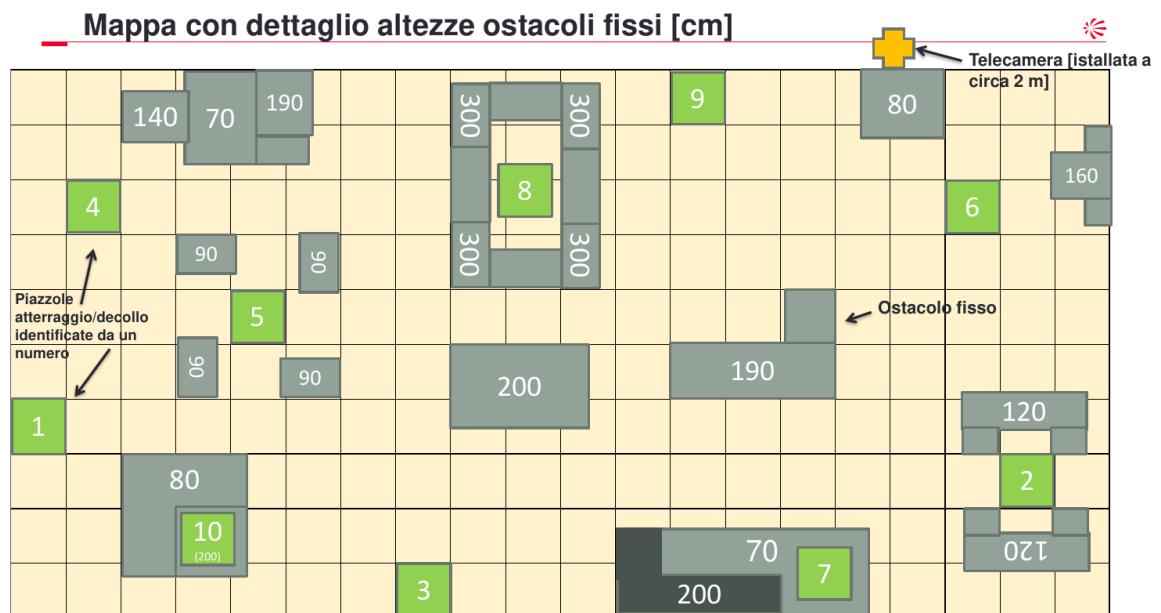


Figura 3.2: Mappa completa del campo di gara.

Hardware

Frame e motori

La primissima decisione progettuale ha riguardato la scelta del *frame*, inteso come struttura portante, su cui montare le varie parti. Nell'economia di un drone volante, in special modo autonomo quindi in grado di stabilizzarsi da solo grazie al feedback di vari sensori di bordo, tale struttura deve essere il più leggera e rigida possibile: il primo requisito riguarda naturalmente le prestazioni, l'autonomia e la controllabilità, mentre il secondo ha risvolti sull'accuratezza delle misure di accelerometri e giroscopi. È infatti comune rilevare durante il volo, mediante uno spettrogramma ricavato dai campioni degli accelerometri, delle vibrazioni dovute a vari fattori. Un esempio è mostrato in Figura 3.3, relativo ad una prova di volo automatico in pattugliamento. Prendendola ad esempio, si possono notare dei contributi frequenziali predominanti intorno ai 90 Hz, correlati ad altri precedenti nello spettro e di ampiezza minore: sono le armoniche dei motori. Tali vibrazioni non possono naturalmente essere eliminate, e vanno filtrate e reiezione dal sistema di controllo e stabilizzazione angolare. Alla fine dello spettrogramma (a circa 6:40) si nota quello che sembra un breve urto, che corrisponde al contatto col suolo in fase di atterraggio, in questo caso preciso e privo di rimbalzo. Eventuali altri contributi significativi, qui assenti, sarebbero indice di problemi meccanici o tarature sbagliate del sistema di controllo.

Avendo a che fare con questo genere di prototipo però, è necessario anche uno spazio sufficiente ad ospitare i sistemi elettronici, i sensori e le batterie. Per questo la scelta è ricaduta sul frame Tarot 650, realizzato in carbonio e dotato di una piattaforma centrale sufficientemente ampia da permettere il montaggio dell'elettronica di bordo ricavando due livelli superiori e di un pacco batterie in basso. L'intera struttura è stata pensata avendo cura di spostare il meno possibile lungo la verticale il centro di massa aggiungendo carico al frame: montando la batteria in basso questo tende naturalmente a scendere, imprimendo al sistema un comportamento indesiderato simile a quello di un pendolo composto, caratterizzato da oscillazioni difficili da contrastare. A seguito delle diverse prove e dei vari incidenti di percorso, sono stati aggiunti dei pezzi volti a irrobustire i punti più fragili e ridurre ancor di più le vibrazioni. Le ulteriori parti

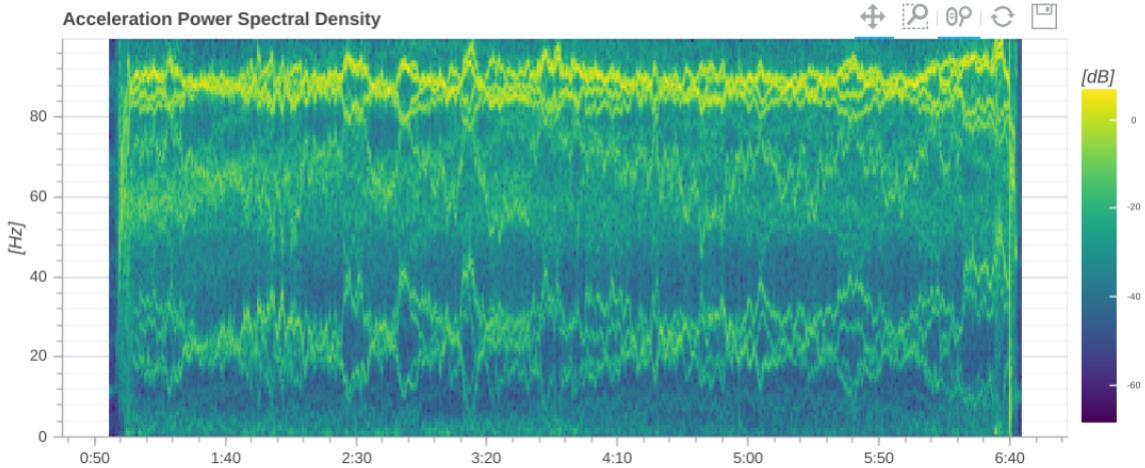


Figura 3.3: Esempio di spettrogramma ricavato dai dati di un volo automatico.

meccaniche necessarie sono state modellate con strumenti CAD e stampate in 3D in PLA e TPU: degli esempi particolarmente rilevanti sono mostrati nelle Figure 3.4, 3.5, 3.6, 3.7.

Per fornire spinta sono stati montati quattro motori brushless T-Motor U3 da 700 W e 700 rpm/V, dotati di eliche da 12 pollici. Essi sono pilotati dal controllore di volo mediante delle ESC¹ F35A BLHeli_32 digitali a 32 bit con protocollo DShot, configurato per trasmissione a 1200 kpbs. Tale protocollo è particolarmente indicato per la trasmissione di comandi agli attuatori dei droni in quanto garantisce bassissima latenza, robustezza agli errori sui bit, e permette di comunicare direttamente con le ESC per configuarle, consentendo ad esempio di impostare il verso di rotazione dei motori senza doverne dissaldare i cavi per invertire la polarità.

Controllore di volo

La prima parte dell'elettronica di bordo è costituita da un sistema hard real-time² di basso livello, collegato direttamente a sensori e motori, con il compito di localizzarsi in tempo reale e stabilizzare il drone durante le diverse fasi del volo. Per il presente lavoro

¹Electronic Speed Controller.

²Con tale locuzione si indica, in genere, un calcolatore elettronico o un controllore numerico per il quale sono specificati una serie di task, ciascuno con delle scadenze temporali precise, e la cui programmazione e realizzazione devono consentire il rispetto tassativo di tali scadenze.

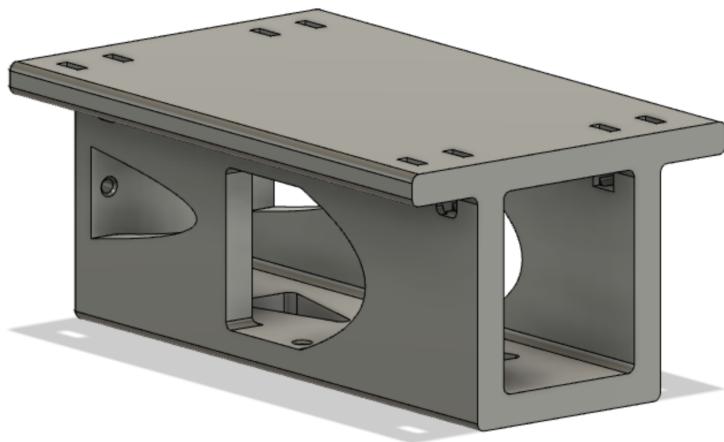


Figura 3.4: Modello CAD del pacco batteria, che funge anche da supporto per la camera inferiore.

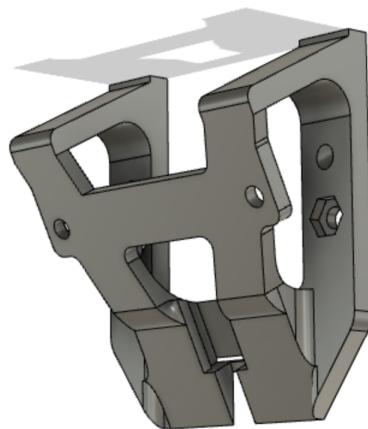


Figura 3.5: Modello CAD del supporto della camera frontale, inclinata verso il basso di 26.6° .

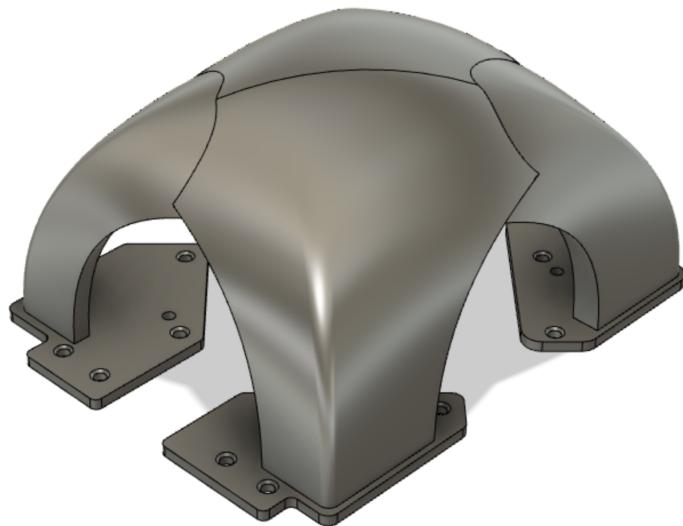


Figura 3.6: Modello CAD della canopy superiore in TPU, a protezione del computer di bordo.

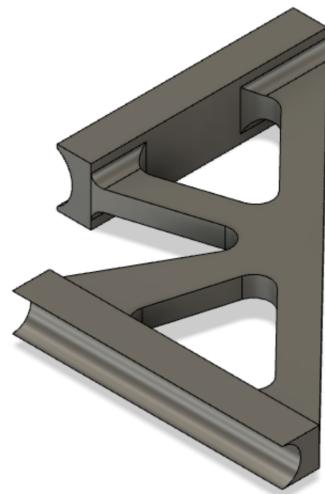


Figura 3.7: Modello CAD del rinforzo triangolare per i carrelli del frame.

CPU	STM32F765 32 Bit Arm Cortex-M7
I/O Processor	STM32F100 32 Bit Arm Cortex-M3
Sensori	IMU ⁴ , magnetometro, barometro, GPS
Interfacce	UART, GPIO, I2C, SPI, CAN, USB

Tabella 3.1: Caratteristiche tecniche del controllore di volo Pixhawk 4.

la scelta è ricaduta sull’Holybro Pixhawk 4, mostrato in Figura 3.8, board compatibile con il firmware open-source PX4 Autopilot³, scritto in C++; la versione qui scelta è la 1.12-beta5. Le caratteristiche tecniche principali della board sono riassunte nella Tabella 3.1.

La CPU è dedicata all’esecuzione del firmware PX4 sul Real-Time Operating System NuttX⁵, mentre l’I/O processor all’acquisizione dei dati dai sensori e alla gestione delle interfacce di comunicazione verso ESC o altri componenti. Come da regolamento del Contest, il GPS è stato disattivato e scollegato. Alla board è stata collegata una ricevente radio per aggiungere il controllo manuale con l’ausilio di un radiocomando FrSky Taranis.

La caratteristica saliente di questo controllore è la compatibilità con il progetto PX4. Quest’ultimo, essendo open-source, ha consentito di imparare molto circa il funzionamento e il controllo di un drone, ma anche di intervenire sulla configurazione in modo mirato quando necessario, per eseguire tarature o identificare e risolvere i problemi. I principali servizi offerti da questo firmware, che sono stati ritoccati ed impiegati come parte di questo lavoro per mettere in volo il prototipo secondo le specifiche definite dal Contest, sono i seguenti:

- possibilità di controllare il velivolo in posizione, velocità, accelerazione o assetto tramite un computer di bordo esterno;
- fusione delle misure acquisite da diversi sensori, sia integrati che esterni ed anche temporalmente sfasati, mediante un filtro di Kalman esteso denominato EKF⁶;
- link di comunicazione seriale robusto ad alta velocità verso computer di bordo;

³<https://px4.io/>

⁵<https://nuttx.apache.org/>

⁶https://docs.px4.io/master/en/advanced_config/tuning_the_ecl_ekf.html

- filtri dinamici regolabili per rigettare dalle misure vibrazioni spurie o altri disturbi esterni.

L’algoritmo di controllo nonlineare implementato da PX4 è descritto in [10], e una sua schematizzazione è mostrata in Figura 3.9, con dettagli nelle Figure 3.10, 3.11, 3.12, 3.13. A ciascun livello, ogni controllore prende in ingresso un riferimento da inseguire e calcola un controllo, da applicare al livello successivo oppure direttamente ai motori tramite opportuno mixing. Ai fini del Contest, tra le modalità disponibili sono state impiegate quelle in posizione e in velocità, a seguito di uno studio accurato del sistema e di una precisa taratura dei suoi vari guadagni e saturazioni.

Il sistema di comunicazione con il computer di bordo è invece costituito da un bridge tra due DDS, denominato *microRTPS Bridge*, comprensivo di meccanismi di serializzazione e deserializzazione dei pacchetti in transito sul link seriale. Nello specifico, il firmware integra il DDS basico *uORB*, usato anche per le varie comunicazioni interprocesso nel firmware stesso, mentre l’altro lato della comunicazione è basato su *FastDDS*⁷ di eProxima. Le due parti della comunicazione seriale sono gestite da due software, uno in esecuzione sul controllore di volo e per questo denominato *Client*, ed uno presente sul computer di bordo e chiamato *Agent*.

Tale soluzione assicura una comunicazione affidabile, full-duplex e ad alta velocità tra i due componenti, nonché la semplice integrazione con altri middleware quali ad esempio ROS 2. Anche questo progetto è open-source, ed è stato modificato in svariati punti durante questo lavoro al fine di produrre una versione migliore di quella ufficiale, nella quale sono state rilevate gravi mancanze nella gestione di eventuali pacchetti malformati, che potevano portare casualmente al crash del programma. Una schematizzazione dell’architettura di comunicazione è mostrata in Figura 3.14.

Fotocamere

Al fine di riconoscere i target montati sui robot mobili a terra, nonché di localizzarsi nell’ambiente, si sono rese necessarie due fotocamere, montate rispettivamente di fronte e verso il basso. La scelta è ricaduta sulle Intel RealSense D435i (Figura 3.15), dotate

⁷<https://www.eprosima.com/index.php/products-all/eprosima-fast-dds>



Figura 3.8: Controllore di volo Pixhawk 4.

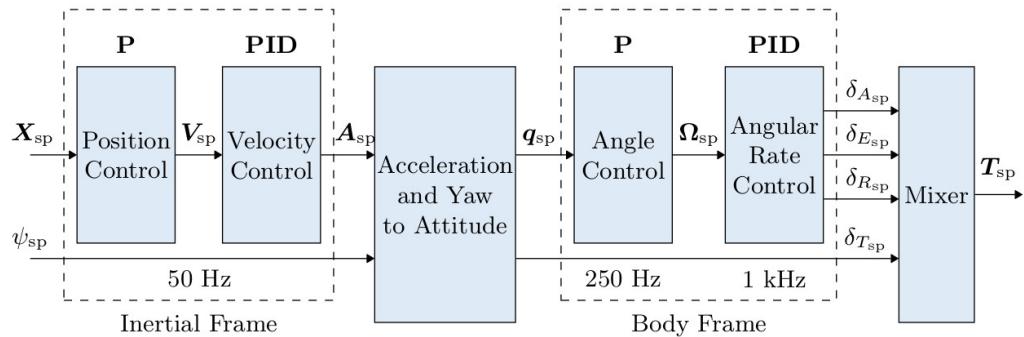


Figura 3.9: Schema del controllore di volo multilivello implementato da PX4, con i diversi tempi di campionamento dei vari sottosistemi.

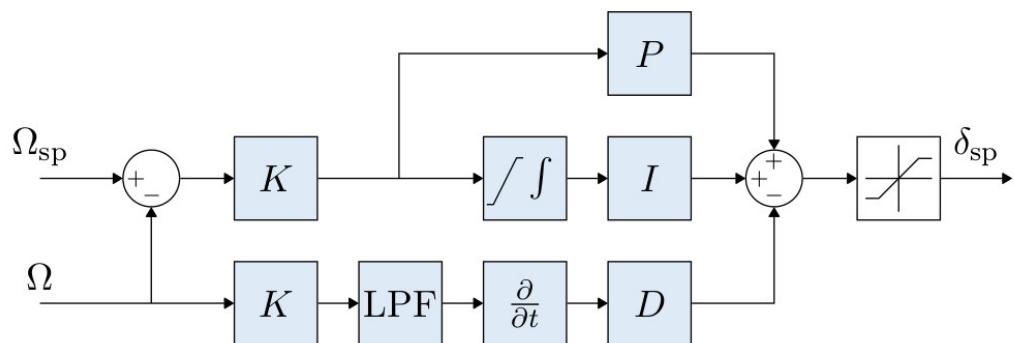


Figura 3.10: Dettaglio del controllore PID con filtri per la stabilizzazione angolare.

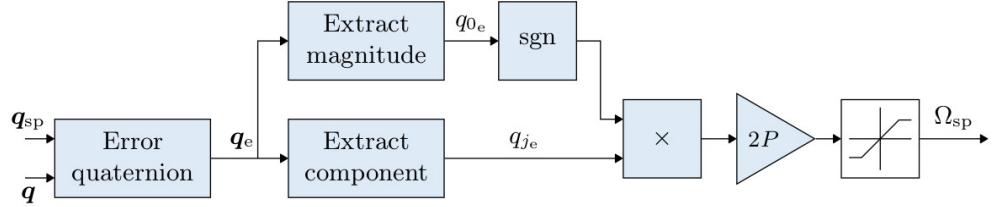


Figura 3.11: Dettaglio del controllore di PX4 per la regolazione dell'assetto basato sui quaternioni d'orientamento.

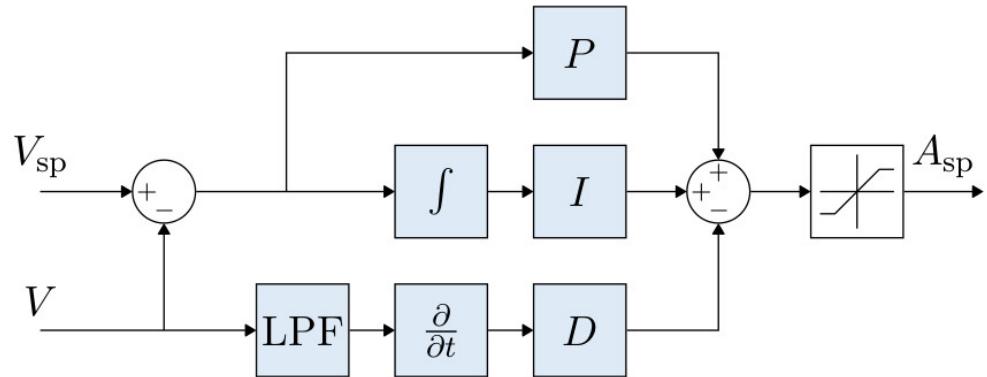


Figura 3.12: Dettaglio del controllore PID con filtri per la regolazione della velocità lineare.

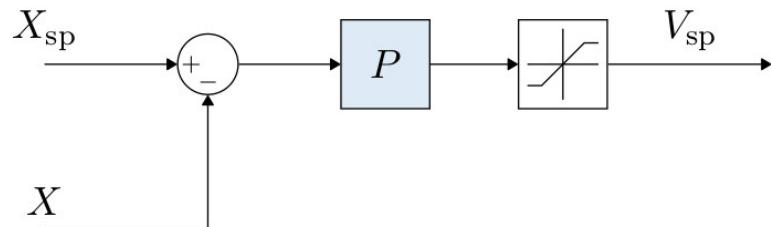


Figura 3.13: Dettaglio del controllore proporzionale per la regolazione della posizione.

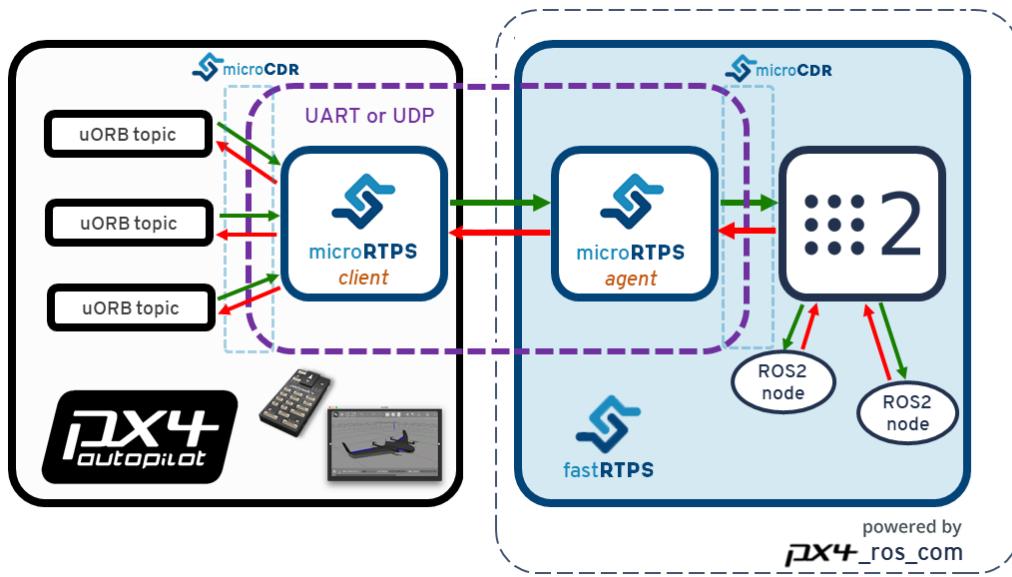


Figura 3.14: Architettura del link di comunicazione microRTPS Bridge.



Figura 3.15: Intel RealSense D435i stereo depth camera.

di interfaccia USB 3.1 e vari sensori, dei quali sono stati usati le camere RGB e ad infrarossi per ricavare una mappa di profondità. L'integrazione col resto del software è stata possibile mediante dei driver open-source resi disponibili da Intel.

Computer di bordo

La restante parte dell'elettronica è costituita dal computer di bordo. Si tratta in questo caso di un calcolatore di più alto livello, dedicato allo svolgimento dei task più lenti di supervisione e decisione, nonché ai compiti di processamento più onerosi per ottenere una stima precisa della posa locale, implementando le logiche di navigazione.

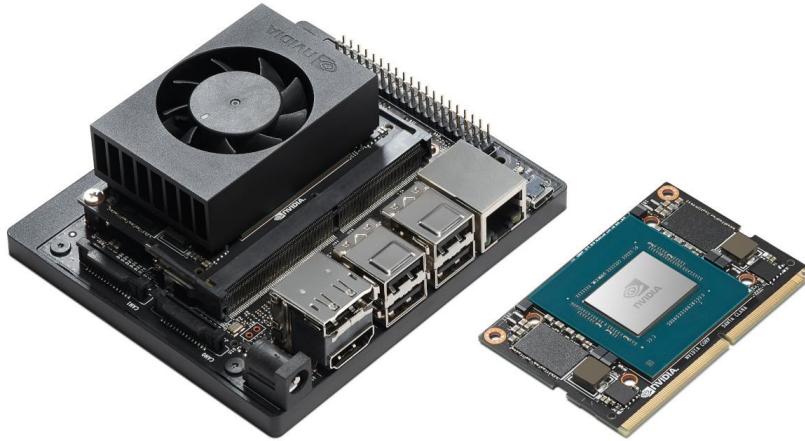


Figura 3.16: SoC Nvidia Jetson Xavier NX.

Le piattaforme Nvidia Jetson sono state selezionate per questo scopo, in quanto soluzioni SoC altamente efficienti e dotate di svariato hardware integrato, predisposte anche per lo sviluppo di software di qualunque tipo grazie al supporto del sistema operativo Linux. Il modello specifico che è stato adottato è la development board Xavier NX, rappresentata in Figura 3.16. La principale caratteristica di queste soluzioni è l'ampia e varia disponibilità di risorse in un package di dimensioni e consumi contenuti. Le caratteristiche salienti e più utili ai fini del presente lavoro sono riassunte nella Tabella 3.2.

CPU	6-core NVIDIA Carmel ARM v8.2 64-bit
GPU	NVIDIA Volta architecture, 384 CUDA cores, CUDA 10.1
RAM	8 GB 128-bit LPDDR4x
Interfacce	Gigabit Ethernet, WiFi, Bluetooth, GPIO, I2C/S, SPI, UART, USB

Tabella 3.2: Caratteristiche tecniche del SoC Nvidia Jetson Xavier NX.

Operating System

Appare chiaro dall'architettura illustrata finora come in questo progetto si sia reso necessario interagire con l'hardware a basso livello, e contemporaneamente sviluppare un software stratificato che lo pilotasse e implementasse opportune logiche decisionali.

Per far questo, è apparsa da subito chiara la necessità di un sistema operativo che consentisse di operare facilmente ad ogni livello. Le soluzioni Jetson, basate su architettura ARM, sono studiate per essere utilizzate con il sistema operativo Linux, che soddisfa totalmente tali requisiti operativi. Ciò nonostante, è stato necessario un lavoro di configurazione ad-hoc a partire dall'installazione fornita da Nvidia per meglio adattarla al presente caso d'uso e consentire l'integrazione del resto dell'hardware e degli strumenti software impiegati.

L'installazione di default offerta da Nvidia è costituita dalla distribuzione Ubuntu Linux 18.04 con kernel 4.9 PREEMPT⁸, racchiusa in una collezione di package denominata *Linux 4 Tegra*. Si tratta di un'installazione stabile e testata, dotata di un set di pacchetti e librerie più limitato rispetto alle versioni base ma di tutti i driver necessari ad usare l'hardware presente sulla board. La prima strada intrapresa è stata quella dell'aggiornamento a Ubuntu 20.04, ancora non standardizzata da Nvidia, lungo la quale sono sorti diversi problemi legati all'aggiornamento delle librerie grafiche pian piano risolti. Successivamente, allo scopo di aumentare ancora di più la responsività e la predicitività del sistema, è stata tentata la costruzione di un'installazione stabile con kernel 4.9 PREEMPT RT⁹. Nonostante ripetuti tentativi di ricompilazione del kernel e di aggiornamento del sistema, sono sempre stati rilevati dei problemi di compatibilità con i driver e le librerie offerti da Nvidia, e gli incrementi prestazionali ricavati sono stati giudicati eccessivamente ridotti; oltre a ciò, si sospetta che l'aver reso interamente interrompibili anche i driver della GPU abbia reso meno efficienti alcuni algoritmi implementati su di essa. Per tali ragioni questa strada è stata abbandonata, e l'installazione PREEMPT è stata consolidata con delle performance accettabili sia sul fronte della varianza dei timer ad alta risoluzione, sia su quello dell'affidabilità delle schedulazioni soft real-time implementate mediante lo scheduler di sistema. Si ritiene, come si mostrerà, che un tale sistema sia adatto anche all'implementazione di loop di controllo veloci con campionamenti non inferiori al singolo millisecondo, naturalmente quando non si è complessivamente vicini ai suoi limiti di carico.

⁸La versione PREEMPT del kernel Linux è caratterizzata da latenze ridotte, in quanto fatta eccezione per alcune sezioni molto critiche esso risulta sempre interrompibile.

⁹La versione PREEMPT RT del kernel, ottenuta mediante l'applicazione di un set di patches prima della compilazione, è caratterizzata dall'interrompibilità totale in qualunque punto e dall'assenza di primitive di sincronizzazione di tipo *locking*.

Per misurare le latenze e valutare le prestazioni complessive è stata impiegata la utility *cyclictest*¹⁰, una cui esecuzione è mostrata a titolo di esempio nel listato seguente.

```
nxdrone:~$ sudo cyclictest -m -p98 --smp
# /dev/cpu_dma_latency set to 0us
policy: fifo: loadavg: 0.33 0.61 0.32 1/381 10460

T: 0 (10428) P:98 I:1000 C: 3365 Min:      7 Act:    30 Avg:    19 Max:    147
T: 1 (10431) P:98 I:1500 C: 2241 Min:      7 Act:    21 Avg:    22 Max:    294
T: 2 (10432) P:98 I:2000 C: 1680 Min:      8 Act:    16 Avg:    27 Max:    352
T: 3 (10434) P:98 I:2500 C: 1343 Min:      8 Act:    23 Avg:    22 Max:    155
T: 4 (10435) P:98 I:3000 C: 1118 Min:      7 Act:    30 Avg:    25 Max:    147
T: 5 (10436) P:98 I:3500 C: 958 Min:      9 Act:    22 Avg:    24 Max:    319
```

Listing 3.1: Esempio di benchmark del SoC montato sul drone eseguito con la utility *cyclictest*.

Da tale output si evince come le latenze siano comunque contenute, in media nell’ordine delle decine di microsecondi, con dei picchi attesi data la configurazione del sistema, dipendenti dal carico ma comunque mai eccessivi.

La procedura di configurazione del sistema e del SoC in ambiente Linux, nonché i vari passaggi dell’installazione del resto del software applicativo che verrà tra poco descritto, sono stati interamente automatizzati mediante appositi script per la shell Bash: nel giro di un paio d’ore ed in modo quasi del tutto automatico, una nuova board può essere predisposta per il montaggio sul drone.

¹⁰<https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start>

Software

La realizzazione di un drone autonomo in grado di eseguire deterministicamente e con precisione i task specificati nel regolamento del Contest ha posto come prima problematica la scelta di un'architettura software efficiente, e anche comoda da usare sia durante lo sviluppo che in fase di testing. Per ragioni che a questo punto dovrebbero risultare evidenti, all'installazione di Linux descritta nella sezione precedente è stato aggiunto il middleware ROS 2 versione Foxy Fitzroy.

I vari task necessari al compimento di una missione sono stati suddivisi in livelli:

- a basso livello: comunicazione con il controllore di volo Pixhawk per invio comandi, ricezione di feedback e informazioni di stato e campionamento dell'odometria;
- ad un livello intermedio: comunicazione con le camere per configurarle e ricevere i frame da cui estrarre informazioni;
- ad un livello più alto: algoritmi di esplorazione, navigazione e precision landing, logica decisionale e comunicazione con la Ground Control Station.

Ciascuno di questi moduli è stato realizzato come package ROS 2, comprendente uno o più nodi secondo le necessità. Ciò che li differenzia è la diversa modalità operativa: i nodi appartenenti al livello più basso hanno quasi sempre callbacks in esecuzione o eventi da gestire, e fanno un uso quasi esclusivo dei messaggi; quelli appartenenti al livello intermedio svolgono lavoro non appena sono disponibili nuovi dati da processare, pubblicando i risultati sempre sotto forma di messaggi; infine, le logiche e gli algoritmi di più alto livello sono attivati solo quando richiesto, ossia in momenti specifici della missione, e sono pertanto invocati sempre mediante dei servizi. Le interfacce che si è dovuto definire ad-hoc sono state raccolte nel package *afg_interfaces*, mentre quelle proprie di PX4 si possono trovare nel package *px4_msgs*¹¹. L'implementazione corretta del microRTPS Bridge è stata raccolta in un fork del package *px4_ros_com*¹². Data la criticità di alcune operazioni, nonché la necessità di lavorare talvolta a diretto contatto con l'hardware a disposizione, l'unico linguaggio di programmazione impiegato è stato

¹¹https://github.com/PX4/px4_msgs

¹²https://github.com/Automazione-Tor-Vergata/px4_ros_com

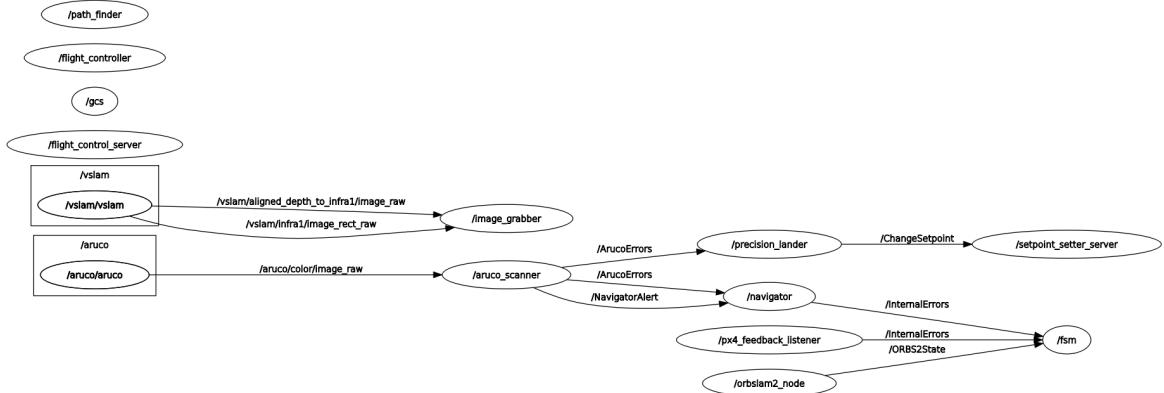


Figura 3.17: Schema complessivo di nodi e topic attivi su drone e GCS.

il C++. Nel resto di questa sezione si procederà dunque a descrivere nel dettaglio tutti i packages e i nodi che compongono l’architettura, secondo il naturale ordinamento gerarchico appena illustrato. Una rappresentazione complessiva di tale architettura è offerta in Figura 3.17: lo schema è stato ricavato con il tool rqt_graph¹³ ed evidenzia tutti i nodi attivi sul drone e la GCS, nonché i topic su cui alcuni di essi si scambiano messaggi. Sono assenti i servizi, non rappresentabili graficamente dal tool, nonché i topic esposti dal microRTPS Agent su cui viaggiano le comunicazioni da e verso il controllore di volo: tutti questi elementi saranno illustrati più chiaramente nel seguito. Ciascuno dei moduli che compongono questa architettura offre, sia all’operatore che agli altri moduli, un set di comandi invocabili mediante richieste ad opportuni servizi. Per testare singolarmente le varie unità senza il controllo della logica di supervisione, sono stati redatti svariati alias e funzioni per la shell Bash al fine di semplificare l’impiego dei comandi del middleware nei vari casi; un esempio è mostrato nel Listing 3.2.

¹³Questo tool talvolta non esclude alcuni elementi interni al middleware dai diagrammi, dunque in alcuni di essi figureranno nodi o topic propri di ROS 2 o RQt stesso.

```
# Routine to set a position setpoint
function fc_setp_pos {
    if [[ $# -ne 4 ]]; then
        echo "Usage: fc_setp_pos X Y Z YAW" 1>&2
        return 1
    fi
    ros2init
    ros2 service call /SetpointSetter afg_interfaces/srv/SetpointSetter\
    "{setpoint_conf: 0, x: $1, y: $2, z: $3, yaw: $(degrad $4)}"
}
```

Listing 3.2: Esempio di funzione Bash per cambiare setpoint di posizione corrente.

Flight Control

Il primo livello dell’architettura software, direttamente al di sopra del firmware PX4 in esecuzione nel controllore di volo Pixhawk, è rappresentato dal package *Flight Control*. I compiti svolti da questo programma possono essere riassunti come segue, e sono stati ripartiti tra un totale di quattro nodi esattamente nello stesso modo:

- invio di comandi a PX4 per l’esecuzione di operazioni relative al volo quali armamento, disarmamento, decollo e atterraggio, quando richiesto da altri nodi o da un operatore;
- invio periodico a PX4 di setpoint di posizione o velocità da inseguire;
- parsing delle richieste formulate dagli altri nodi per cambiare i setpoint da inviare a PX4, al fine di spostare il drone nell’ambiente;
- ricezione e parsing di feedback da PX4 relativi ai comandi operativi, nonché allo stato di basso livello del drone, e degli stessi messaggi di log di PX4.

Una rappresentazione completa di nodi e topic relativi a questo package è mostrata in Figura 3.18, mentre i listati seguenti espongono la struttura di ciascuno.

CAPITOLO 3. CASO DI STUDIO: DRONE AUTONOMO

```

class FlightController : public rclcpp::Node
{
public:
    FlightController();

private:
    rclcpp::Publisher<TrajectorySetpoint>::SharedPtr traj_setpoint_pub_;
    rclcpp::Publisher<OffboardControlMode>::SharedPtr offboard_cmode_pub_;
    rclcpp::CallbackGroup::SharedPtr setpoints_clbk_group_;
    rclcpp::TimerBase::SharedPtr setpoint_pub_timer_;

    void setpoint_timer_clbk(void);
};

```

Listing 3.3: Definizione del nodo *flight_controller*.

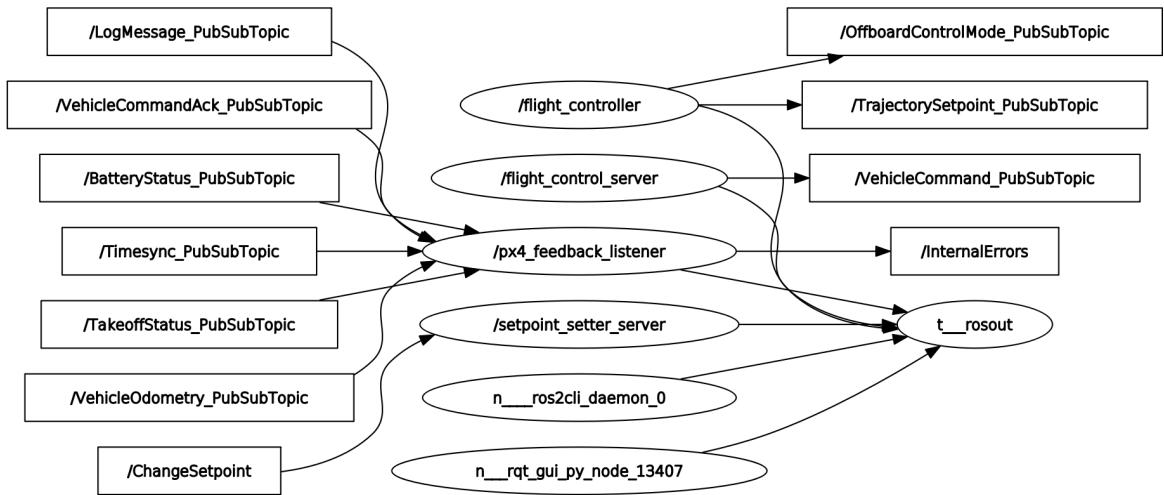


Figura 3.18: Schema riassuntivo di nodi e topic del package *Flight Control*.

```

class ControlServer : public rclcpp::Node
{
public:
    ControlServer();

    void arm(FlightControl::Response::SharedPtr resp) const;
    void disarm(FlightControl::Response::SharedPtr resp) const;
    void takeoff(FlightControl::Response::SharedPtr resp) const;
    void land(FlightControl::Response::SharedPtr resp) const;
    void manual_on(FlightControl::Response::SharedPtr resp) const;
    void manual_off(FlightControl::Response::SharedPtr resp) const;
    void setp_on(FlightControl::Response::SharedPtr resp) const;
    void setp_off(FlightControl::Response::SharedPtr resp) const;
    void reset(FlightControl::Response::SharedPtr resp) const;

private:
    rclcpp::Publisher<VehicleCommand>::SharedPtr vehicle_cmd_pub_;
    rclcpp::Service<FlightControl>::SharedPtr flight_control_srv_;
    rclcpp::CallbackGroup::SharedPtr flight_control_clbk_group_;

    uint16_t publish_vehicle_cmd(uint16_t cmd,
                                 float p1 = NAN,
                                 float p2 = NAN,
                                 float p3 = NAN,
                                 float p4 = NAN,
                                 float p5 = NAN,
                                 float p6 = NAN,
                                 float p7 = NAN) const;
    void flight_control_clbk(const FlightControl::Request::SharedPtr request,
                           const FlightControl::Response::SharedPtr response);
};


```

Listing 3.4: Definizione del nodo *control_server*.

```
class SetpointServer : public rclcpp::Node
{
public:
    SetpointServer();

    bool change_target_setpoint(uint8_t conf,
                               float x,
                               float y,
                               float z,
                               float vx,
                               float vy,
                               float vz,
                               float yaw);

private:
    rclcpp::Service<SetpointSetter>::SharedPtr setpoint_set_srv_;
    rclcpp::Subscription<ChangeSetpoint>::SharedPtr setpoint_set_sub_;
    rclcpp::CallbackGroup::SharedPtr setpoint_msg_clbk_group_;
    rclcpp::CallbackGroup::SharedPtr setpoint_srv_clbk_group_;

    void setpoint_set_srv_clbk(const SetpointSetter::Request::SharedPtr request,
                               const SetpointSetter::Response::SharedPtr response);
    void setpoint_set_msg_clbk(const ChangeSetpoint::SharedPtr msg);
};
```

Listing 3.5: Definizione del nodo *setpoint_server*.

CAPITOLO 3. CASO DI STUDIO: DRONE AUTONOMO

```
class PX4FeedbackListener : public rclcpp::Node
{
public:
    PX4FeedbackListener();

private:
    rclcpp::Subscription<VehicleCommandAck>::SharedPtr vehicle_cmd_ack_sub_;
    rclcpp::Subscription<TakeoffStatus>::SharedPtr takeoff_status_sub_;
    rclcpp::Subscription<BatteryStatus>::SharedPtr battery_status_sub_;
    rclcpp::Subscription<Timesync>::SharedPtr timesync_sub_;
    rclcpp::Subscription<LogMessage>::SharedPtr log_message_sub_;
    rclcpp::Subscription<VehicleOdometry>::SharedPtr odometry_sub_;

    rclcpp::CallbackGroup::SharedPtr vehicle_cmd_ack_clbk_group_;
    rclcpp::CallbackGroup::SharedPtr takeoff_status_clbk_group_;
    rclcpp::CallbackGroup::SharedPtr battery_status_clbk_group_;
    rclcpp::CallbackGroup::SharedPtr timesync_clbk_group_;
    rclcpp::CallbackGroup::SharedPtr log_message_clbk_group_;
    rclcpp::CallbackGroup::SharedPtr odometry_clbk_group_;

    rclcpp::Publisher<InternalError>::SharedPtr error_pub_;

    rclcpp::Clock clock_ = rclcpp::Clock(RCL_STEADY_TIME);
    rclcpp::Time low_bat_timer_ = rclcpp::Time(0, 0, RCL_STEADY_TIME);

    void vehicle_cmd_ack_msg_clbk(const VehicleCommandAck::SharedPtr msg);
    void takeoff_status_msg_clbk(const TakeoffStatus::SharedPtr msg);
    void battery_status_msg_clbk(const BatteryStatus::SharedPtr msg);
    void timesync_msg_clbk(const Timesync::SharedPtr msg);
    void log_message_clbk(const LogMessage::SharedPtr msg);
    void odometry_msg_clbk(const VehicleOdometry::SharedPtr msg);
};



---


```

Listing 3.6: Definizione del nodo *px4_feedback_listener*.

PX4 offre diverse modalità di volo, sia automatico che manuale; ai fini di questo lavoro quelle d'interesse sono le seguenti:

- **MANUAL:** In questa modalità il drone è totalmente sotto il controllo del pilota, che può operare mediante un normale radiocomando la cui ricevente deve essere collegata al Pixhawk.
- **OFFBOARD:** È la modalità di volo automatico standard, durante la quale si presuppone che il controllore di volo debba pilotare il drone in modo da eseguire dei comandi impartiti da un companion PC montato a bordo. Tali comandi sono dei setpoint di posizione nello spazio e angolo di yaw, oppure di velocità lineare e di imbardata e angolo di yaw; affinché il controllore di volo ritenga stabile la comunicazione col companion PC e non entri in failsafe, i setpoint devono essere inviati in continuazione ad un rate il più possibile stabile e superiore a 2 Hz. L'inserimento di questa modalità deve avvenire successivamente all'armamento e porta il drone a decollare verso il setpoint corrente.
- **LAND:** Non appena viene inserita questa modalità, presupponendo che il drone sia in volo, viene iniziata una discesa verticale stabilizzata verso il suolo. Un modulo interno a PX4 noto come *Land Detector*, e che consiste essenzialmente in una macchina a stati, si occupa di regolare la spinta e integrare le misure dell'odometria per capire quando il drone ha toccato terra e si è fermato, ossia può considerarsi atterrato.

Il Flight Control è stato pensato per essere una sorta di radiocomando a disposizione dei moduli di più alto livello, che consentisse di sfruttare le funzionalità sopracitate assicurando al contempo la possibilità di inserire prontamente il controllo manuale in caso di problemi. Per garantire la massima efficienza il package è stato programmato in modo concorrente, ripartendo le diverse callbacks su più thread e minimizzando l'uso di lock.

Alla base di tutto vi è il nodo *flight_controller*, la cui struttura è esposta nel Listing 3.3. Il suo compito è semplicemente quello di postare un setpoint verso PX4 sul topic */TrajectorySetpoint*, e il tipo di setpoint tra quelli prima descritti sul topic */OffboardControlMode*, mediante opportuni publishers. Tale operazione è invocata

da un timer, attraverso il quale si può impostare la frequenza di pubblicazione dei setpoint che è stata fissata a 20 Hz.

Il setpoint corrente può essere cambiato inviando una opportuna richiesta al nodo *setpoint_server*, descritto nel Listing 3.5. Tale nodo offre due modi per farlo: velocemente ma senza feedback sulla riuscita dell'operazione, inviando un messaggio contenente il nuovo setpoint sul topic */ChangeSetpoint*, oppure più lentamente ma con feedback formulando una richiesta al servizio */SetpointSetter*. In entrambi i casi viene invocata la routine *change_target_setpoint*, la quale controlla la correttezza del setpoint richiesto e solo in caso positivo lo inserisce nel sistema. Il servizio */SetpointSetter* offre anche la possibilità di aggiornare il setpoint corrente con uno di posizione ed attendere che questo sia effettivamente raggiunto, eventualmente anche con l'assetto stabilizzato. Per ottenere questo effetto la callback del servizio controlla l'odometria postata da PX4, e ritorna una risposta solo quando il drone si trova entro una sfera attorno al setpoint, il cui raggio è pure specificabile nella richiesta, ha l'angolo di yaw prossimo a quello richiesto ed eventualmente quelli di roll e pitch sufficientemente limitati. Naturalmente tutte queste soglie, come un'infinità di altri parametri da cui dipende il funzionamento di questo package, sono stati definiti nel suo header file e tarati opportunamente. Sempre per massimizzare l'efficienza e consentire il raggiungimento di rate di pubblicazione anche molto elevati, i dati globali condivisi tra gli ultimi due nodi in cui sono codificati i setpoint da postare sono gestiti secondo uno schema lock-free, in cui la concorrenza è risolta mediante i tipi atomici di C++ e un uso opportuno dei *memory orders* per indicare ai sottosistemi di CPU, cache e controller delle memorie quando e come eseguire le singole operazioni di lettura e scrittura ad essi relative. Tale meccanismo si è dimostrato molto efficiente ed è stato pertanto applicato più volte in situazioni simili presentatesi nel resto del software. Andando già verso un livello leggermente più alto si incontra il nodo *control_server*, illustrato nel Listing 3.4. Il suo compito è offrire al resto dell'architettura un modo semplice di cambiare lo stato del drone, eseguendo le operazioni di base quali armamento, disarmamento, decollo e atterraggio. L'interfaccia con l'esterno è rappresentata dal servizio */FlightControl*, mediante cui ciascuna di tali operazioni può essere eseguita formulando una opportuna richiesta; in caso d'errore, la risposta riporterà lo stesso

codice d'errore ritornato da PX4. Come si evince da listato, ciascuna operazione è codificata in una specifica routine invocata selettivamente dalla callback del servizio, che si risolve sempre nella pubblicazione di un messaggio verso PX4 sul topic `/VehicleCommand` costruito nella routine `publish_vehicle_command`. Gli ACK dei vari comandi inviati, nonché lo stato dell'esecuzione delle operazioni richieste, vengono opportunamente tracciati mediante altri topic su cui pubblica PX4 ed utilizzati per capire se l'operazione ha avuto successo o meno e si può restituire una risposta. Le logiche d'invio dei comandi prevedono anche tentativi di trasmissione multipli e timeouts per far fronte ad eventuali problemi del link seriale o del microRTPS Bridge. Vi è poi tutta una serie di routine destinate ad essere invocate tramite richieste formulate da un operatore in fase di testing, quali ad esempio la configurazione del Flight Control per la selezione delle modalità di volo tramite radiocomando, l'attivazione dello stream di setpoint pubblicati verso PX4, o il reset dello stato interno del modulo.

La realizzazione di quest'ultimo nodo si è resa necessaria fin da subito, in quanto anche il semplice inserimento di una modalità richiede la costruzione di un messaggio per PX4 dal formato poco intuitivo, e per controllare l'esito dell'operazione c'è bisogno di analizzare i messaggi in transito su altri topic nonché tenere traccia di quale sia inizialmente lo stato del drone stesso. Per questi motivi, e per come è codificato, il Flight Control si configura come una piccola macchina a stati finiti di limitata complessità, ma che appare all'esterno come un unico servizio con cui richiedere al controllore di volo operazioni di base formulando semplicissime richieste ROS. Costruire un modulo del genere, che rendesse semplice pilotare automaticamente il drone, è stato il primo problema ad essere affrontato e poi risolto grazie a ROS 2, ponendo le basi per il resto dell'architettura.

Tutti i feedback ricevuti da PX4, assieme alle misure in tempo reale dell'odometria da esso pubblicate, sono raccolti dal nodo `px4_feedback_listener`, illustrato nel Listing 3.6. Il nodo dispone di diversi subscribers relativi ai seguenti topic:

- **/VehicleCommandAck:** Ogni volta che un comando operativo viene postato a PX4, l'esito dell'operazione viene scritto in uno di questi messaggi.
- **/TakeoffStatus:** Le macchine a stati interne di PX4 pubblicano messaggi su questo topic ogni volta che il drone viene armato, fatto decollare, portato a

terra o disarmato; questi messaggi assieme a quelli del punto precedente vengono dunque usati per capire se un decollo o un atterraggio sono andati a buon fine, attraverso un meccanismo di segnalazione tra thread basato su condition variables che coinvolge le callbacks di questi subscribers e quella del servizio /FlightControl.

- **/BatteryStatus:** Da questi messaggi viene letta la tensione di batteria corrente, e se si rileva che essa si è mantenuta per un certo intervallo di tempo al di sotto di una soglia critica viene allertata la logica di livello superiore mediante un messaggio sul topic */InternalErrors*.
- **/Timesync:** PX4 posta in tempo reale il valore in microsecondi del suo orologio interno, il cui zero corrisponde all'istante di accensione; questo valore è condiviso tra tutti i nodi del package in quanto va inserito in tutti i messaggi da postare verso PX4.
- **/LogMessage:** PX4 integra un sistema di logging multilivello simile a quello del kernel Linux; di norma, i messaggi vengono scritti assieme agli altri dati in un log file memorizzato nel Pixhawk e che può essere scaricato al termine del volo, ma sono anche pubblicati su questo topic e dunque la relativa callback li stamperà a schermo in modo che l'operatore possa tenere traccia in tempo reale anche dell'operato e dello stato del controllore di volo.
- **/VehicleOdometry:** In questi messaggi, pubblicati solitamente ad una frequenza di 100 Hz, sono scritte le misure integrate dal filtro di Kalman EKF2 relative a posizione, velocità e assetto del drone sotto forma di un quaternione di orientamento; ai fini di questo package, vengono estratti solo la posizione e i tre angoli di Eulero.

I primi test condotti con il prototipo sono stati orientati alla verifica del funzionamento del sistema e dell'architettura fino a questo livello, dunque ad accertarsi che quanto sin qui illustrato funzionasse e che il drone si potesse far decollare, spostare ed atterrare in modo deterministico. Per una varietà di problemi tecnici sorti in varie fasi del progetto, conseguire questo obiettivo ha richiesto quattro dei sei mesi a disposizione, e

le ultime criticità sono state individuate e risolte solo durante il secondo giorno di gara. È però con immensa soddisfazione che si sottolinea come, nonostante le altre numerose limitazioni strutturali e costruttive, da quel momento in poi il prototipo abbia volato in modo totalmente affidabile e preciso, senza necessità di alcun intervento umano.

RealSense Node

Un altro package di fondamentale importanza per il funzionamento del sistema è quello che si occupa di configurare le due camere Intel RealSense D435i ed esporle al resto dell'architettura. Trattandosi di soluzioni naturalmente orientate all'ambito della robotica, Intel distribuisce apertamente un package ROS 2 che comprende dei driver Linux per le camere RealSense, il cui funzionamento non è sempre affidabile ma ai fini di questo progetto è stato giudicato sufficientemente buono. L'eseguibile generato a partire da tale package può essere configurato in fase di avvio, usando opportuni launch files inclusi, per creare un nodo per ogni camera collegata, configurandola opportunamente e attivando o disattivando i vari sensori. Nel caso in esame la camera frontale, denominata *vslam*, è stata configurata attivando il sensore a infrarossi e quello di profondità, mentre a quella inferiore denominata *aruco* è stato attivato solo il sensore RGB per ottenere delle immagini a colori di Roomba e piazzole. Il frame rate è stato impostato a 30 fps per entrambe le camere. Le immagini scattate e i dati degli altri sensori sono pubblicati dai nodi di questo package su dei topic opportunamente denominati, usando interfacce standard di ROS 2 e una policy di QoS best-effort. Lo schema riassuntivo della configurazione ottenuta con le due camere è mostrato in Figura 3.19.

ORB_SLAM2

Le regole del Contest prevedono che il drone non possa volare con l'ausilio di segnale GNSS, ma debba comunque localizzarsi nell'ambiente in tempo reale. Uno dei primissimi problemi da affrontare è stato proprio scegliere ed implementare sull'architettura costruita un algoritmo sufficientemente efficiente ed accurato, che potesse fornire a

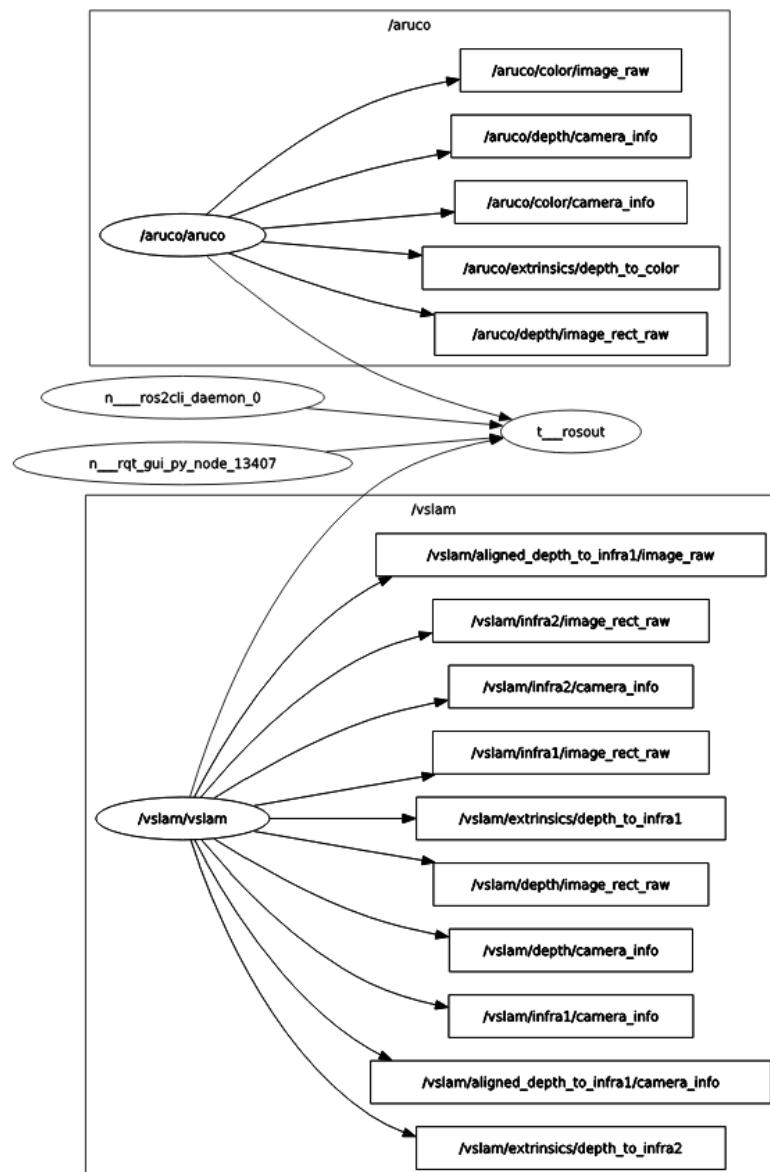


Figura 3.19: Organizzazione di nodi e topic relativi alle due camere Intel RealSense.

PX4 una stima attendibile e regolare della posa¹⁴ locale. Il filtro di Kalman EKF2 può infatti essere configurato per volare senza GPS, escludendo completamente tale modulo ed affidandosi completamente ad altri metodi di localizzazione quali la visione artificiale, ma necessita di uno stream continuo di informazioni coerenti. Confrontando diverse soluzioni disponibili ed adeguatamente documentate, la scelta è ricaduta sul sistema di VSLAM¹⁵ ORB_SLAM2. Si tratta della seconda versione del sistema ORB_SLAM descritto in [11], le cui estensioni rispetto alla versione originale sono elencate in [12]. In sintesi, ORB_SLAM2 si basa sull'estrazione di features binarie da un frame, sulla base delle quali viene man mano costruita una mappa locale risolvendo problemi di *bundle adjustment*. Le stesse features sono usate ovunque nel sistema in modo da evitare conversioni e massimizzare l'efficienza. I punti principali di una mappa locale, che ne costituiscono la parte essenziale, appartengono a un ristretto sottoinsieme di frames denominati *keyframes*. I frame vengono incrociati per calcolare la posa locale corrente componendo un grafo chiamato *Grafo di Covisibilità*, usato sia per localizzare il sistema che per costruire in memoria una mappa dell'ambiente. Man mano che la mappa cresce viene costruito un grafo che ne riassume le caratteristiche salienti, noto come *Grafo Essenziale*, grazie al quale vengono individuati i loop; all'occorrenza del rilevamento di un loop viene effettuata una *chiusura*: tutti i grafi vengono riesaminati e aggiornati, i keyframes e i nodi ridondanti vengono eliminati, e la mappa locale viene corretta in base ai rilevamenti fatti in modo da ridurre al minimo gli errori e renderla il più possibile coerente con la realtà. Per i dettagli teorici dietro il funzionamento di questo sistema si rimanda sempre a [11]. Dal punto di vista implementativo, tutto può essere fatto in real-time dato che ciascuna delle tre fasi di tracking, local mapping e loop closing descritte poc'anzi è affidata ad uno specifico thread. La versione 2 aggiunge un quarto thread, attivato subito dopo la chiusura di un loop, il cui compito è ripercorrere l'intera mappa risolvendo un problema di BA globale. L'organizzazione di questi thread è riassunta in Figura 3.20. Altri miglioramenti apportati sono stati l'aggiunta del supporto alle camere stereo, come le RealSense impiegate in questo

¹⁴Per *posa* s'intende una combinazione di dati che esprimono la posizione nello spazio e l'orientamento di un corpo rigido.

¹⁵Tale sigla sta per "Visual Simultaneous Localization And Mapping", ed indica una classe di algoritmi basati sul riconoscimento di elementi all'interno di un'immagine al fine di mappare l'ambiente circostante un robot e localizzare quest'ultimo al suo interno in tempo reale.

progetto, e lo sviluppo di una modalità chiamata *localizzazione*, in cui il sistema carica all'avvio una mappa salvata su file e si limita a localizzarsi al suo interno, senza tentare di ottimizzarla o estenderla. In Figura 3.21, a titolo d'esempio, è rappresentata la costruzione dei grafi sopracitati ed un esempio di loop closure effettuate su uno dei dataset usati per testare la prima versione del sistema.

L'impiego di ORB_SLAB2 si è dimostrato inizialmente problematico. Nonostante l'implementazione fornita di base dagli autori funzioni bene, si rileva facilmente da un'ispezione del codice e degli ottimizzatori di terze parti cui si appoggia come sia fortemente legata all'architettura x86, soprattutto in alcuni punti relativi all'estrazione ed elaborazione delle features, eseguite con istruzioni vettoriali AVX/SSE. Il passaggio all'architettura ARM ha richiesto di disabilitare queste ottimizzazioni, dato che l'alternativa sarebbe stata riscrivere i punti corrispondenti usando le istruzioni vettoriali NEON, cosa impossibile visti i tempi a disposizione. Si è invece apportato un miglioramento riscrivendo alcune parti del sistema per far uso della GPU montata sulla Jetson NX, attraverso istruzioni per l'architettura CUDA e l'uso della toolchain basata sul compilatore NVCC, e sfruttando al massimo le ottimizzazioni della toolchain GCC in fase di compilazione della libreria. Nonostante tutto, le performance ottenute sono complessivamente solo accettabili, ed un ulteriore lavoro di porting sarà necessario per costruire un'implementazione per ARM che possa considerarsi efficiente al pari della controparte x86. Questo parte è stato curata a più riprese in collaborazione con l'Ing. Fabrizio Romanelli, cui si devono anche altri sostanziali miglioramenti alla codebase di ORB_SLAB2 relativi alla gestione della mappa, nonché una esaustiva spiegazione del funzionamento e delle modalità d'uso di questo sistema assieme alle camere RealSense. Dato che l'esito di un qualunque volo dipende dal funzionamento di questo sistema, sono state tentate varie strade per trovare il modo migliore di utilizzarlo. Inizialmente è stata intrapresa quella della localizzazione: si è acquisita una mappa del campo gara, contenuta in un file grande un centinaio di MB, e la si è ricaricata a runtime sul drone al fine di fargliela usare per localizzarsi. Questo approccio ha avuto fin da subito due grossi problemi: nonostante la precisione con cui la mappa poteva essere acquisita, ci sarebbero sempre stati dei punti in cui il drone avrebbe fatto fatica a localizzarsi poiché poveri di features, e in tale modalità il sistema non ne avrebbe cercate di nuove; inoltre,

dovendo processare in tempo reale una mappa globale di grandi dimensioni, la scarsa ottimizzazione per CPU ARM si faceva sentire sotto forma di rate di campionamento molto bassi, nell'ordine di 8-10 Hz, mentre si è sperimentato che PX4 richiede dati di posa a non meno di 16 Hz. A quest'ultima criticità si è tentato di far fronte implementando degli estrapolatori, dapprima quadratici e poi lineari, al fine di predire un campione in più da un buffer circolare di lunghezza fissa. Nonostante il drone fosse così in grado di volare abbastanza bene, la suscettibilità del sistema di localizzazione su mappa alle variazioni delle condizioni di luce hanno fatto optare per un ritorno al volo in SLAM puro. In questa modalità, tarando opportunamente il numero massimo di features da estrarre da un'immagine, è stato possibile ottenere rate di campionamento stabili e anche superiori ai 20 Hz; dopo aver riscontrato quale fosse il carico del resto del software, il numero massimo di features ottimale è stato individuato in 600 contro le 1000 di base, corrispondente ad un rate di circa 20 Hz a pieno carico. È stato necessario ridurre così tanto questo valore anche perché un rate di campionamento instabile corrisponde a dei frame spazialmente troppo distanti, dai quali si è riscontrato che il sistema estrae senza volerlo delle misure errate, con scostamenti anche di svariate decine di centimetri che non è poi più in grado di correggere. Infine si è osservato che i tetti degli ostacoli del campo erano costruiti con un materiale traslucido, che causava riflessi nelle immagini visti come elementi in movimento assieme al drone, il quale tendeva dunque a correggere spostamenti che di fatto non aveva eseguito; la soluzione per ridurre questo effetto è stata montare la camera frontale su un supporto inclinato di 26.6° di pitch verso il basso. Così facendo si è ottenuto un sistema molto performante e preciso, reso ancora più affidabile dall'aggiunta di un filtro sulle misure di x , y e z che ne attenuasse le variazioni e ritardasse le correzioni apportate dal drone in caso di campioni spuri.

Il sistema ORB_SLAM2 è usato attraverso il package ROS 2 *orbslam2*, scritto sempre in collaborazione con l'Ing. Romanelli e composto da due nodi in esecuzione su un singolo thread, denominati rispettivamente *image_grabber* ed *orbslam2_node*. I listati seguenti ne illustrano la struttura, mentre nelle Figure 3.22 e 3.23 è riportato un loro schema riassuntivo dal punto di vista del middleware.

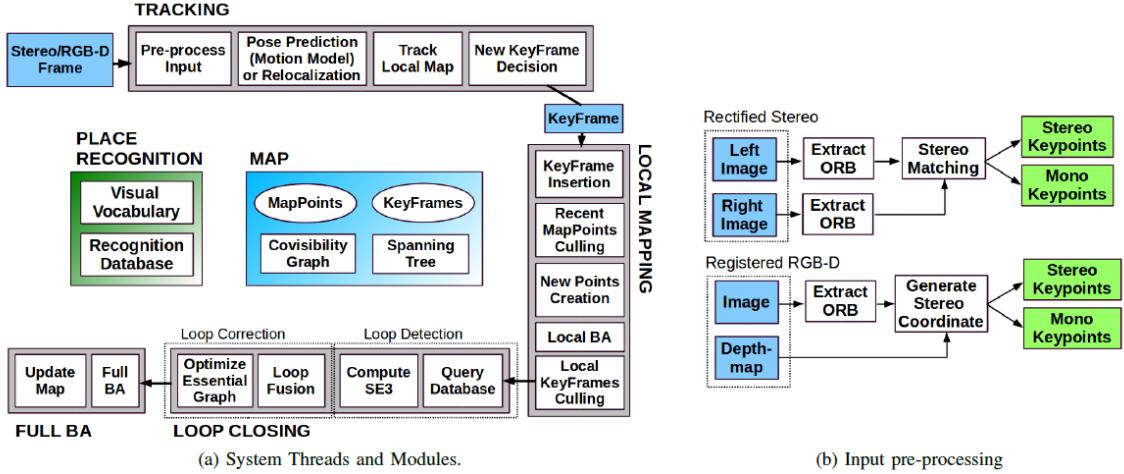


Figura 3.20: Organizzazione dei thread del sistema ORB_SLAM2.

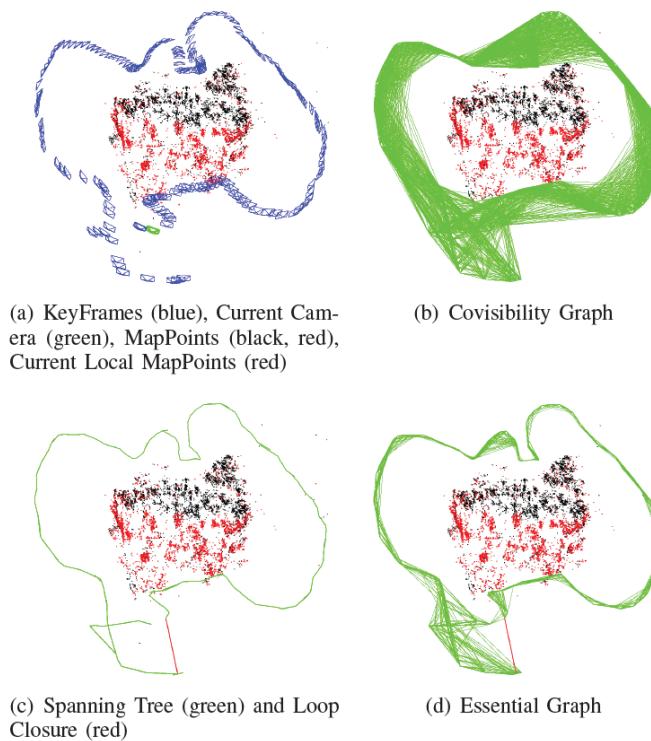


Figura 3.21: Costruzione dei grafi mediante frames e loop closure effettuate dal sistema ORB_SLAM su uno dei dataset di test.

CAPITOLO 3. CASO DI STUDIO: DRONE AUTONOMO

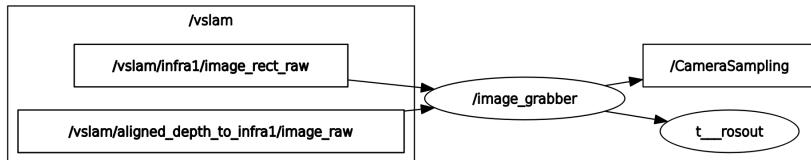


Figura 3.22: Organizzazione del nodo *image_grabber*.

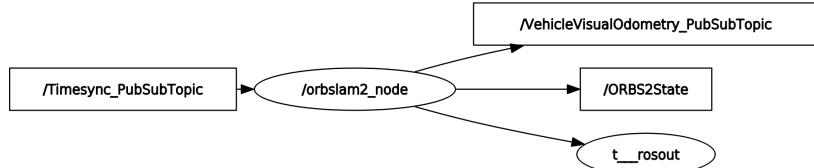


Figura 3.23: Organizzazione del nodo *orbslam2_node*.

```

class ORBSLAM2Node : public rclcpp::Node
{
public:
    ORBSLAM2Node(ORB_SLAM2::System *pSLAM,
                 ORB_SLAM2::System::eSensor _sensorType,
                 float camera_pitch,
                 int start_pad,
                 bool filter);

    void setPose(cv::Mat _pose);
    void setState(int32_t _state);

private:
    rclcpp::CallbackGroup::SharedPtr vio_clbk_group_;
    rclcpp::TimerBase::SharedPtr vio_timer_;
    rclcpp::Publisher<Int32>::SharedPtr state_publisher_;

    std::mutex poseMtx;
    std::mutex stateMtx;

    ORB_SLAM2::System *mpSLAM;
    ORB_SLAM2::System::eSensor sensorType;
    int32_t orbslam2State = ORB_SLAM2::Tracking::eTrackingState::SYSTEM_NOT_READY;

    cv::Mat orbslam2Pose = cv::Mat::eye(4, 4, CV_32F);

```

```
float camera_pitch_;
float cp_sin_, cp_cos_;
int start_pad_;
float x_offset_, y_offset_, z_offset_;
bool filter_;

const float a_0_ = 0.239018f;
const float a_1_ = -0.7379f;
const float b_0_ = 0.19080f;
const float b_1_ = 0.31028f;
const float tau_dynamic_thresh_[3] = {0.5f, 0.7f, 0.2f};
const float tau_thresh_min_[3] = {0.1f, 0.1f, 0.08f};
const float tau_dynamic_min_ = 0.1f;
const float tau_dynamic_slow_ = 0.98f;

float y_filtered_old_[3] = {0.0f, 0.0f, 0.0f};
float y_filtered_old_old_[3] = {0.0f, 0.0f, 0.0f};
float orb_data_old_[3] = {0.0f, 0.0f, 0.0f};
float orb_data_old_old_[3] = {0.0f, 0.0f, 0.0f};
float y_timevariant_old_[3] = {0.0f, 0.0f, 0.0f};
float tau_dynamic_[3] = {0.0f, 0.0f, 0.0f};
float orb_buffer_[3][8];

void timer_vio_callback(void);

void timestamp_callback(const Timesync::SharedPtr msg);

float orb_filter(float sample, int axis);

std::atomic<uint64_t> timestamp_;
rclcpp::CallbackGroup::SharedPtr timestamp_clbk_group_;
rclcpp::Publisher<VehicleVisualOdometry>::SharedPtr vio_publisher_;
rclcpp::Subscription<Timesync>::SharedPtr ts_sub_;
};
```

Listing 3.7: Definizione del nodo *orbslam2_node*.

```

typedef message_filters::sync_policies::ApproximateTime<Image, Image> sync_pol;

class ImageGrabber : public rclcpp::Node
{
public:
    ImageGrabber(ORB_SLAM2::System *pSLAM,
                 std::shared_ptr<ORBSLAM2Node> pORBSLAM2Node,
                 ORB_SLAM2::System::eSensor sensorType,
                 bool irDepth);

    void GrabRGBD(const Image::SharedPtr &msgRGB,
                  const Image::SharedPtr &msgD);
    void GrabStereo(const Image::SharedPtr &msgLeft,
                    const Image::SharedPtr &msgRight);

    ORB_SLAM2::System *mpSLAM;
    std::shared_ptr<ORBSLAM2Node> mpORBSLAM2Node;

private:
    message_filters::Subscriber<Image> stream1_sub_;
    message_filters::Subscriber<Image> stream2_sub_;
    std::shared_ptr<message_filters::Synchronizer<sync_pol>> sync_;

#ifdef BENCHMARK
    rclcpp::Publisher<Bool>::SharedPtr sampling_publisher_;
#endif
};

```

Listing 3.8: Definizione del nodo *image_grabber*.

Facendo riferimento al Listing 3.8, si nota come tutto abbia inizio entro il nodo *image_grabber*: esso si sottoscrive ai topic della camera *vslam* su cui vengono pubblicate le immagini a infrarossi e la mappa di profondità. Per farlo, usa una funzionalità di ROS nota come *message filter*: i due stream sono temporalmente sfasati tra loro, nonostante i due campioni pubblicati a poca distanza siano stati acquisiti nello stesso istante; il message filter applicato sui due topic bufferizza quello che arriva prima, ritardando la pubblicazione a quando anche l'altro è disponibile. Questa

implementazione inoltre riduce a 1 la dimensione della coda di messaggi provenienti dalle camere, per evitare di processare campioni vecchi. Successivamente i campioni vengono processati per generare una posa locale attraverso una chiamata al metodo *setPose* del nodo *orbslam2_node*. Al termine di queste operazioni, se la corrispondente feature è abilitata, viene postato un messaggio sul topic */CameraSampling*, del quale si può misurare il rate di pubblicazione al fine di stimare le performance del sistema ORB_SLAM2 in tempo reale.

Il nodo *orbslam2_node* è composto praticamente solo da un timer, alle cui occorrenze viene invocata una callback che legge l'ultima posa calcolata dal sistema e compone un messaggio da postare verso PX4 sul topic */VehicleVisualOdometry*. Nel far questo, la posa viene ruotata per compensare l'inclinazione della camera e trasformata dal sistema di riferimento di ORB_SLAM2 a quello "mondo" di PX4, che è NED¹⁶. Viene inoltre applicata una traslazione delle coordinate basata sulla piazzola di partenza, in quanto ORB_SLAM2 pone lo zero nel punto di accensione mentre il resto del software, come si vedrà, presuppone che l'origine delle coordinate sia in un punto fisso del campo. L'uso del timer è sostanzialmente un refuso delle precedenti versioni basate su localizzazione, dove era utile per decidere quando far intervenire gli estrapolatori, mentre ora consente soltanto di fissare il rate di pubblicazione. Il messaggio postato verso PX4 consta essenzialmente di coordinate NED x , y , z , un quaternione d'orientamento ed un timestamp. Opzionalmente, alle tre coordinate spaziali può essere applicato il filtro numerico menzionato precedentemente, il cui funzionamento può essere riassunto così:

$$\xi^+ = \tau\xi + (1 - \tau)y \quad (3.1a)$$

$$\tau^+ = \begin{cases} \max(\tau_{\min}, \tau - \frac{1}{120}) & , \text{ if } T_r < \sigma_T \\ \tau_{\max} & , \text{ otherwise} \end{cases} \quad (3.1b)$$

dove y è l'ultimo campione acquisito dal sistema di localizzazione sullo specifico asse, ξ è il corrispondente campione generato dal filtro, e τ è un coefficiente calcolato sempre per ogni asse sulla base di un buffer circolare di campioni. L'idea è che tale buffer riassume la storia recente del sistema e consenta di individuare campioni spuri, che

¹⁶North-East-Down, ossia x in avanti, y verso destra e z verso il basso.

verranno filtrati di più o di meno a seconda del valore di τ , a sua volta aggiornato sulla base di T_r che è calcolato controllando la differenza tra i campioni nel buffer. Tutti i parametri e le saturazioni τ_{\min} , τ_{\max} e σ_T sono stati tarati sulla base di dati raccolti durante i voli. L'effetto di questa soluzione è ridurre l'impatto degli outliers prodotti da ORB_SLAM2 sul comportamento del drone, ritardandone leggermente gli effetti e dando il tempo al sistema di produrre nuovi campioni più corretti evitando contemporaneamente che il drone corregga falsi spostamenti.

Aruco Scanner

Le immagini RGB provenienti dalla camera inferiore sono necessarie per individuare i robot sconosciuti, acquisire la sequenza dei punteggi e localizzare le piazzole di atterraggio. Il processamento di queste immagini è affidato al nodo *aruco_scanner*, la cui struttura è descritta nel Listing 3.9 e illustrata in Figura 3.24. Il suo eseguibile va avviato fornendo gli ID dei robot conosciuti e la distanza focale, in pixel, della camera. Tutto il lavoro compiuto da questo nodo è codificato nella callback *down_camera_clbk*, che acquisisce con un'apposita policy di QoS l'ultima immagine scattata e la elabora al fine di trovare eventuali landmark ArUco presenti in essa. Per questo task è stata impiegata la libreria OpenCV, compilata per architettura CUDA in modo da far uso anche in questo contesto della GPU. Se nell'immagine è presente un landmark, la callback ha due modalità di funzionamento:

- se alla ricerca di un robot, ed il landmark rilevato non è tra quelli noti, viene salvata in locale la foto scattata e inviato un messaggio d'allerta alla logica di navigazione sul topic */NavigatorAlert*;
- se in navigazione verso una piazzola d'atterraggio o durante un precision landing, viene calcolato un vettore d'errore di posizionamento rispetto al centro della piazzola nel sistema di riferimento NED solidale al drone, le cui componenti vengono poi poste assieme all'ID della piazzola sul topic */ArucoErrors* ed eventualmente su */LandError* per finalità di testing.

L'eseguibile comprende anche una modalità di calibrazione per poter determinare la distanza focale in pixel della camera in uso.

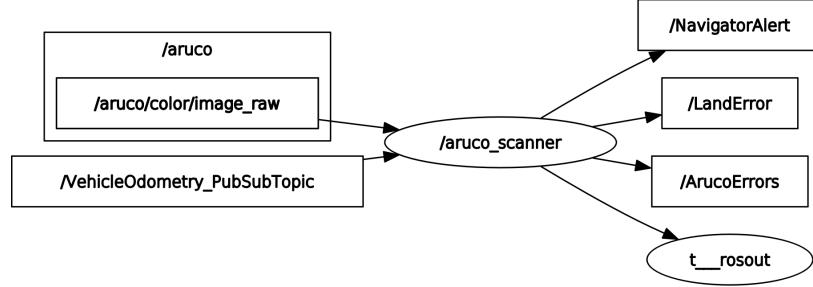


Figura 3.24: Organizzazione del nodo *aruco_scanner*.

Durante la navigazione, il calcolo dell'errore di posizionamento rispetto al centro della piazzola viene effettuato in metri. Il modello matematico applicato per determinare ciascuna componente del vettore errore, estremamente semplice e basato sostanzialmente su similitudini tra triangoli, è noto come *pinhole camera model* e può essere formulato come segue:

$$W = \frac{P \cdot D}{F} \quad (3.2)$$

dove P è la lunghezza misurata in pixel dall'immagine, D è la distanza dal landmark misurata usando l'altitudine del drone e compensando una eventuale elevazione della piazzola, F è la distanza focale della camera in pixel e W è la lunghezza effettiva in metri. Chiaramente questo approccio non è esente da errore: è stato rilevato che le misure sono attendibili fino al centimetro, una precisione sufficiente ma che richiede di troncare il risultato oppure azzerarlo quando troppo basso per escludere il rumore, ed inoltre essendo la camera inferiore solidale al drone queste misure hanno senso solo quando esso è livellato, problema a cui come si vedrà si è fatto fronte all'atto dell'acquisizione dei campioni negli altri moduli.

CAPITOLO 3. CASO DI STUDIO: DRONE AUTONOMO

```
class ArucoScanner : public rclcpp::Node
{
public:
    ArucoScanner(bool calibrate,
                 float calib_hgt,
                 float focal_len,
                 std::vector<int> ids_good);

private:
    rclcpp::Publisher<ArucoDataVector>::SharedPtr error_vector_pub_;
    rclcpp::Publisher<Float32>::SharedPtr focal_len_pub_;
    rclcpp::Publisher<Bool>::SharedPtr stop_pub_;

    rclcpp::Subscription<VehicleOdometry>::SharedPtr odometry_sub_;
    rclcpp::Subscription<Image>::SharedPtr down_camera_sub_;

    rclcpp::CallbackGroup::SharedPtr camera_clbk_group_;
    rclcpp::CallbackGroup::SharedPtr odometry_clbk_group_;

    bool calibrating_;
    float calibration_hgt_;
    float focal_len_;
    std::atomic<float> drone_z_;
    std::vector<int> ids_good_;
    int pics_count_;

    cv::Ptr<cv::aruco::Dictionary> aruco_dictionary_;
    cv_bridge::CvImagePtr cv_ptr_;
    cv::Mat image_copy_;
    std::vector<int> ids_;
    std::vector<std::vector<cv::Point2f>> corners_;

    void odometry_msg_clbk(const VehicleOdometry::SharedPtr msg);
    void down_camera_clbk(const Image::SharedPtr msg);
};
```

Listing 3.9: Definizione del nodo *aruco_scanner*.

Navigator

Salendo ora al livello più alto nella gerarchia dei moduli, in Figura 3.25 è rappresentata la struttura del nodo incaricato di gestire la navigazione: il Navigator. Supponendo che i sistemi di localizzazione e controllo volo funzionino correttamente, le operazioni che deve realizzare sono:

- decollo da una qualunque piazzola;
- esplorazione dell’ambiente alla ricerca del robot sconosciuto;
- navigazione da un qualunque punto della mappa verso una specifica piazzola;
- loop di ricerca attorno alla presunta posizione di una piazzola.

La sua struttura è illustrata nel Listing 3.10. L’organizzazione è semplice, poiché si appoggia totalmente ai servizi offerti dagli altri package, offrendo un ulteriore livello di astrazione operativa alla logica di supervisione da cui è comandato tramite il servizio */Navigation*. L’odometria viene campionata dal topic */VehicleOdometry*, per fissare l’esatta posizione da cui decollare al fine di evitare drift in salita e per salvare la posizione della piazzola verso cui si sta andando durante la navigazione, se questa dovesse essere rilevata nel tragitto, anche grazie agli errori di posizionamento ricevuti dal topic */ArucoErrors*. Durante l’esplorazione è l’Aruco Scanner ad allertare il Navigator che un robot sconosciuto è stato individuato, tramite il topic */NavigatorAlerts*; in tal caso, il drone viene messo in hovering e viene formulata una richiesta al servizio della GCS per richiedere la sequenza di atterraggi all’operatore. Le operazioni di volo sono realizzate mediante i servizi */FlightControl* e */SetpointSetter* offerti dal Flight Control. I percorsi da seguire in ogni fase sono definiti per punti, ed ogni leg tra due punti intermedi viene ulteriormente discretizzata nella routine *reach_point*, la quale richiede poi di attendere il raggiungimento di ogni punto mediante l’apposito servizio del Flight Control. Per velocizzare l’andatura, il raggio di confidenza in cui si suppone che un setpoint sia stato raggiunto viene allargato e ristretto dinamicamente secondo necessità. I percorsi d’esplorazione sono predefiniti per ogni piazzola in base alla mappa, al fine di massimizzare la probabilità di trovare il robot sconosciuto, mentre quelli di navigazione vengono richiesti ad un altro nodo che si occupa di calcolarli.

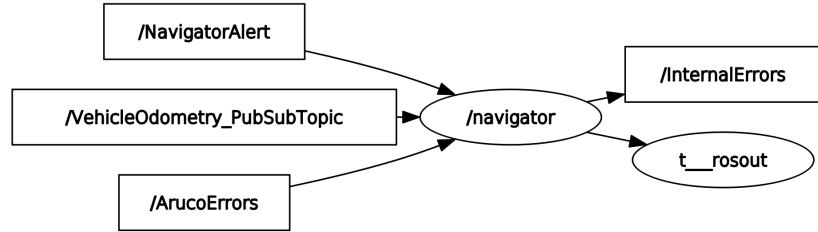


Figura 3.25: Organizzazione del nodo *navigator*.

```

class Navigator : public rclcpp::Node
{
public:
    Navigator();

private:
    rclcpp::Subscription<VehicleOdometry>::SharedPtr odometry_sub_;
    rclcpp::Subscription<Bool>::SharedPtr alert_sub_;
    rclcpp::Subscription<ArucoDataVector>::SharedPtr pad_sub_;

    rclcpp::Service<Navigation>::SharedPtr navigation_srv_;

    rclcpp::CallbackGroup::SharedPtr odometry_clbk_group_;
    rclcpp::CallbackGroup::SharedPtr alert_clbk_group_;
    rclcpp::CallbackGroup::SharedPtr navigation_clbk_group_;

    rclcpp::Client<FlightControl>::SharedPtr flight_control_client_;
    rclcpp::Client<LandingSequence>::SharedPtr landing_sequence_client_;
    rclcpp::Client<PathFinder>::SharedPtr path_finder_client_;
    rclcpp::Client<SetpointSetter>::SharedPtr setp_client_;
    rclcpp::Client<ArucoScanner>::SharedPtr scanner_client_;

    rclcpp::Publisher<InternalError>::SharedPtr error_pub_;

    std::mutex odom_lock_;
    float drone_x_, drone_y_, drone_z_;

    std::atomic<bool> stop_;

    std::atomic<int> curr_pad_;
  
```

```
float last_pad_pos_[2];

void odometry_clbk(const VehicleOdometry::SharedPtr msg);
void alert_clbk(const Bool::SharedPtr msg);
void pad_clbk(const ArucoDataVector::SharedPtr msg);
void navigation_clbk(const Navigation::Request::SharedPtr request,
                     const Navigation::Response::SharedPtr response);

void takeoff(const Navigation::Request::SharedPtr req,
             const Navigation::Response::SharedPtr resp);
void explore(const Navigation::Request::SharedPtr req,
             const Navigation::Response::SharedPtr resp);
void navigate(const Navigation::Request::SharedPtr req,
              const Navigation::Response::SharedPtr resp);
void search_pad(const Navigation::Request::SharedPtr req,
                const Navigation::Response::SharedPtr resp);

bool reach_point(float target_x, float target_y, float target_z,
                  bool exploring, bool check_yaw, bool force_disc, float step,
                  const Navigation::Response::SharedPtr resp);
};


```

Listing 3.10: Definizione del nodo *navigator*.

Path Finder

Non appena un robot sconosciuto è stato rilevato e l'operatore alla GCS ha trasmesso la sequenza di atterraggi da compiere, nonché dopo ogni decollo relativo a tale sequenza, si rende necessario navigare dal punto in cui ci si trova fino alla posizione della piazzola successiva. Il calcolo del percorso migliore dalla posizione corrente a quella desiderata è effettuato dal nodo *Path Finder*, la cui struttura è illustrata nel Listing 3.11. Tale nodo offre semplicemente il servizio */PathFinder* al Navigator, tramite il quale viene richiesto e restituito il percorso ottimo punto per punto, nel caso esista. L'algoritmo impiegato per il calcolo è A*, applicato su un grafo ricostruito dal pointcloud di una mappa accurata del campo acquisita mediante lo stesso sistema di localizzazione del drone, ossia ORB_SLAM2. Oggetto di questo lavoro è stata la realizzazione e

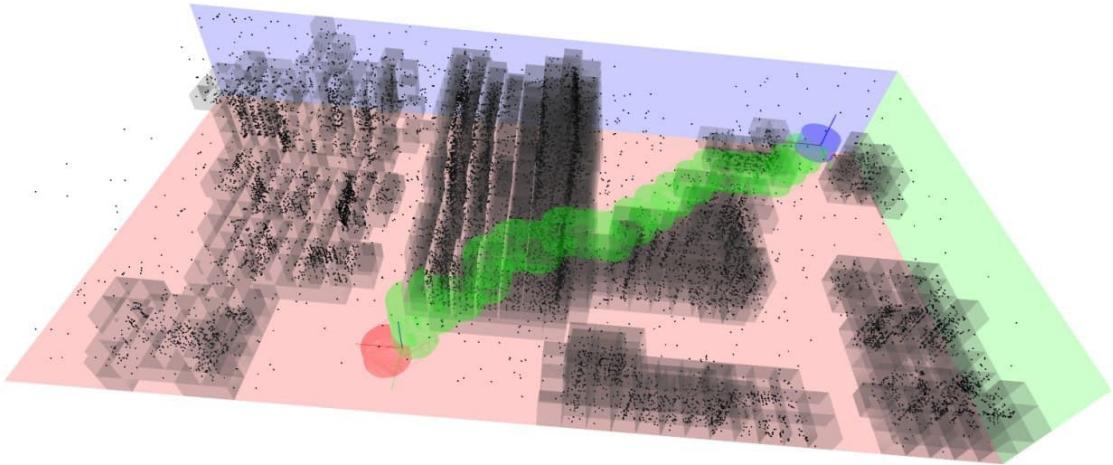


Figura 3.26: Risultato dell'algoritmo A* per il calcolo del percorso da piazzola 6 a 3.

integrazione di un nodo ROS 2 che offrisse tale servizio, mentre l'implementazione dell'algoritmo si deve al collega Alessandro Tenaglia, con un sentito ringraziamento personale. L'algoritmo tiene inoltre conto delle dimensioni del drone, con dei margini di sicurezza, assicurandosi di poterlo effettivamente spostare nei punti che trova. Un esempio d'esecuzione è mostrato in Figura 3.26, che evidenzia gli ostacoli e i punti del percorso.

```
class PathFinderNode : public rclcpp::Node
{
public:
    PathFinderNode(char *config_file, char *space_file);

private:
    float xStart, yStart, zStart, xMargin, yMargin, zMargin, yaw;
    CubeSpace *space;
    rclcpp::Service<PathFinder>::SharedPtr path_finder_srv_;
    void path_finder_clbk(PathFinder::Request::SharedPtr request,
                          PathFinder::Response::SharedPtr response);
};
```

Listing 3.11: Definizione del nodo *path_finder*.

Ground Control Station

In esecuzione sul PC adibito a Ground Control Station è stato posto un nodo, denominato appunto *GCS*, che consentisse all'operatore di inserire la sequenza degli atterraggi da eseguire dopo aver individuato il robot sconosciuto. La struttura di questo nodo è illustrata nel Listing 3.12 ed è estremamente semplice, in quanto si riduce ad un server per il servizio */LandingSequence* la cui callback offre all'operatore un'interfaccia a riga di comando per inserire la sequenza. È stata anche tentata la trasmissione, mediante un opportuno topic, delle foto dei robot sconosciuti tramite questo nodo, ma la procedura è stata rimossa in quanto come si è puntualizzato nel capitolo precedente la trasmissione di messaggi di grandi dimensioni come le immagini è al momento problematica per i DDS su reti soggette a perdita, come WiFi. Pertanto, tale funzionalità è stata rimpiazzata da uno script di shell che trasferisce via *scp* tutte le nuove foto scattate.

```
class GCSNode : public rclcpp::Node
{
public:
    GCSNode();

private:
    rclcpp::Service<LandingSequence>::SharedPtr landing_seq_srv_;
    rclcpp::CallbackGroup::SharedPtr land_seq_clbk_group_;

    void landing_seq_callback(const LandingSequence::Request::SharedPtr request,
                             const LandingSequence::Response::SharedPtr response);
};
```

Listing 3.12: Definizione del nodo *gcs*.

Precision Landing

Una volta raggiunta la presunta posizione di una piazzola, ed accertato tramite la camera inferiore e l'Aruco Scanner che essa è in vista, ci si deve atterrare. Questa

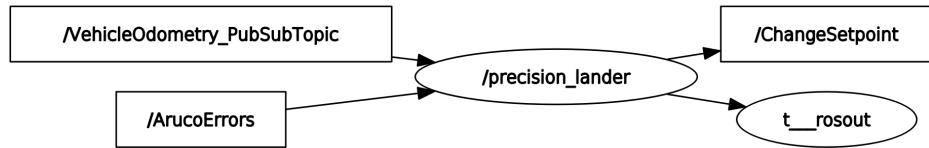


Figura 3.27: Struttura del nodo *precision_lander*.

operazione è a carico di un nodo, denominato *precision_lander* e facente parte del package *Precision Landing*, la cui struttura è mostrata nel Listing 3.13 e in Figura 3.27.

```
class PrecisionLander : public rclcpp::Node
{
public:
    PrecisionLander();

private:
    rclcpp::Publisher<afg_interfaces::msg::ChangeSetpoint>::SharedPtr vel_setp_pub_;
    rclcpp::Client<FlightControl>::SharedPtr flight_control_client_;
    rclcpp::Client<SetpointSetter>::SharedPtr setp_set_client_;

    rclcpp::Subscription<ArucoDataVector>::SharedPtr aruco_data_sub_;
    rclcpp::Subscription<VehicleOdometry>::SharedPtr vehicle_odometry_sub_;
    rclcpp::Service<PrecLanding>::SharedPtr prec_landing_srv_;

    rclcpp::CallbackGroup::SharedPtr aruco_data_clbk_group_;
    rclcpp::CallbackGroup::SharedPtr odometry_clbk_group_;
    rclcpp::CallbackGroup::SharedPtr prec_landing_clbk_group_;

    double drone_x_, drone_y_, drone_z_;
    double drone_vx_, drone_vy_, drone_vz_;
    double drone_roll_, drone_pitch_, drone_yaw_;
    std::mutex odometry_status_lock_;

    std::atomic<int> landing_id_;

    bool do_prec_landing_;
    std::mutex prec_landing_lock_;
    std::condition_variable prec_landing_cv_;
```

```
    std::atomic<bool> timeo_expired_;
```

```
    clock_t start_;
```

```
    clock_t t_now_, t_prec_;
```

```
    double t_passed_;
```



```
    float x_err_int_, y_err_int_;
```

```
    float x_err_der_, y_err_der_;
```

```
    float x_err_old_, y_err_old_;
```

```
    float z_ref_, z_init_;
```

```
    float land_alt_;
```

```
    float land_yaw_;
```

```
    float last_x_, last_y_;
```



```
    bool is_on_target(float x_err, float y_err);
```



```
    void aruco_data_clbk(const ArucoDataVector::SharedPtr msg);
```

```
    void odometry_states_clbk(const VehicleOdometry::SharedPtr msg);
```

```
    void precision_landing_clbk(const PrecLanding::Request::SharedPtr request,
```

```
                                const PrecLanding::Response::SharedPtr response);
```

```
};
```

Listing 3.13: Definizione del nodo *precision_lander*.

Come si evince dal listato, questo nodo è programmato in modo concorrente: vi è una callback che campiona l'odométria per capire come è posizionato ed orientato il drone totalmente in parallelo rispetto al resto, una callback dal servizio */PrecisionLanding* tramite cui può essere richiesta la procedura e una dal topic */ArucoErrors* da cui viene estratto l'errore di posizionamento rispetto alla piazzola. Quando giunge una richiesta per il servizio, il thread che la gestisce attiva il processamento dei dati nella callback da */ArucoErrors*, dove è implementato l'algoritmo di controllo. Tutte le operazioni di volo sono ancora una volta gestite mediante i servizi ed i topic offerti da Flight Control, ed il termine della procedura è soggetto a timeout e gestito mediante opportuni meccanismi di segnalazione tra thread basati su condition variables. Se tutto procede senza intoppi, la callback da */ArucoErrors* corregge progressivamente l'allineamento del drone facendolo scendere ed infine segnala all'altro thread, rimasto

in attesa, che questo è pronto per la discesa finale. Sarà la callback del servizio, una volta riattivata, a richiedere l'esecuzione dell'atterraggio a Flight Control.

L'algoritmo di controllo per correggere l'allineamento ed eseguire la discesa preliminare è stato sviluppato a più riprese: inizialmente si è testata in simulazione, con buoni risultati, una terna di PID¹⁷ sui vari assi che controllasse il drone in velocità lineare inviando messaggi sul topic */ChangeSetpoint*. Tale soluzione si è rivelata però, a pochi giorni dalla competizione, inapplicabile per problemi di carico del sistema che inficiavano le prestazioni della camera: il rate di campionamento era infatti troppo basso e troppo variabile per consentire l'implementazione di un qualunque algoritmo di controllo veloce. Per queste ragioni e per il pochissimo tempo a disposizione è stato implementato un algoritmo più semplice denominato *Land Corrector*: l'errore di posizionamento campionato dall'Aruco Scanner col drone livellato viene usato per calcolare un nuovo setpoint di posizione, che si raggiunge con stabilizzazione formulando una richiesta al servizio */SetpointSetter*; tale procedura viene ripetuta, diminuendo la quota se il drone è sufficientemente allineato e stabile, fino al raggiungimento di un'altitudine limite al di sotto della quale viene iniziata la discesa finale.

FSM

La collezione di moduli software termina con una macchina a stati finiti, che implementa la logica di supervisione posta al livello più alto. Visto da qui il drone appare come un automa in grado di svolgere una missione completa nei termini stabiliti dal regolamento del Contest, gestendo per quanto possibile situazioni impreviste come guasti, esaurimento della batteria o perdite di tracking del sistema di localizzazione. Alla luce della discussione fatta fino a questo punto, dovrebbe risultare chiaro come tutte le fasi della missione possano essere realizzate mediante delle richieste formulate ad opportuni servizi, alle quali sarà prodotta una risposta solo quando tale fase sarà stata completata, con successo o meno. Tali richieste vengono inviate durante le transizioni tra i vari stati, che dovranno essere innescate sulla base dello stato corrente

¹⁷Il PID è una tipologia standard di controllori che si basano sulla composizione di un'azione di controllo finalizzata ad azzerare un segnale d'errore, e calcolata sulla base di esso, la sua derivata rispetto al tempo e la sua somma integrale entro un opportuno intervallo.

e potenzialmente di altri eventi accaduti nel frattempo. Durante la transizione verso lo stato iniziale, viene controllato lo stato di tutti gli altri sottosistemi attraverso delle API offerte dal middleware ROS 2: se risultano tutti pienamente operativi, viene iniziata la sequenza di missione. Per la codifica della macchina è stata impiegata la libreria *Meta State Machine*¹⁸ facente parte delle *boost C++ libraries*. Nonostante l'impostazione un po' macchinosa, questa libreria ha consentito di scrivere con facilità l'implementazione dell'automa, consentendo di definire:

- stati come strutture con metodi in cui codificare cosa fare in ingresso ed in uscita da ciascuno;
- eventi come strutture a cui aggiungere eventualmente dei dati da far processare ai metodi degli stati durante le transizioni;
- transizioni e macchine intere istanziando classi template, il cui codice però assomigliava ad una vera e propria tabella.

La descrizione della macchina e dei suoi stati ricalca di fatto la dinamica di una missione così come stabilito dal regolamento. L'intera missione è stata dunque codificata, a seguito dei vari livelli di astrazione illustrati, in un'unica routine nella quale vengono invocate le varie transizioni tra gli stati a seconda delle condizioni in cui si trova il drone. La gestione delle condizioni d'errore è stata possibile mediante il concetto di *meta-macchina* di MSM: le parti pregnanti di una missione avvengono in degli stati che costituiscono una FSM a sé stante, ma dalla quale si può sempre uscire verso gli stati di terminazione. Dopo una serie di prove, si è riscontrato che è molto rischioso richiedere un atterraggio d'emergenza automatico al controllore di volo quando qualcosa non va, soprattutto quando il sistema di localizzazione ha smesso di funzionare; per tale ragione si è deciso che in caso di problemi la FSM dovesse interrompere immediatamente l'esecuzione senza richiedere nessun'altra operazione, permettendo così al pilota di inserire il controllo manuale il più presto possibile. Una rappresentazione grafica delle due macchine di alto e basso livello è offerta nelle Figure 3.29 e 3.30.

Per interagire con gli altri moduli attraverso il middleware, è stato aggiunto al package un nodo denominato *fsm*, la cui struttura è illustrata nel Listing 3.14 e in Figura 3.28.

¹⁸https://www.boost.org/doc/libs/1_64_0/libs/msm/doc/HTML/index.html

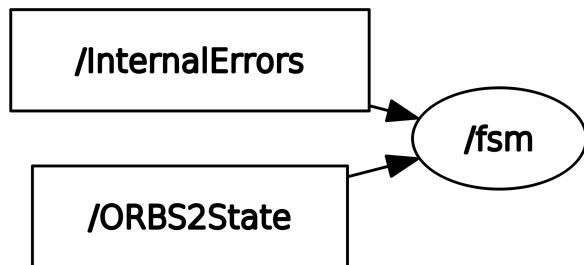


Figura 3.28: Struttura del nodo *fsm*.

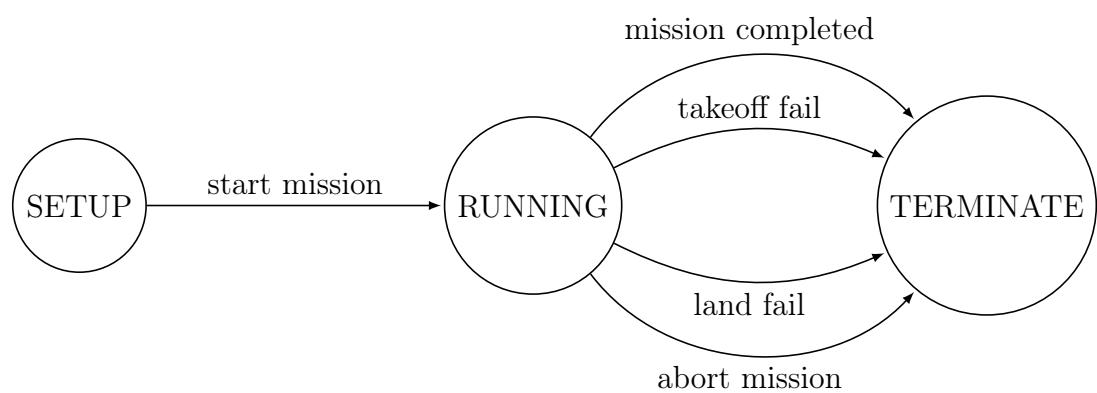


Figura 3.29: Diagramma di stato della FSM di alto livello.

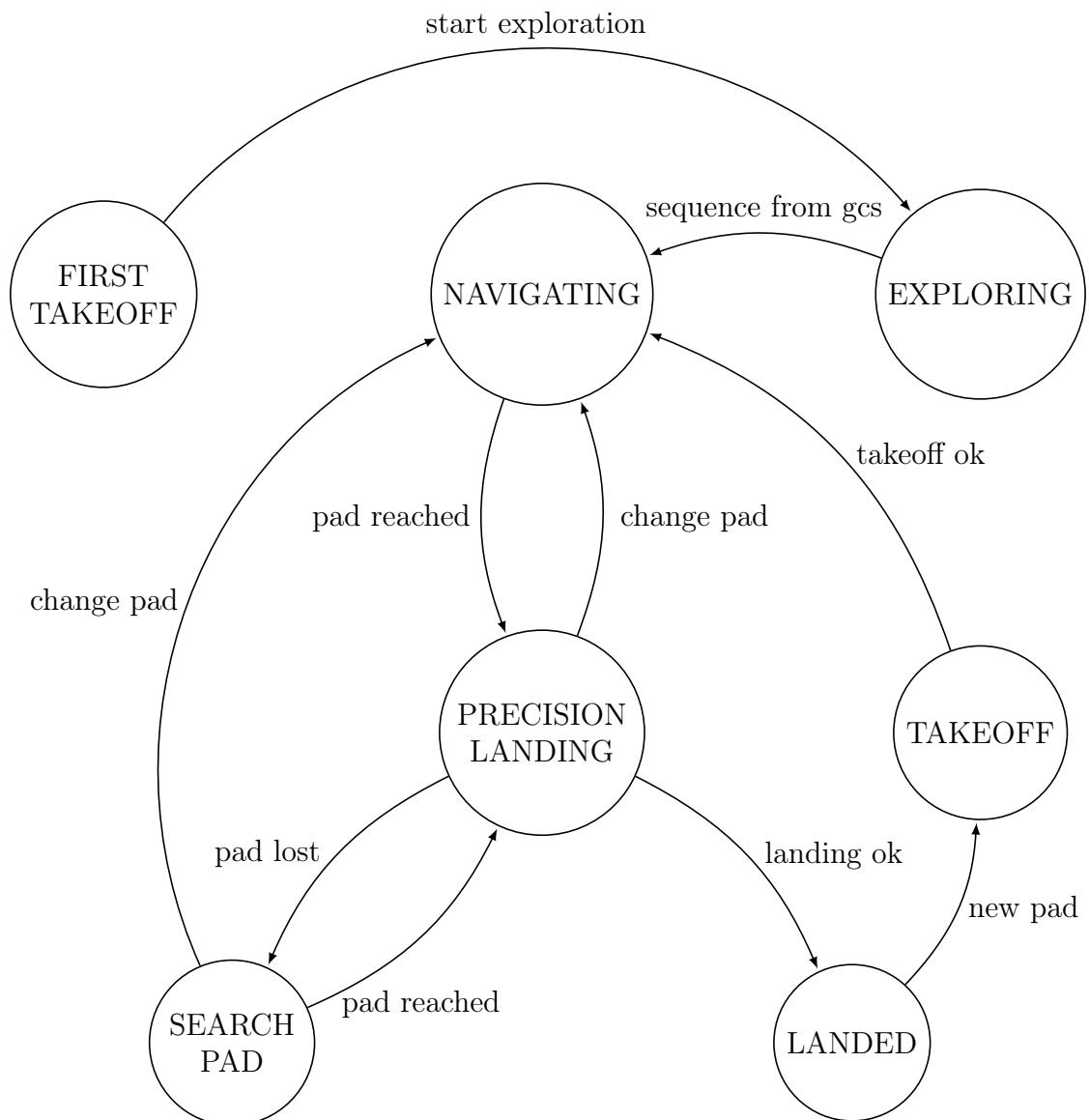


Figura 3.30: Diagramma di stato della meta-FSM contenuta nello stato *RUNNING*.

```
class FSMNode : public rclcpp::Node
{
public:
    FSMNode();

    rclcpp::Client<Navigation>::SharedPtr nav_client;
    rclcpp::Client<PrecLanding>::SharedPtr pl_client;
    rclcpp::Client<FlightControl>::SharedPtr fc_client;
    rclcpp::Client<SetpointSetter>::SharedPtr sp_client;
    rclcpp::Client<ArucoScanner>::SharedPtr aruco_client;
    rclcpp::Client<LandingSequence>::SharedPtr land_seq_client;
    rclcpp::Client<PathFinder>::SharedPtr path_finder_client;

private:
    rclcpp::Subscription<InternalError>::SharedPtr error_subscription_;
    rclcpp::Subscription<Int32>::SharedPtr orbslam2_state_sub_;
    void error_msg_callback(const InternalError::SharedPtr msg);
    void orbslam2_state_callback(const Int32::SharedPtr msg);
};


```

Listing 3.14: Definizione del nodo *fsm*.

Il nodo è in questo package semplicemente un punto d'accesso verso il middleware, che racchiude clients verso tutti i servizi finora descritti e subscribers ai topic *ORBS2State* per controllare lo stato del sistema di localizzazione ORB_SLAM2 ed */InternalErrors* per ricevere segnalazioni d'errore prodotte dagli altri moduli dell'architettura.

Realizzazione

Sono stati realizzati due prototipi strutturali del drone, ottimizzando la costruzione nel passaggio dal primo al secondo, rispettivamente nei mesi di Maggio e Giugno 2021; non essendo il primo di alcun aiuto ai fini della presente trattazione, esso è stato tralasciato. I diversi moduli di quest'architettura sono stati sviluppati singolarmente e testati prima in simulazione su Gazebo, poi sul prototipo reale nei locali della Facoltà d'Ingegneria dell'Università di Roma "Tor Vergata" e nel corso di due ricognizioni del campo gara

CAPITOLO 3. CASO DI STUDIO: DRONE AUTONOMO

presso la Divisione Velivoli di Leonardo S.p.A. a Torino, Italia, rispettivamente nei mesi di Giugno e Settembre 2021. L'operatore alla GCS entra in comunicazione con il sistema montato sul drone mediante 10 terminali SSH¹⁹ attivati grazie al terminal emulator *Terminator*. Da ciascuno di essi è possibile attivare ognuno dei moduli sin qui illustrati, secondo un ordinamento definito in una checklist. Una volta stabilite le modalità di volo e corretti gli ultimi problemi, il prototipo è stato portato in gara. Le prime due delle quattro manches sono state totalmente infruttuose, a causa di un cambiamento dell'ultimo minuto apportato da Leonardo alla costruzione del campo gara e di problemi di carico che si riflettevano sulla stabilità del sistema di localizzazione, questi ultimi peraltro presenti fin dall'inizio ma mai individuati con sicurezza fino a quel punto. Una volta isolate e risolte le cause, nelle ultime due manches tutti i sistemi hanno funzionato alla perfezione, ed il prototipo ha volato con precisione portando a termine entrambe le missioni per esaurimento della batteria, totalizzando sempre ottimi punteggi. La giuria ha apprezzato questo comportamento, nonché la completa assenza di problemi dal momento in cui il drone decollava fino al termine della gara. Gli atterraggi, eseguiti grazie al *Land Detector*, sono stati giudicati da giuria e pubblico i migliori tra quelli effettuati da tutti i team, per la loro precisione e semplicità d'esecuzione. Il team dell'Università di Roma "Tor Vergata", la cui composizione è stata più volte citata all'inizio di questo lavoro, si è aggiudicato pertanto il secondo posto superando quello del Politecnico di Torino e venendo preceduto per un solo atterraggio da quello del Politecnico di Milano.

¹⁹In informatica e telecomunicazioni SSH (Secure SHell) è un protocollo che permette di stabilire una sessione remota cifrata tramite interfaccia a riga di comando con un altro host di una rete informatica.

Capitolo 4

Controllori e middleware

Premessa

Nel Capitolo 3 è stata discussa l'implementazione di un'architettura di controllo basata sul middleware ROS 2. Su tale architettura sono state realizzate le logiche di controllo e supervisione di livello medio-alto del drone autonomo presentato come caso di studio. L'obiettivo di questo capitolo è presentare uno studio preliminare delle possibilità di tale architettura in contesti di più basso livello, relativi ai sistemi di controllo veloci. Verrà dunque proposto un controllore relativo allo stesso caso di studio precedente e che sarà realizzato tramite moduli ROS 2 ed infine testato usando il modello del drone simulato in Gazebo.

Problema di controllo

Il precision landing è una fase critica della missione: dal suo corretto svolgimento dipende l'assegnazione del punteggio relativo alla piazzola su cui lo si esegue. Qualunque disturbo presente in fase di allineamento e discesa, se non correttamente rilevato, può compromettere la stabilità del volo e dunque la precisione dell'atterraggio sul target. Il problema di controllo che verrà ora studiato è quello del precision landing in presenza di vento. Quest'ultimo è inteso come un disturbo costante di ampiezza e

segno non noti a priori, che va a sommarsi alla velocità assunta dal drone mediante il suo sistema di controllo veloce illustrato precedentemente. Si assume inoltre che, per effettuare la procedura con la massima precisione possibile, il drone sia controllato in velocità lineare mediante invio di setpoint appositi, calcolati da un opportuno controllore il cui ingresso sia un segnale d'errore di posizionamento rispetto al target, espresso in metri e campionato dalla camera inferiore a 20 Hz grazie ad uno dei moduli precedentemente descritti. Tale soluzione si configura dunque come un'alternativa rispetto a quella presentata nel Capitolo 3 e implementata nel modulo Land Corrector, che era basata su una correzione del posizionamento richiesta direttamente al controllo di posizione di PX4. Per semplicità verranno controllati separatamente i posizionamenti lungo gli assi x ed y del sistema di riferimento solidale al suolo, mentre la quota verrà progressivamente abbassata fintantoché l'allineamento sarà corretto secondo la stessa logica del modulo originale, e sempre mediante un controllo in velocità proporzionale alla differenza tra la quota corrente ed il nuovo riferimento. Tale semplice controllo sull'asse z si è dimostrato avere buone performance e pertanto è stato mantenuto.

Identificazione del sistema da controllare

Il problema descritto poc'anzi richiede un'analisi preliminare del sistema da porre sotto controllo: si tratta infatti di una situazione non-standard. Si vuole, in sintesi, controllare il drone, già sotto il controllo stabilizzante del Pixhawk, impostando un riferimento di velocità da fargli inseguire, mediante delle azioni di controllo realizzate dai motori che sono però calcolate da PX4 e supposte ignote. Nonostante l'ipotesi semplificativa di separare il controllo sui due assi, è necessario identificare due diversi sistemi, che esprimano il modo in cui il drone stabilizzato insegue riferimenti di velocità su ciascun asse. Tali sistemi saranno chiaramente nonlineari nella realtà, ma ai fini di questo lavoro è stata giudicata sufficiente una loro approssimazione con modelli lineari di ordine opportuno.

La fase di raccolta dati è stata eseguita su Gazebo, impiegando il modulo Flight Control per controllare il drone e definendo delle routine di test per ciascun asse che consistevano nell'invio di diversi setpoint di velocità da inseguire, implementate come

script di shell. Durante i voli di prova sono stati registrati tramite bagging i dati di input/output su cui lavorare, nella forma di setpoint correnti e velocità misurate da EKF2 ad ogni istante: i primi inviati dal nodo `flight_controller` a 20 Hz sul topic `/TrajectorySetpoint` e le seconde trasmesse da PX4 sul topic `/VehicleOdometry` a 100 Hz. Successivamente un nodo ROS 2 scritto ad-hoc ha consentito di ricampionare entrambi i segnali allo stesso rate dell'odometria, dunque 100 Hz, salvandoli infine tutti in un file CSV. I datasets così ottenuti sono mostrati nelle Figure 4.1 e 4.2. Da essi si evince chiaramente come si tratti, in entrambi i casi, di sistemi approssimabili linearmente al primo ordine.

La procedura d'identificazione è stata eseguita con l'ausilio della System Identification Toolbox¹ di MATLAB. Il Toolbox è stato impiegato in entrambi i casi per identificare la rappresentazione nello spazio di stato di un sistema lineare del primo ordine usando la *Prediction Error Minimization*, da cui calcolare poi una funzione di trasferimento. I dataset sono stati divisi in due parti, usate rispettivamente per la stima e la validazione. Un confronto tra i modelli approssimati e quelli reali eseguito sui validation sets, unitamente agli intervalli di confidenza, è disponibile nelle Figure 4.3 e 4.4, mentre le due funzioni di trasferimento ad essi corrispondenti sono:

$$W_x(s) = \frac{1}{s + 1.2775} \quad (4.1a)$$

$$W_y(s) = \frac{1}{s + 1.4350} \quad (4.1b)$$

i cui guadagni statici sono stati approssimati pari ad 1 in quanto il controllore di volo insegue i riferimenti di velocità in modo da annullare l'errore a regime.

Design del controllore

La prima soluzione ad essere stata pensata per il problema del precision landing consisteva in un controllore PID da tarare opportunamente mediante esperimenti. Un suo schema ideale è mostrato in Figura 4.5. Al di là dei problemi tecnici riscontrati con

¹<https://it.mathworks.com/products/sysid.html>

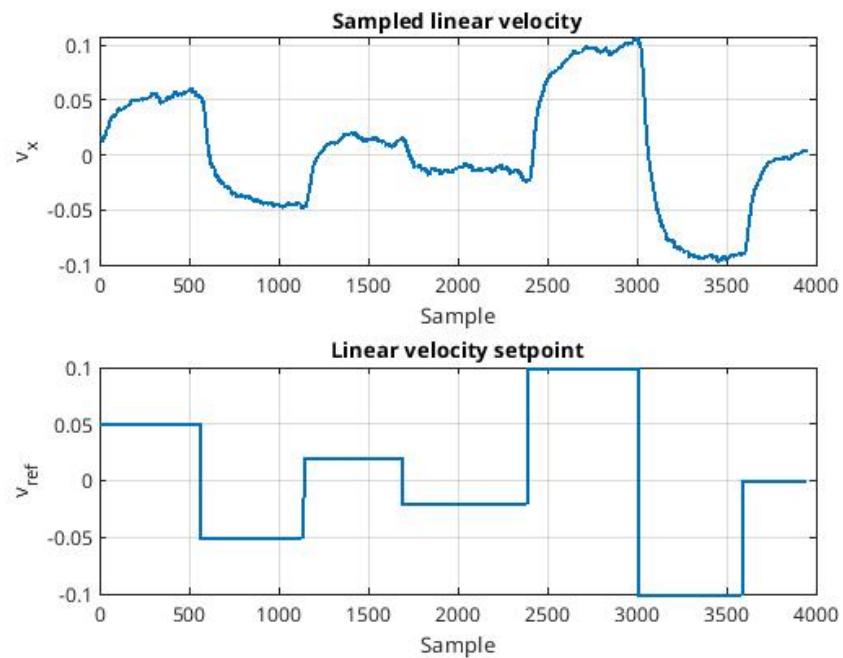


Figura 4.1: Dataset I/O relativo all'asse x .

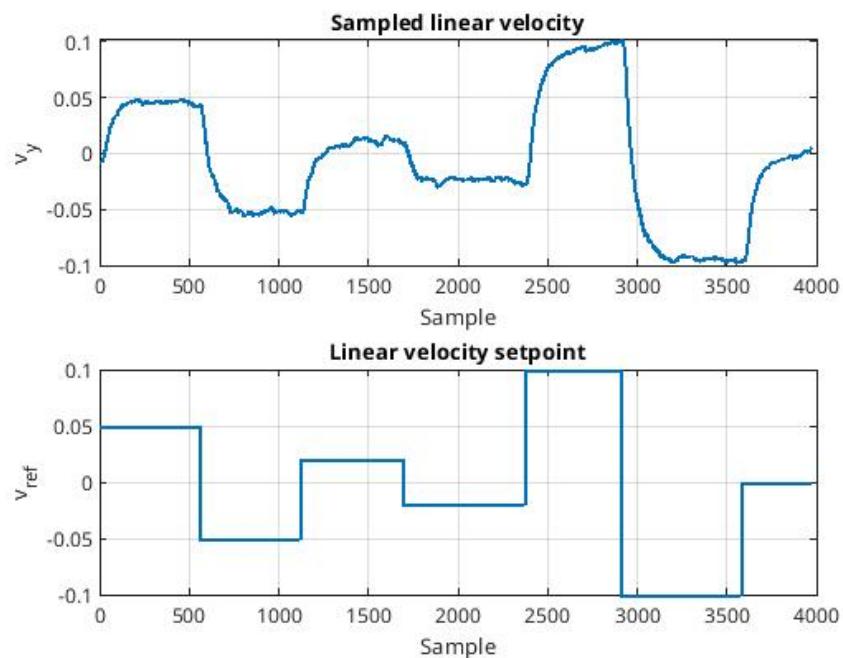


Figura 4.2: Dataset I/O relativo all'asse y .

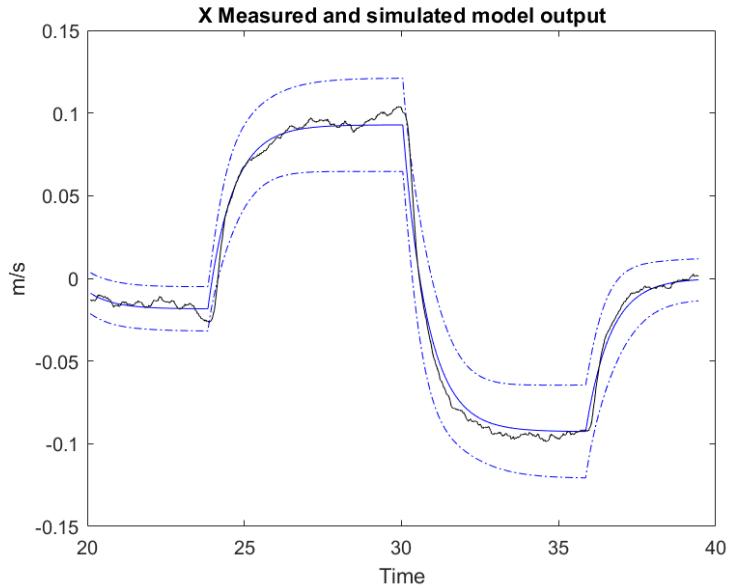


Figura 4.3: Confronto tra l'output misurato e quello approssimato (in blu) sul validation set dell'asse x . In tratteggio l'intervallo di confidenza.

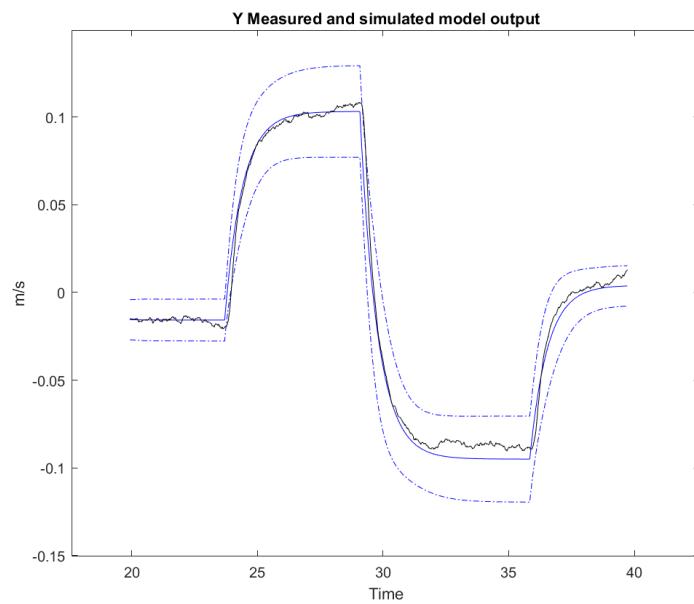


Figura 4.4: Confronto tra l'output misurato e quello approssimato (in blu) sul validation set dell'asse y . In tratteggio l'intervallo di confidenza.

la camera inferiore nella realtà questa soluzione non assicurava performance ottimali, anche per via del tempo di campionamento abbastanza elevato.

In questo capitolo, nel nuovo contesto posto dalla definizione del problema, si propone una soluzione alternativa basata sulla seguente idea: iniziare a caricare l'integrale solo a regime. Tale costruzione è stata adeguatamente descritta in [13], e conferisce al sistema di controllo la caratteristica di essere *switching*, rientrando in una classe più complessa e nonlineare ma le cui prestazioni sono in genere migliori delle controparti lineari. Un diagramma riassuntivo del controllore progettato applicato al sistema da controllare è mostrato in Figura 4.6: si noti in particolare come il sistema sia stato modellato come una serie tra il modello identificato e un puro integratore della sua uscita, per ottenere la posizione lungo l'asse corrente restituita da PX4 sulla base della quale è calcolato l'errore pubblicato dall'Aruco Scanner.

La principale differenza di questo schema rispetto a un classico PI è la presenza di una commutazione internamente al blocco integratore: ad ogni istante di tempo il guadagno k_i del termine integrale può assumere uno tra due possibili valori: pari ad un guadagno costante opportunamente scelto o ad un coefficiente molto piccolo. La selezione di tali differenti guadagni dipende dallo stato del sistema d'errore che qui viene approssimato al secondo ordine, (e, \dot{e}) , nelle rispettive coordinate x ed y . È in questo che risiede la peculiarità di tale controllore, in quanto appunto switching: si tratta di un sistema dinamico nonlineare che integra al suo interno due dinamiche, la prima a tempo continuo corrispondente a quella di un PI e la seconda a tempo discreto, la quale ad ogni istante decide se commutare il guadagno del termine integrale da quello corrente all'altro. Volendo riassumere la teoria descritta in [13] supponiamo di disporre, come in questo caso è, di una misura del segnale d'errore $e(t)$. Definiamo pertanto il seguente funzionale di costo J :

$$J(t) = \int_0^{\infty} (w_1 e^2(\tau) + w_2 \dot{e}^2(\tau)) d\tau \quad (4.2)$$

dove la variabile indipendente t indica un generico istante di commutazione e $w_1, w_2 > 0$. Si può dimostrare che tale funzionale ammette svariati punti di minimo locale. È lecito supporre che il drone sia sufficientemente stabilizzato per poter essere mantenuto a

regime una volta raggiunto il target, ipotesi operativa che riporta nelle stesse condizioni del caso di specie trattato in [13]. È dunque possibile provare che il problema di controllo si traduce così in quello di determinare, tra tutti i possibili punti di minimo locale di $J(t)$, quello che corrisponde al minimo valore possibile di suddetto funzionale. Sia t^* tale punto, allora per conseguire l'obiettivo di controllo è sufficiente una sola commutazione del guadagno dell'integrale, in modo da iniziare a caricarlo solo una volta che si è raggiunta la condizione di regime, la quale corrisponde ad un box set nel piano (e, \dot{e}) che indica una situazione in cui il drone è sufficientemente vicino al target e sta iniziando ad assestarsi in sua prossimità; una rappresentazione esplicativa di tale insieme è offerta in Figura 4.7. Infine, un altro problema da risolvere riguarda il calcolo della derivata del segnale d'errore, ossia $\dot{e}(t)$. Essendo in presenza di un sistema di controllo evidentemente a tempo discreto in quanto operante sulla base di segnali campionati a frequenze fissate, nel presente lavoro è stato impiegato un algoritmo iterativo che calcola una derivata numericamente approssimata come un rapporto incrementale, i cui estremi corrispondono alle medie campionarie valutate su due finestre poste all'inizio ed alla fine di un buffer circolare di campioni. Le dimensioni del buffer e delle finestre sono state decise in base alle seguenti osservazioni:

- il buffer deve essere sufficientemente ampio da poter osservare variazioni significative del segnale, dunque se la costante di tempo del sistema che lo genera è pari a τ ed il tempo di campionamento è T_s , la dimensione deve essere sufficientemente maggiore di $\frac{\tau}{T_s}$;
- data la dimensione del buffer, le finestre devono essere ampie quanto basta affinché i loro centri distino temporalmente di un quantitativo comparabile a τ .

Il design del controllore è stato eseguito in Simulink per ciascuno dei due sistemi identificati, implementandolo in codice MATLAB comprensivo di opportune saturazioni della variabile di controllo, pari alle reali impostazioni eseguite su PX4 che limitano le velocità lineari in ogni modalità ad un'ampiezza massima di 0.5 m/s. Lo schema impiegato per il design e la taratura su ciascun asse prevede due sistemi da controllare identici ma soggetti l'uno all'azione del PI switching, l'altro a quella di un PI discretizzato tarato automaticamente da Simulink stesso. A titolo d'esempio si

k_p	0.9
k_i	0.3
σ_e	0.2 m
σ_{de}	0.1 m/s
σ_u	0.5 m/s
Dimensione buffer	16 campioni
Dimensione finestre	5 campioni

Tabella 4.1: Parametri di funzionamento ottimali dei controllori PI switching.

riporta in Figura 4.8 il modello costruito per l’asse y . Si noti in esso l’aggiunta del disturbo costante approssimante il vento sul ramo d’ingresso del sistema identificato, puramente per necessità implementative nella simulazione su Gazebo.

La simulazione è stata impostata posizionando inizialmente il sistema a 2 metri dal target. I risultati migliori con il controllore PI switching sono stati ottenuti su entrambi gli assi in corrispondenza dei parametri riportati nella Tabella 4.1. I risultati delle simulazioni con tali parametri sono presentati nelle Figure 4.9 e 4.10. Da essi si evince come le performance nel transitorio del controllore switching siano nettamente superiori a quelle della controparte standard: l’inserimento a regime dell’azione integrale evita del tutto le sovraelongazioni e le oscillazioni intorno al valore di regime, consentendo di raggiungere quest’ultimo poco dopo il tempo di salita, pari a circa 6 secondi. Durante la salita il segnale di controllo generato da entrambe le soluzioni satura immediatamente: ciò impedisce di velocizzare la risposta alzando il guadagno del termine proporzionale, o aggiungendo uno switch anche su di esso. Di contro, nella realtà tali saturazioni sono necessarie per evitare movimenti troppo bruschi compiuti dal drone in caso di problemi in questo o in altri punti dell’architettura, e sono state considerate anche in questa simulazione al fine di renderla il più possibile significativa.

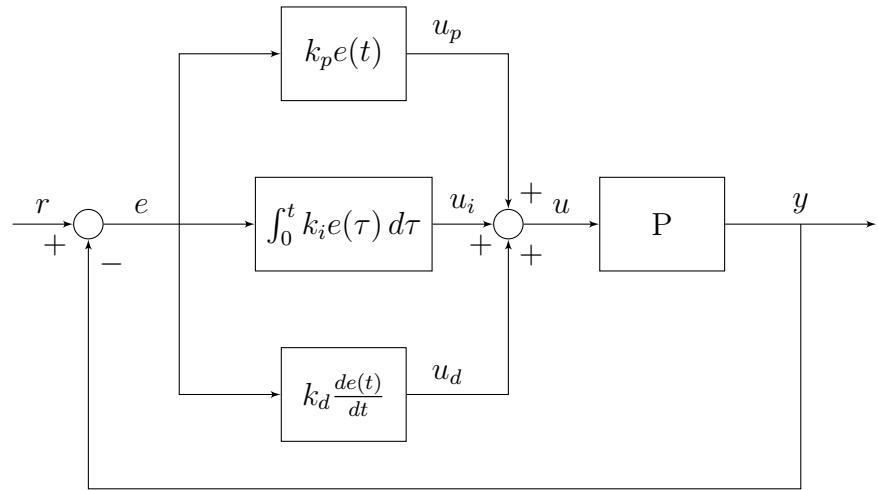


Figura 4.5: Diagramma a blocchi di un controllore PID ideale.

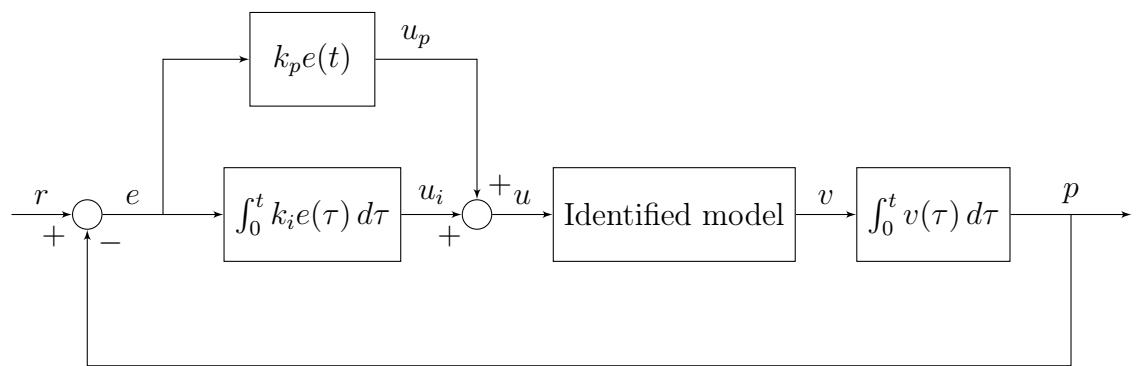


Figura 4.6: Diagramma a blocchi del sistema di controllo sviluppato.

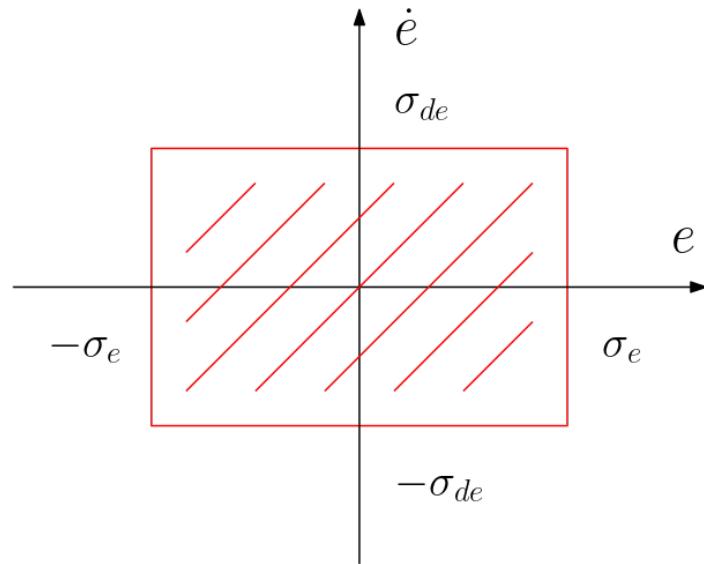


Figura 4.7: Box set rappresentante la condizione di regime nel piano (e, \dot{e}) .

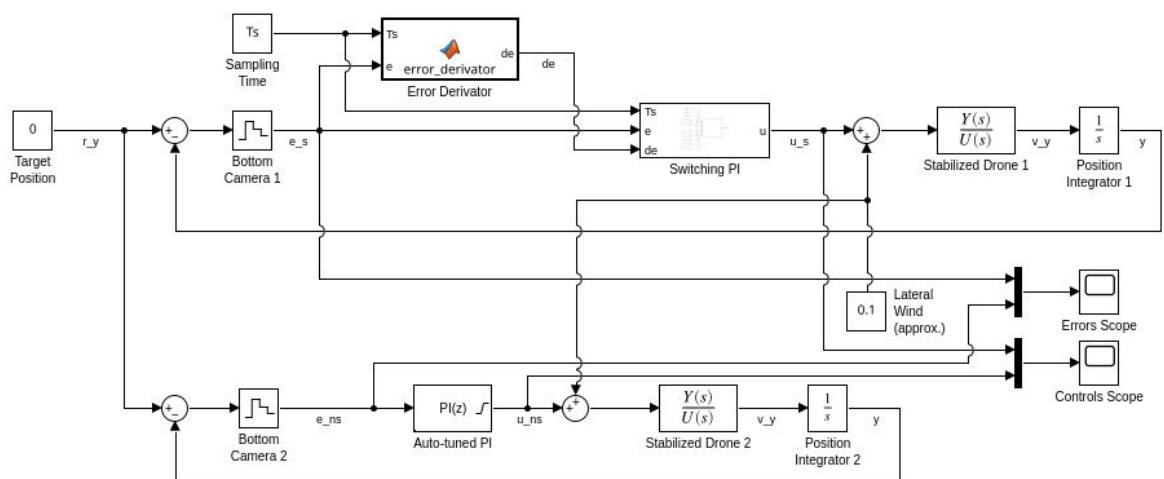


Figura 4.8: Modello impiegato per il design del controllore relativo all'asse y .

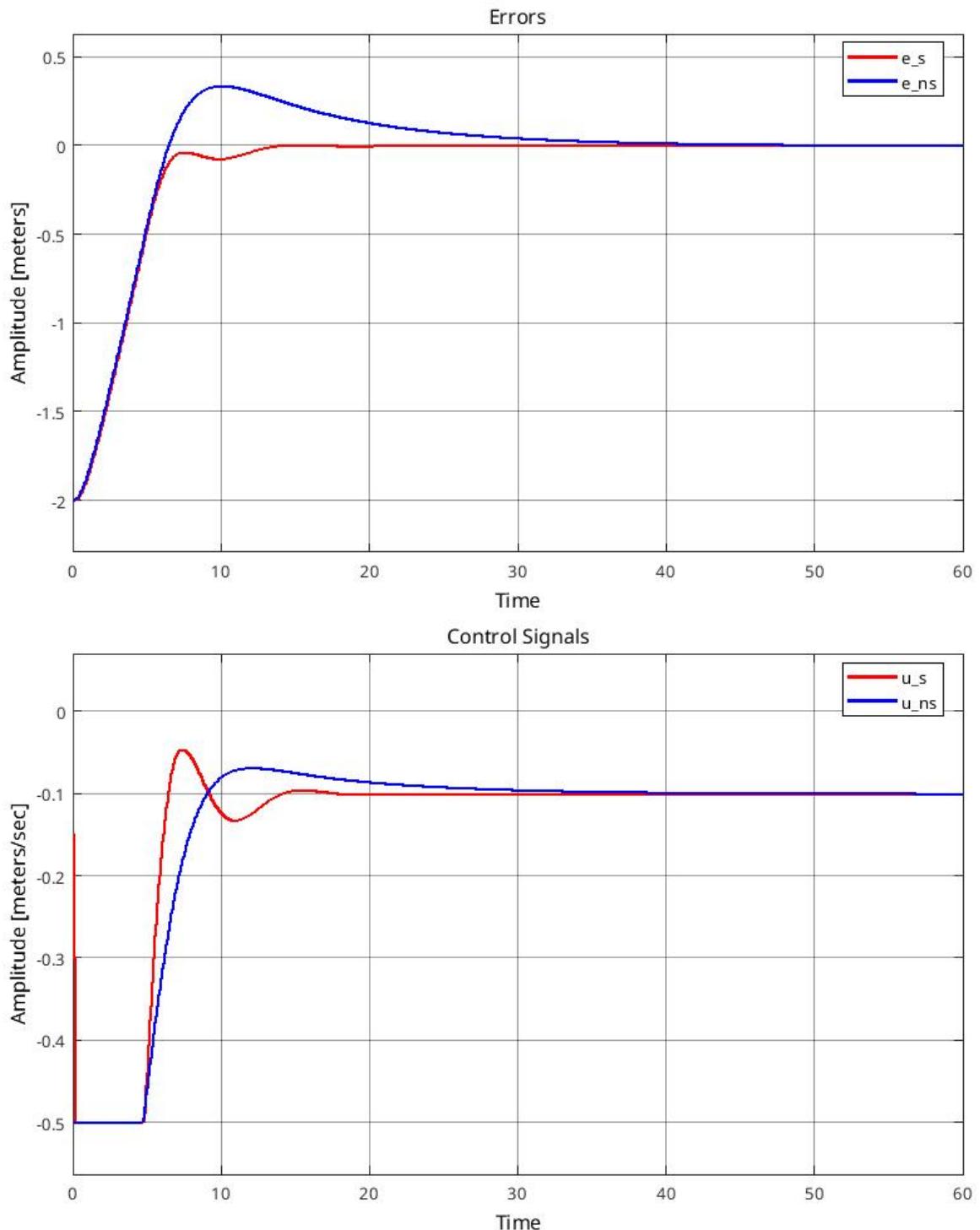


Figura 4.9: Confronto tra l'andamento dell'errore e del segnale di controllo generati dal controllore PI switching (rosso) e non-switching (blu) per l'asse x .

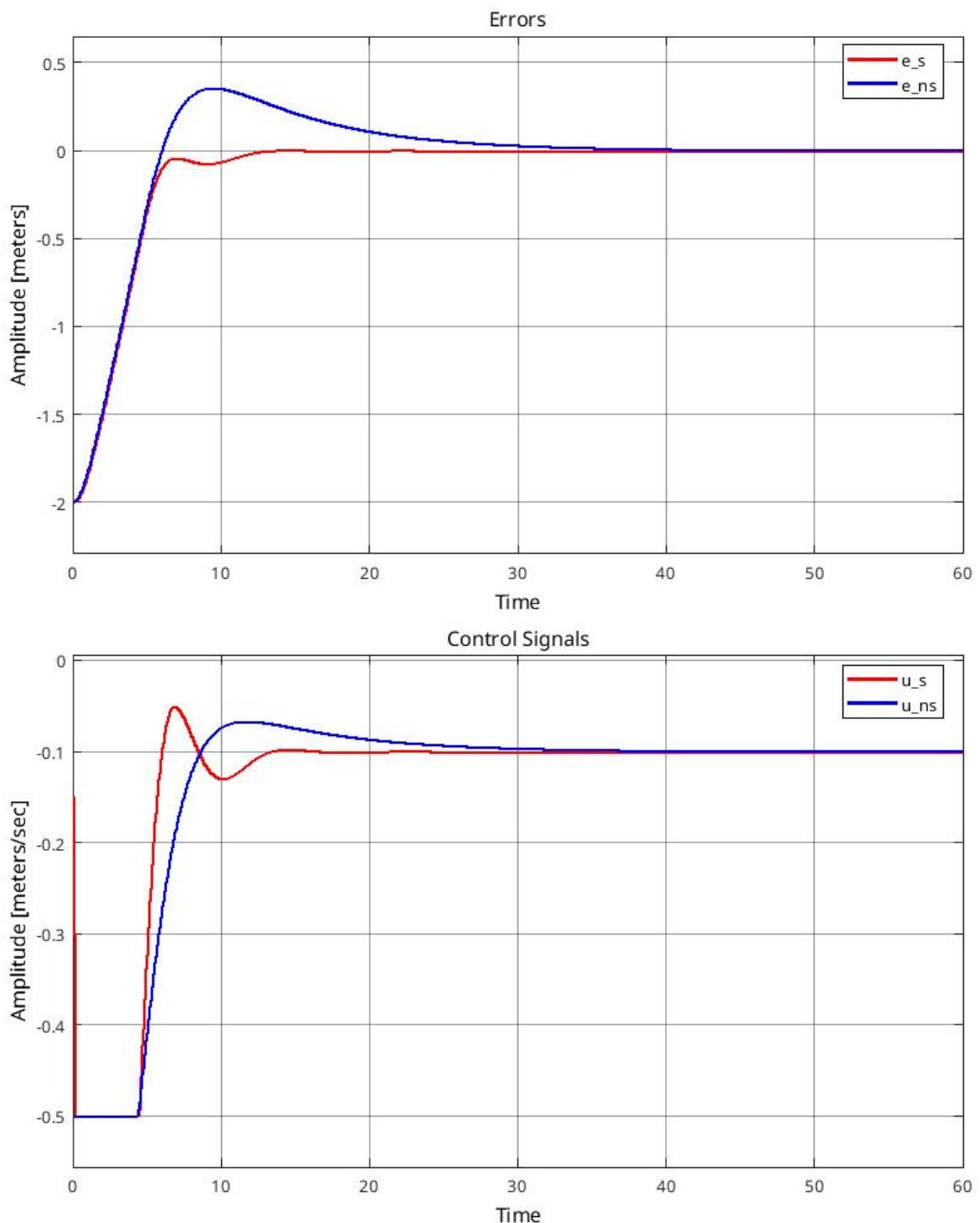


Figura 4.10: Confronto tra l'andamento dell'errore e del segnale di controllo generati dal controllore PI switching (rosso) e non-switching (blu) per l'asse y .

Implementazione in ROS 2

Il confronto effettuato nella sezione precedente è stato ripetuto in Gazebo. L'implementazione del PI non-switching era già stata realizzata ed è stata pertanto riadattata cambiando i guadagni, posti pari a quelli risultati ottimali per la versione switching. Quest'ultima ed i derivatori numerici sono stati invece implementati in C++ come classi, ed impiegati in una nuova callback dal topic /ArucoErrors con cui è stato compilato un nuovo eseguibile del package precision_landing, denominato *switching_pi*. Disponendo già del resto dell'architettura, nonché del codice del nodo precision_landing in cui integrare immediatamente la nuova callback, l'implementazione dei nuovi controllori ha richiesto uno sforzo molto contenuto. La simulazione in Gazebo è stata effettuata portando il drone a 2 metri di quota al di sopra di una replica della piazzola 1, applicando un vento trasversale con contributi sia sull'asse *x* che sull'asse *y*, e richiedendo il precision landing tramite l'apposito servizio. La posizione iniziale era la stessa a meno di minime oscillazioni del drone posto in hovering. I risultati sono riportati in Figura 4.11. Si nota immediatamente il passaggio al modello nonlineare: sono infatti presenti delle minime oscillazioni nel tracciato relativo al controllore switching. Esso porta comunque a termine l'allineamento in breve tempo, a seguito del quale viene richiesto l'atterraggio al Flight Control. La cosa che più salta agli occhi però è l'incapacità del controllore PI non-switching, implementato con gli stessi guadagni della controparte, di eseguire l'allineamento: nonostante ripetuti tentativi, di cui quello mostrato è uno, le oscillazioni indotte dal termine integrale erano eccessive e portavano la camera inferiore a perdere il target, situazione dalla quale la logica di supervisione usciva facendo fallire la procedura dopo un timeout come previsto. Questa simulazione, nonostante sia puramente indicativa, porta comunque a delle conclusioni interessanti: innanzitutto si è dimostrato come su un'architettura di controllo basata su ROS 2 sia stato possibile implementare efficacemente sistemi di controllo anche complessi, come appunto dei PI switching, e poi si è reso evidente come la classe dei controllori switching, anche nelle versioni più semplici, consenta di ottenere delle performance di molto migliori delle controparti non-switching anche in contesti problematici come quello in esame, caratterizzato da un elevato tempo di campionamento del segnale d'errore e da una notevole approssimazione del modello del sistema da controllare.

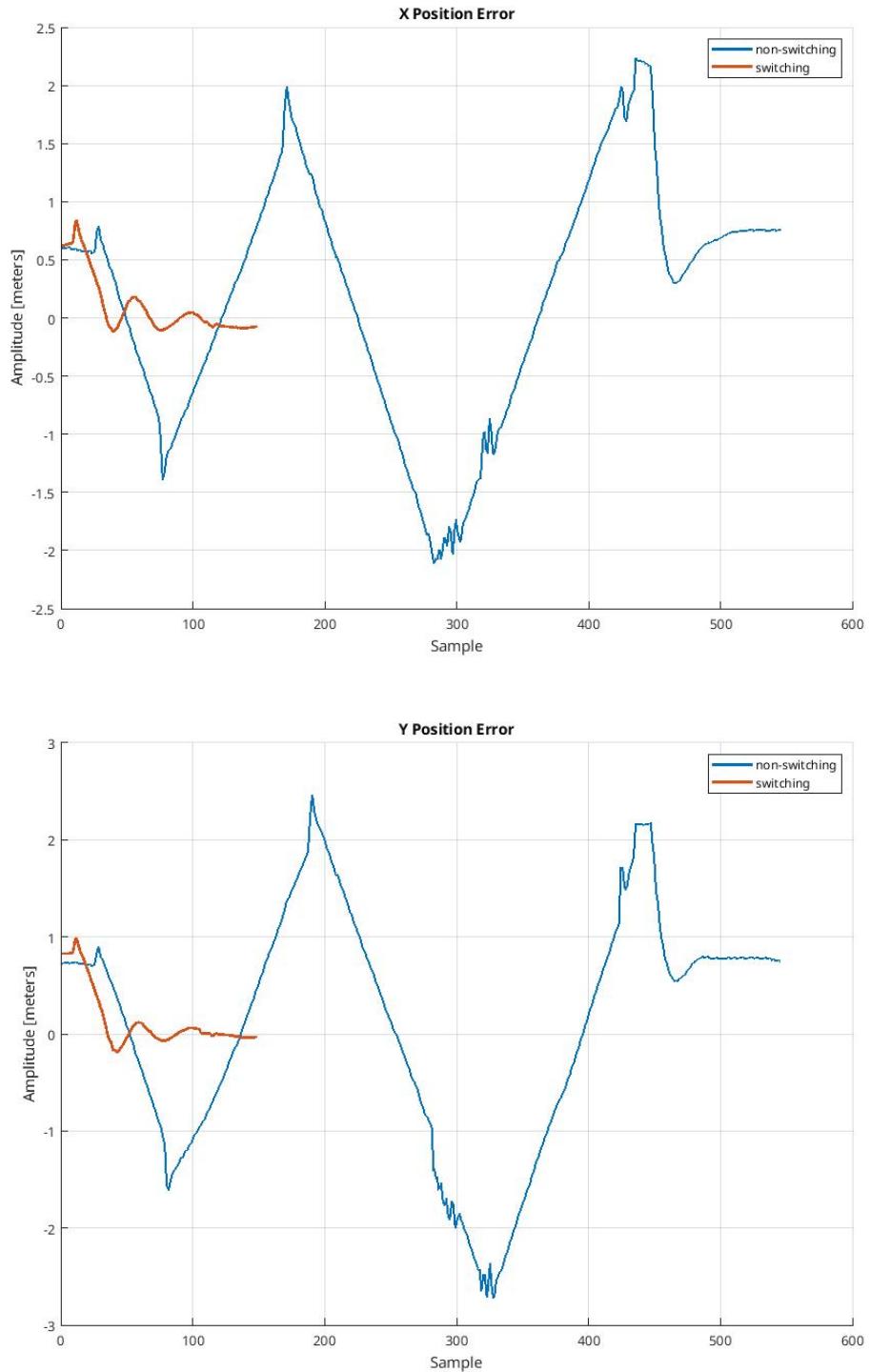


Figura 4.11: Andamento dell'errore indotto dai controllori PI switching (rosso) e non-switching (blu) durante un precision landing in Gazebo.

Capitolo 5

Conclusioni

In questo lavoro è stata presentata una nuova serie di soluzioni hardware e software finalizzate allo sviluppo ed al deployment di architetture di controllo per sistemi autonomi complessi. Nel Capitolo 2 è stato presentato il concetto di middleware come strato di astrazione intermedio tra l'hardware ed il software applicativo volto a semplificare alcune problematiche classiche della programmazione di apparati autonomi. Particolare attenzione è stata rivolta a ROS 2 come soluzione di nuova concezione basata su DDS e completamente open-source. Nel Capitolo 3 è stato presentato il caso di un drone autonomo per volo automatico, capace di localizzarsi in un ambiente privo di segnale GNSS e compiervi una missione basata su esplorazione dell'ambiente ed atterraggi multipli in zone predefinite. L'architettura di controllo è stata interamente costruita impiegando il middleware ROS 2 a supporto dello sviluppo del software di acquisizione dati, controllo e supervisione, nonché per gestire le comunicazioni tra i vari moduli e la postazione dell'operatore. Si è proceduto dunque ad un'accurata descrizione della sua implementazione e realizzazione e ad un commento dei risultati ottenuti. Nel Capitolo 4 è stato poi presentato un ulteriore caso di studio relativo all'implementazione su tale architettura di un controllore switching per l'esecuzione di atterraggi di precisione. Sono stati esposti nei dettagli i limiti operativi ed applicativi di una tale soluzione, e sono state descritte le varie fasi del processo di identificazione del sistema da controllare. Successivamente si è discussa l'implementazione del controllore ed i risultati ottenuti nelle varie prove in simulazione.

Sviluppi futuri

Il presente lavoro dimostra quali siano la versatilità e l'ampio ventaglio di possibilità aperte dalle soluzioni proposte. Come si è detto nel Capitolo 1, la direzione intrapresa dai recenti sviluppi tecnologici indica come le problematiche risolte da questi strumenti siano sempre più attuali, ed i risultati ottenuti ne provano l'efficacia e l'efficienza. Dal punto di vista dei middleware, ed in particolare dei DDS, il sempre più vicino deployment delle reti 5G aprirà alla possibilità di costruire architetture di controllo distribuite ad elevata capillarità, alta efficienza e bassa latenza, il caso d'uso per eccellenza di questo tipo di framework. Riguardo ROS 2 invece, nonostante l'ampia presentazione che se ne è fatta nel Capitolo 2 ed i numerosi rimandi alle parti tralasciate per brevità, è stata solo scalfita la superficie. Per favorirne l'integrazione con sistemi ampiamente diversificati, sono attualmente in sviluppo versioni ridotte di questo middleware che hanno come target sistemi embedded e a risorse limitate, particolarmente indicate per implementare controllori come quello descritto nel Capitolo 4, il quale nonostante tutto ha funzionato a dovere su un sistema general-purpose su cui era in esecuzione anche il resto dell'architettura più il simulatore. Queste soluzioni sarebbero da accoppiare con altre, sempre in sviluppo ma già aperte alla sperimentazione, volte a rendere i DDS compatibili con link di comunicazione diversi da quelli di Rete, come bus seriali o paralleli, in modo simile a quanto visto nel Capitolo 3. Un altro aspetto molto interessante ma che non è stato toccato in questo lavoro è poi quello della containerizzazione: ROS 2 offre già, mediante il proprio build system, la possibilità di compilare nodi e package come librerie condivise a caricamento dinamico. Ciò rende possibile attivare e disattivare i nodi a piacimento in modo totalmente configurabile, migliorando il carico e le prestazioni del sistema ma anche rendendo più efficiente l'implementazione in generale, dato che tutti i moduli attivi in un dato istante risiederebbero nello stesso *address space*. Sarebbe anche possibile gestire più semplicemente situazioni eccezionali come i guasti, nonché eseguire il testing dei singoli moduli e della loro integrazione, evitando di dover portare offline l'intera architettura per agire su anche solo un singolo modulo. Infine, un altro ambito da esplorare è quello dell'implementazione di schedulazioni hard real-time, nei limiti consentiti dall'impiego di un kernel Linux PREEMPT su un sistema quasi general-purpose, che coinvolgano i processi che fanno uso del middleware.

CAPITOLO 5. CONCLUSIONI

Riguardo lo specifico ambito dei droni, essere riusciti a metterne in volo uno così complesso e totalmente autonomo dimostra come molto altro si possa fare usando il framework costruito. Dal punto di vista della sensoristica di bordo si può integrare il sistema di localizzazione e mapping ORB_SLAM2 con altre misure, ottenute mediante algoritmi diversi e fuse secondo opportuni algoritmi di stima robustificati. È poi possibile lavorare sui livelli più bassi del sistema di controllo veloce, essendo anche questo totalmente aperto, per implementare leggi di controllo più sofisticate e raffinate, conseguendo nuove e migliori performance di volo. La disponibilità di un'architettura di controllo naturalmente distribuita e così facile da impiegare rende infine possibile costruire flotte di droni cooperativi, che realizzino autonomamente ed in gruppo task simili a quelli discussi. Infine, nel Capitolo 4 è stato presentato un caso d'applicazione di un sistema di controllo switching, mostrando come le performance sotto la sua azione migliorassero sensibilmente rispetto ad una soluzione standard. È questo un ambito che merita uno studio approfondito, corredata da prove sperimentali ed implementazioni reali rese semplicemente possibili da architetture come quella proposta. Relativamente al caso di specie si può studiare l'impiego di un controllore switching in feedback dallo stato, invece che dalla misura dell'errore, che preveda più d'una commutazione.

Elenco delle figure

2.1	Organizzazione del software in esecuzione su un calcolatore generico	15
2.2	Semplice esempio di schema generato da <i>rqt_graph</i>	33
3.1	Prototipo del drone (photo credit: Lorenzo Bianchi)	36
3.2	Mappa completa del campo di gara	36
3.3	Esempio di spettrogramma ricavato dai dati di un volo automatico	38
3.4	Modello CAD del pacco batteria, che funge anche da supporto per la camera inferiore	39
3.5	Modello CAD del supporto della camera frontale, inclinata verso il basso di 26.6°	39
3.6	Modello CAD della canopy superiore in TPU, a protezione del computer di bordo	40
3.7	Modello CAD del rinforzo triangolare per i carrelli del frame	40
3.8	Controllore di volo Pixhawk 4	43
3.9	Schema del controllore di volo multilivello implementato da PX4, con i diversi tempi di campionamento dei vari sottosistemi	43
3.10	Dettaglio del controllore PID con filtri per la stabilizzazione angolare . .	43
3.11	Dettaglio del controllore di PX4 per la regolazione dell'assetto basato sui quaternioni d'orientamento	44
3.12	Dettaglio del controllore PID con filtri per la regolazione della velocità lineare	44
3.13	Dettaglio del controllore proporzionale per la regolazione della posizione .	44
3.14	Architettura del link di comunicazione microRTPS Bridge	45
3.15	Intel RealSense D435i stereo depth camera	45
3.16	SoC Nvidia Jetson Xavier NX	46

3.17	Schema complessivo di nodi e topic attivi su drone e GCS.	50
3.18	Schema riassuntivo di nodi e topic del package <i>Flight Control</i> .	52
3.19	Organizzazione di nodi e topic relativi alle due camere Intel RealSense.	61
3.20	Organizzazione dei thread del sistema ORB_SLAM2.	65
3.21	Costruzione dei grafi mediante frames e loop closure effettuate dal sistema ORB_SLAM su uno dei dataset di test.	65
3.22	Organizzazione del nodo <i>image_grabber</i> .	66
3.23	Organizzazione del nodo <i>orbslam2_node</i> .	66
3.24	Organizzazione del nodo <i>aruco_scanner</i> .	71
3.25	Organizzazione del nodo <i>navigator</i> .	74
3.26	Risultato dell'algoritmo A* per il calcolo del percorso da piazzola 6 a 3.	76
3.27	Struttura del nodo <i>precision_lander</i> .	78
3.28	Struttura del nodo <i>fsm</i> .	82
3.29	Diagramma di stato della FSM di alto livello.	82
3.30	Diagramma di stato della meta-FSM contenuta nello stato <i>RUNNING</i> .	83
4.1	Dataset I/O relativo all'asse <i>x</i> .	89
4.2	Dataset I/O relativo all'asse <i>y</i> .	89
4.3	Confronto tra l'output misurato e quello approssimato (in blu) sul validation set dell'asse <i>x</i> . In tratteggio l'intervallo di confidenza.	90
4.4	Confronto tra l'output misurato e quello approssimato (in blu) sul validation set dell'asse <i>y</i> . In tratteggio l'intervallo di confidenza.	90
4.5	Diagramma a blocchi di un controllore PID ideale.	94
4.6	Diagramma a blocchi del sistema di controllo sviluppato.	94
4.7	Box set rappresentante la condizione di regime nel piano (e, \dot{e}).	95
4.8	Modello impiegato per il design del controllore relativo all'asse <i>y</i> .	95
4.9	Confronto tra l'andamento dell'errore e del segnale di controllo generati dal controllore PI switching (rosso) e non-switching (blu) per l'asse <i>x</i> .	96
4.10	Confronto tra l'andamento dell'errore e del segnale di controllo generati dal controllore PI switching (rosso) e non-switching (blu) per l'asse <i>y</i> .	97
4.11	Andamento dell'errore indotto dai controllori PI switching (rosso) e non-switching (blu) durante un precision landing in Gazebo.	99

Listings

2.1	Definizione del messaggio <i>sensor_msgs/Image</i>	26
2.2	Definizione delle classi dei nodi server e client del servizio <i>AddTwoInts</i>	29
3.1	Esempio di benchmark del SoC montato sul drone eseguito con la utility <i>cyclictest</i>	48
3.2	Esempio di funzione Bash per cambiare setpoint di posizione corrente.	51
3.3	Definizione del nodo <i>flight_controller</i>	52
3.4	Definizione del nodo <i>control_server</i>	53
3.5	Definizione del nodo <i>setpoint_server</i>	54
3.6	Definizione del nodo <i>px4_feedback_listener</i>	55
3.7	Definizione del nodo <i>orbslam2_node</i>	66
3.8	Definizione del nodo <i>image_grabber</i>	68
3.9	Definizione del nodo <i>aruco_scanner</i>	72
3.10	Definizione del nodo <i>navigator</i>	74
3.11	Definizione del nodo <i>path_finder</i>	76
3.12	Definizione del nodo <i>gcs</i>	77
3.13	Definizione del nodo <i>precision_lander</i>	78
3.14	Definizione del nodo <i>fsm</i>	84

Bibliografia

- [1] F. L. Bauer et al. *Software Engineering*. Rapp. tecn. North Atlantic Treaty Organization Science Committee, ott. 1968.
- [2] *OMG Data Distribution Service (DDS) Version 1.4*. Rapp. tecn. formal/2015-04-10. Object Management Group, apr. 2015.
- [3] *The Real-time Publish-Subscribe Protocol DDS Interoperability Wire Protocol (DDSI-RTPSTM) Specification Version 2.5*. Rapp. tecn. formal/2021-03-03. Object Management Group, mar. 2021.
- [4] Brian Gerkey. *Why ROS 2?* Rapp. tecn. Open Source Robotics Foundation, Inc. URL: https://design.ros2.org/articles/why_ros2.html.
- [5] Dirk Thomas. *Changes between ROS 1 and ROS 2*. Rapp. tecn. Open Source Robotics Foundation, Inc. URL: <https://design.ros2.org/articles/changes.html>.
- [6] William Woodall. *ROS on DDS*. Rapp. tecn. Open Source Robotics Foundation, Inc. URL: https://design.ros2.org/articles/ros_on_dds.html.
- [7] Dirk Thomas. *A universal build tool*. Rapp. tecn. Open Source Robotics Foundation, Inc. URL: https://design.ros2.org/articles/build_tool.html.
- [8] Dirk Thomas. *The build system 'ament_cmake' and the meta build tool 'ament_tools'*. Rapp. tecn. Open Source Robotics Foundation, Inc. URL: <https://design.ros2.org/articles/ament.html>.

- [9] Dirk Thomas. *Interface definition using .msg / .srv / .action files*. Rapp. tecn. Open Source Robotics Foundation, Inc. URL: https://design.ros2.org/articles/legacy_interface_definition.html.
- [10] D. Brescianini, M. Hehn e R. D'Andrea. *Nonlinear Quadrocopter Attitude Control Technical Report*. Rapp. tecn. ETH Zürich, 2013.
- [11] Raúl Mur-Artal, J. M. M. Montiel e Juan D. Tardós. «ORB-SLAM: A Versatile and Accurate Monocular SLAM System». In: *IEEE Transactions on Robotics* 31.5 (2015), pp. 1147–1163. DOI: 10.1109/TRO.2015.2463671.
- [12] Raúl Mur-Artal e Juan D. Tardós. «ORB-SLAM2: An Open-Source SLAM System for Monocular, Stereo, and RGB-D Cameras». In: *IEEE Transactions on Robotics* 33.5 (2017), pp. 1255–1262. DOI: 10.1109/TRO.2017.2705103.
- [13] L. Boncagni et al. «Performance-based controller switching: An application to plasma current control at FTU». In: *2015 54th IEEE Conference on Decision and Control (CDC)*. 2015, pp. 2319–2324. DOI: 10.1109/CDC.2015.7402553.