POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Monitoraggio in tempo reale di dispositivi robotici tramite i sensori di uno smartphone e l'utilizzo del framework ROS

Relatore

prof. Enrico Masala

Candidato

Davide Brau

Aprile 2019

Indice

Indice Figure

Capitolo I

Introduzione

Nell'ambito della robotica, l'attività di monitoraggio è una componente fondamentale per il funzionamento dell'intero sistema su cui questa viene svolta, sia esso controllato da un essere umano o da un algoritmo progettato per prendere le decisioni in maniera autonoma.
In questo contesto, con monitoraggio si intende infatti un processo svolto in modo continuo, o a intervalli regolari, che ha lo scopo di acquisire le informazioni provenienti dalle varie parti del sistema robotico, riferendosi in modo esplicito ai sensori, utili per poter esercitare un controllo efficace su di esso. Quest'attività presuppone il coinvolgimento di una serie fasi oltre all'acquisizione. Occorre infatti focalizzarsi anche sulle modalità di interfacciamento e sulle tipologie di trasmissione, tenendo presente che in questo momento quelli robotici, sono sempre più dei sistemi distribuiti e perciò l'attività di controllo viene spesso svolta da remoto.
Interfacciamento e trasmissione in questo caso sono due concetti che si intrecciano. Con interfacciamento si intende infatti quella fase associata alla definizione di una struttura comune, che possa essere utilizzata da due parti, diverse tra loro, che necessitano di comunicare. Ciò si concretizza con la definizione delle strutture dati e dei protocolli che andranno a caratterizzare la trasmissione stessa.
Il monitoraggio può essere svolto in diversi contesti, si fa pertanto una distinzione tra: sistemi che lavorano in un contesto non critico e sistemi che invece operano in un contesto critico, dove con criticità si intende l'importanza che assumono le tempistiche e la reattività, con cui viene svolta l'attività di monitoraggio, nei confronti del resto del sistema. Nel secondo caso questi vincoli risultano essere abbastanza stringenti, in quanto lo scopo è quello di voler osservare dei sistemi particolarmente sensibili e caratterizzati da una forte dinamicità, dove un controllo errato o troppo lento, può portare a un malfunzionamento, danneggiamento o addirittura distruzione del sistema stesso.
Nell'ambito robotico si ricade in questa seconda categoria, in quanto lo scopo del monitoraggio è finalizzato al controllo, spesso a distanza, del dispositivo, e quindi un errore o ritardo nella comunicazione di un dato, può rappresentare un problema anche grave. Si pensi ai robot impiegati nelle sale operatorie, ma anche solo ai droni in volo, ormai sempre più diffusi in diversi ambiti. In questi casi la ricezione errata di un valore, un dato mancante o che comunque non è stato ricevuto in tempi utili, possono compromettere l'operazione da svolgere.
Nell'ambito di questa attività di tesi, l'obbiettivo è stato quello di esplorare le possibilità e le soluzioni adatte per svolgere questo processo in un ambiente distribuito, utilizzando e analizzando le opportunità fornite dal framework: ROS. Per fare ciò, è stato necessario capire la logica utilizzata all'interno di questo ambiente per interfacciarsi con i vari moduli, ponendo particolare enfasi sull'infrastruttura di comunicazione e sui formati utilizzati per i vari dati; focalizzandosi in seguito su quelli multimediali, in particolare per quanto riguarda l'elaborazione e la trasmissione delle immagini. Nello

specifico, dopo uno studio che ha riguardato l'ambiente di sviluppo nel suo complesso, sono state esplorate le varie modalità messe a disposizione da ROS per l'acquisizione dei dati e la loro trasmissione da remoto.
Nella seconda parte si è cercato di mettere in pratica le nozioni acquisite, attraverso lo sviluppo di un'applicazione Android basata su ROS, quest'ultima si propone di fornire un supporto concreto durante l'attività di monitoraggio. Essa infatti consente di acquisire una grande varietà di informazioni, grazie alla moltitudine di sensori presenti sui dispositivi odierni, utili per fornire un supporto aggiuntivo e a basso costo per le operazioni di controllo, nel momento in cui il dispositivo in questione viene messo a bordo di un qualsiasi tipo di drone o rover.
Si è quindi cercato di evidenziare le varie criticità dovute: sia ai limiti computazionali e di reattività tipici dei dispositivi mobili, sia alle caratteristiche della rete utilizzata. Infatti, questi sono entrambi fattori che possono determinare l'aumento delle latenze o la perdita dei dati, elementi tollerabili entro un certo limite, se si vuole che l'attività sia finalizzata a un controllo real-time.
L'applicazione è stata sviluppata nell'ambito di un progetto più ampio che comprende il lavoro svolto in parallelo dal Dott. Federico Barone, che si è occupato dello sviluppo di un'applicazione web in grado di comunicare con l'ambiente ROS. Grazie a questa è possibile interfacciarsi con il dispositivo Android, attraverso un'interfaccia grafica accessibile dal browser.
Infine, l'ultima parte di questa attività, ha visto coinvolte entrambe le applicazioni in un utilizzo congiunto, ed è terminata con una prova effettuata grazie all'ausilio del rover Clearpath Jackal, quest'ultima è risultata particolarmente utile per verificare in un contesto reale le difficoltà e i limiti che ci ritrova a dover affrontare nello sviluppo di questo tipo applicazioni.

Capitolo II

ROS: Robot Operating System

2.1 Panoramica

ROS può essere definito come un meta-sistema operativo di tipo open-source, ovvero un insieme di processi in esecuzione su una o più macchine, capaci di comunicare tra loro e di seguire una logica comune; esso è in grado di fornire tutti quei servizi che ci si aspetta da un qualsiasi sistema operativo, tra cui l'astrazione dell'hardware, il controllo dei dispositivi a basso livello, l'introduzione delle funzionalità necessarie per la comunicazione tra processi e per la gestione dei programmi. Infine, fornisce anche gli strumenti e le librerie indispensabili per creare, compilare ed eseguire il codice, che sarà poi in grado di essere utilizzato su più dispositivi e di comunicare con le altre parti del sistema attraverso essi.
In questo senso, l'architettura di un sistema ROS può essere rappresentata a livello logico attraverso un grafo orientato. I processi possono essere distribuiti potenzialmente su più macchine, che comunicano tra loro grazie a un'infrastruttura di comunicazione pensata per garantire robustezza attraverso l'indipendenza e il disaccoppiamento tra le varie parti, che in questo caso rappresentano delle unità a sé stanti in grado di svolgere il loro compito senza dover dipendere direttamente da altri elementi del sistema.
ROS implementa diversi tipi di comunicazione, includendo comunicazioni sincrone in stile RPC attraverso quelli che nella terminologia vengono identificati come "ROS *Services*", comunicazioni asincrone: attraverso la condivisione di un flusso di dati serializzati in un formato opportuno tramite i cosiddetti "*topic*", seguendo il modello *Publish-Subscribe*; infine è possibile che un processo intervenga sul comportamento di un altro, interagendo con uno specifico elemento di cui si parlerà in seguito, chiamato *"parameter server"*.
I processi in esecuzione nel sistema prendono il nome di nodi, ciascun nodo è in grado di comunicare con gli altri grazie alla presenza di un ulteriore elemento chiamato master, che si occupa di gestire le richieste provenienti dai vari nodi e di fornire le informazioni necessarie affinché questi ultimi siano in grado di comunicare tra loro.
I nodi a loro volta sono organizzati in package, ciascun package oltre al codice sorgente, conterrà sempre al suo interno i file necessari per la compilazione e la risoluzione delle dipendenze oltre a un file manifest, dove sono specificate le informazioni necessarie per la condivisione e l'importazione del package in questione, all'interno di un generico sistema basato su ROS.

ROS topic: il modello publish-subscribe

Il modello *publish-subscribe* garantisce la completa autonomia tra le sorgenti che generano le informazioni, che in questo contesto prendono il nome di *publisher*, e coloro che le ricevono ovvero i *subscriber*, l'unico riferimento che accomuna le due parti prende il nome di *topic*.

Il funzionamento prevede che una o più sorgenti pubblichino dei dati su un determinato topic sotto forma di messaggi e che i destinatari interessati ad essi effettuino la sottoscrizione allo stesso topic per poterli ricevere, il tutto senza che nessuna delle due parti a livello logico interagisca direttamente con l'altra.

L'utilizzo del paradigma publish-subscribe rende più resistente il sistema nei confronti di eventuali guasti o errori e semplifica la comunicazione tra le entità coinvolte, evitando, a livello di librerie client, di dover implementare interfacce Socket o meccanismi di sincronizzazione, lasciando quindi che sia l'infrastruttura sottostante ad occuparsene. Tuttavia, oltre ai vantaggi elencati, il modello publish-subscribe presenta anche alcuni limiti che rendono necessario l'utilizzo di un'altra soluzione, affrontata nel paragrafo successivo, per ottenere specifici comportamenti altrimenti irrealizzabili.

ROS Service: la soluzione client-server

Il modello *publish-subscribe* rappresenta un paradigma di comunicazione molto flessibile, ma il concetto di comunicazione "molti a molti" su cui è basato risulta inadatto in quelle situazioni in cui è necessario ricevere un servizio effettuando una richiesta per la quale ci si aspetta una risposta in qualche modo legata ad essa. Questo comportamento è tipico delle architetture Client-Server, spesso richieste all'interno dei sistemi distribuiti. Per questo motivo sono stati introdotti i *"ROS Service"*, ognuno di essi definisce una copia di messaggi: richiesta - risposta.

Il nodo che fa da server, offre il proprio servizio dichiarandone il nome sotto forma di stringa, il generico client accede al servizio inviando un messaggio di richiesta, secondo il formato prestabilito, e attende una risposta in relazione ad essa. Le librerie Client solitamente presentano questa interazione al programmatore come una classica chiamata di funzione.

I ROS-Service vengono definiti utilizzando dei file con formato "srv", che vengono poi compilati all'interno del codice sorgente delle librerie Client. Un client può rendere persistente la connessione verso un determinato servizio, ottenendo in questo modo alte performance, ma introducendo allo stesso tempo una minore solidità rispetto ai cambiamenti del server.

A ciascun servizio viene associato a un "tipo", derivato dal nome dello stesso file che lo definisce, all'interno di quest'ultimo è definita la struttura del messaggio utilizzato. Oltre al tipo di servizio, i ROS-Service sono dotati di un controllo di versione, realizzato mediante l'utilizzo di un digest MD5, sempre ottenuto a partire dallo stesso file citato in precedenza. Un nodo può usufruire di un servizio solo se il tipo a cui appartiene e il digest sono verificati, questo assicura che sia il client che il server siano stati realizzati in modo coerente.

La comunicazione tra client e server avviene tramite un meccanismo sincrono, gestito a livello di programmazione con una *callback* lato server e una chiamata di funzione con effetto bloccante, lato client.
Un classico utilizzo di ROS-Service si ha ad esempio quando si desidera ricevere una specifica informazione o richiedere di portare a termine un determinato compito (andare dal punto A al punto B) dove può risultare ostico, inefficiente o spesso impossibile l'utilizzo dei topic.

Package
Il software all'interno di ROS è organizzato in *package*. Un package può contenere: il codice di uno o più nodi, definizioni di messaggi ROS, classi di supporto per la comunicazione e il trasporto, classi per i test, ecc..; tuttavia, oltre agli elementi prettamente attinenti a ciò che riguarda l'architettura ROS, è anche possibile inserire librerie indipendenti da quest'ultima, file di configurazione, pezzi di software di terze parti e in generale qualsiasi altra cosa che possa costituire un modulo a sé stante a livello logico. Per questo motivo, il package viene considerato come l'unita più piccola (atomica) in grado di essere compilata singolarmente all'interno dell'ambiente ROS. Il suo scopo è quello di fornire le proprie funzionalità nel modo più semplice possibile cosicché il software al suo interno possa essere facilmente riutilizzato. In generale, un package dovrebbe includere al proprio interno abbastanza funzionalità da poter essere considerato utile, ma non così tante da renderlo pesante e difficile da utilizzare dal resto del software. È possibile realizzare un package, sia creando le cartelle necessarie manualmente, sia con appositi strumenti, ad esempio il comando *catkin_create_pkg*.

Stack
I package all'interno dell'ambiente ROS possono essere raggruppati all'interno degli *stack*. Mentre lo scopo dei package consiste nel creare l'insieme di minimo classi, in grado di funzionare in modo indipendente per raggiungere uno scopo, l'obbiettivo degli stack vuole essere quello di raggruppare quei package che collettivamente forniscono delle funzionalità e al contempo hanno una serie di dipendenze reciproche, al fine di semplificare il processo di condivisione del codice.
Infatti, gli stack costituiscono il meccanismo primario all'interno di ROS per la distribuzione del software. Ogni stack è associato a una versione e può dichiarare delle dipendenze da altri stack. Anche le singole dipendenze sono associate a una specifica versione, questo rende la fase di sviluppo più stabile, evitando che l'aggiornamento di uno stack comprometta il funzionamento degli altri che ne fanno utilizzo.

Nodi
I nodi identificano i processi all'interno del sistema, ROS è progettato per essere il più possibile modulare in ogni suo aspetto, anche nello svolgimento delle operazioni più basilari, infatti, un sistema di controllo robotico solitamente comprende diverse unità, distinguibili oltreché a livello fisico anche da un punto di vista logico. Ogni nodo deve essere in grado di funzionare indipendentemente, così da garantire una certa

robustezza, in termini di tolleranza rispetto ad eventuali: aggiornamenti, bug o guasti degli altri elementi.
Inoltre, l'introduzione di un alto livello di modularità, ha il vantaggio di ridurre la complessità del codice rispetto agli equivalenti sistemi monolitici, i dettagli implementativi di uno dei moduli non sono fondamentali per gli altri elementi. Tra l'altro, si ha il grande vantaggio di poter sviluppare le diverse parti in linguaggi diversi per sistemi diversi, con un unico elemento comune, ovvero l'infrastruttura di comunicazione, che come già detto non è quasi mai gestita direttamente dal programmatore.
Un nodo durante la comunicazione può svolgere uno o più ruoli tra quelli previsti, perciò può essere allo stesso tempo: publisher, subscriber, client e server; in base al compito ad esso attribuito. Il modo più comune per avviare un nodo ROS è attraverso l'utilizzo del comando: *rosrun*, seguito dal nome del *package* di cui il nodo fa parte, a sua volta seguito dal nome dell'eseguibile del nodo; infine, è possibile specificare gli eventuali argomenti da linea di comando utilizzabili durante la fase di configurazione.
All'interno della stessa macchina, come spesso capita, possono essere presenti più nodi, questi possono essere di tipi diversi e ricoprire ruoli differenti, ma è anche possibile avere contemporaneamente in esecuzione più nodi dello stesso tipo.
Lo scopo è arrivare ad un livello di astrazione rispetto al quale nel codice dei singoli nodi, nel momento in cui essi comunicano con gli altri, non ci sia differenza tra la comunicazione fra nodi eseguiti sulla stessa macchina e nodi in esecuzione su altri dispositivi, che comunicano tra loro sfruttando rete, in quanto ci si limita a pubblicare e ricevere messaggi sui topic o a usufruire di servizi attraverso quelle che vengono presentate dalle librerie client come semplici chiamate di funzione, ma che in realtà possono prevedere o meno il coinvolgimento di: socket, protocolli di rete, ecc..; perciò il comportamento in seguito alla chiamata delle medesima funzione potrà variare notevolmente in base alla posizione relativa dei moduli coinvolti, il tutto è completamente trasparente al programmatore.

I messaggi ROS

All'interno dell'architettura ROS ogni nodo comunica con gli altri pubblicando le informazioni sui topic e utilizzando delle strutture dati contenenti diversi campi che variano in base al contesto e che prendono il nome di messaggi.
Ciascun messaggio può essere definito sulla base dei tipi standard: int, float, boolean, string, ecc.. ; ma è anche possibile definire degli array. Infine, esiste la possibilità di definire nuovi tipi di messaggio utilizzando come campi dei messaggi preesistenti, realizzando così una struttura annidata più o meno complessa.
La struttura dei singoli messaggi viene definita all'interno di file con estensione: ".msg" all'interno della sottocartella "msg" che andrà creata all'interno del package.
Come anticipato, un messaggio può a sua volta includere al suo interno altri messaggi ROS sotto forma di campi, un caso particolare è il campo: *header*, che prende il nome dall'omonimo tipo e dovrebbe essere presente in tutti i messaggi ROS. Al suo interno include alcune informazioni comuni come: il numero di sequenza, il *timestamp* e il *frame-ID*; ognuno di essi può essere impostato o dal programmatore oppure settato automaticamente dalle librerie client durante la fase di pubblicazione.

Il numero di sequenza è un intero su 32 bit e viene automaticamente incrementato per ogni messaggio inviato da un dato publisher. Il timestamp è costituito a sua volta da una struttura chiamata: *"Time"*, che definisce al suo interno i campi: sec e nsec, che come suggeriscono i nomi, conterranno rispettivamente i secondi e nanosecondi dell'istante inserito. Il timestamp può avere una semantica differente in base al contesto, ad esempio se il messaggio serve per impartire un comando a un nodo remoto il timestamp potrà contenere l'istante in cui il messaggio è stato inviato, così da avere ben chiare le tempistiche, mentre se ad esempio il messaggio contiene i dati acquisiti da un sensore, il timestamp potrà contenere l'istante in cui è stato acquisito il nuovo dato, che sarà diverso dall'istante di pubblicazione; perciò è consigliabile specificare sempre la convenzione utilizzata per questo campo nella documentazione del messaggio, così da evitare le ambiguità.
I vari nodi possono anche scambiarsi informazioni attraverso particolari tipi di messaggi, già introdotti in precedenza, che prevedono sempre una coppia: *request-response*. Questi ultimi vengono impiegati all'interno dei ROS Service e la loro struttura viene definita all'interno di appositi file con estensione: ".srv", è possibile collocare anch'essi all'interno della cartella "msg".

2.2 Protocolli di trasporto

Esistono diverse alterative per l'invio dei dati attraverso la rete e ognuna di queste porta con sé vantaggi e svantaggi, che dipendono soprattutto dal tipo di applicazione che si vuole realizzare.
L'intera architettura ROS si basa a livello rete sul protocollo IP mentre a livello trasporto sono disponibili due alternative a seconda che si scelga di utilizzare TCP o UDP.
Il maggiore utilizzo di TCP è dovuto al fatto che esso garantisce un meccanismo di comunicazione semplice ed affidabile, i pacchetti arrivano sempre in ordine e vengono ritrasmessi in caso di perdite. Sebbene queste caratteristiche siano utili nella maggior parte dei casi, spesso rappresentano un problema in reti con un alto tasso di perdite, dove la maggior parte del tempo viene perso effettuando la ritrasmissione dei pacchetti smarriti e attendendo la scadenza dei vari time-out del protocollo, ciò causa una progressiva riduzione della quantità di dati trasmessi nell'unità di tempo e di conseguenza un rallentamento nel funzionamento dell'intero sistema; in questo caso sarebbe più appropriato utilizzare UDP.
Quest'ultimo inoltre, risulta più efficiente anche nelle situazioni in cui più nodi di una stessa sottorete sono iscritti a uno stesso topic, la trasmissione dei messaggi a questo punto può essere fatta utilizzando UDP broadcast, riducendo notevolmente il numero di connessioni *peer-to-peer* tra i singoli nodi.
Per questi motivi all'interno di ROS è stato introdotto un supporto ad entrambi i protocolli, l'utilizzo di uno o dell'altro viene gestito passando attraverso una fase di negoziazione dove il subscriber può specificare il tipo di trasporto da utilizzare, tuttavia,

è il server XMLRPC del publisher a stabilire quale protocollo verrà effettivamente adottato in base alla compatibilità con esso. Di default se il subscriber non lo specifica, viene scelto TCPROS, altrimenti si può valutare anche l'utilizzo di UDPROS.

2.2.1 TCPROS

Come suggerisce il nome, questo livello di trasporto sfrutta lo stack TCP/IP per il trasporto dei dati. Le connessioni in entrata vengono ricevute dal socket server TCP, ogni connessione è caratterizzata da un "connection header" TCPROS costituito da diversi campi. Grazie ad esso è possibile stabilire il tipo di messaggio ROS trasportato e ricavare le informazioni riguardanti il routing dei pacchetti.
Una prima differenziazione viene fatta in base alla presenza o meno nell'header del campo topic o service, nel primo caso il messaggio verrà instradato in una connessione verso un ROS Publisher mentre nel secondo sarà destinato a un ROS Service.
Durante l'iscrizione a un topic, il subscriber deve inviare i seguenti campi: definizione del messaggio, il proprio nome, il nome del topic a cui ci si sta connettendo, il *digest* md5 ottenuto a partire dal *message-type* e il tipo di messaggio relativo al topic; il subscriber può inoltre specificare se utilizzare o meno, mediante un flag, il: "TCP_NODELAY"; che determina il comportamento dei *time-out* nel protocollo tcp.
Dall'altro lato il publisher che dichiara di voler trasmettere i dati su un topic deve a sua volta rispondere agli eventuali iscritti con cui è stato possibile avviare la connessione inviando: il digest md5 calcolato a partire dal message-type e il message-type stesso. Il calcolo del digest in entrambi i casi è necessario per stabilire da ciascuna parte, se l'altra è in grado di produrre/leggere lo specifico messaggio, dichiarato per quel topic. Il publisher può eventualmente inviare come informazione aggiuntiva il proprio nome, infatti, sebbene questo campo non sia obbligatorio, il suo impiego risulta particolarmente utile in fase di *debugging*.
Un'altra informazione che il publisher è in grado di specificare, è se si sta utilizzando o meno la modalità latch, quest'ultima consiste nel salvataggio dell'ultimo messaggio pubblicato sul topic. In questo modo, ogni volta che un nuovo subscriber completa la propria iscrizione al topic, gli si potrà in ogni caso recapitare l'ultimo messaggio disponibile, senza dover attendere la prossima pubblicazione.
Oltre alle connessioni relative ai topic, TCPROS viene utilizzato anche per i ROS Service, un nodo client deve inserire nella propria richiesta: il proprio nome (che lo identifica in maniera univoca), il nome del servizio che sta richiedendo, il digest md5 relativo al tipo di servizio e il message type associato ad esso; può opzionalmente specificare tramite un flag apposito se vuole rendere persistente la connessione al servizio, così da avere la possibilità di effettuare richieste multiple senza dover ogni volta ripetere la fase di negoziazione, questo comporterà inevitabilmente una perdita di flessibilità e robustezza. Dall'altro lato il server può specificare il proprio nome, per lo stesso motivo indicato in precedenza nel caso del publisher. L'header TCPROS consente inoltre di specificare se utilizzare o meno per i messaggi di errore, il formato: "human-readable", nel caso in cui non sia stato possibile stabilire la connessione.

Nel caso specifico dei ROS Service è necessario citare il flag: "ok", presente nella risposta verso il client: se il suo valore è uguale ad uno, significa che la richiesta per quel servizio è andata a buon fine e seguirà un messaggio contenente la risposta, nel caso in cui il valore è zero, ciò indica che seguirà un messaggio serializzato contenente la stringa con la descrizione dell'errore riscontrato.

2.2.2 UDPROS

UDPROS è un protocollo di livello trasporto definito per lo scambio di messaggi ROS che sfrutta lo standard UDP per il trasporto dei dati serializzati.
Lo scopo di questo protocollo è quello di sfruttare i vantaggi offerti da UDP ed è pensato per applicazioni che necessitano di bassa latenza, anche a discapito di un trasporto affidabile, un esempio può essere lo streaming multimediale. Come nel caso riguardante TCP anche per UDPROS viene definito un header con formato XML, utilizzato durante la comunicazione. Ogni datagram UDP contiene un proprio header UDPROS, seguito dal messaggio vero e proprio. Il formato è il seguente:

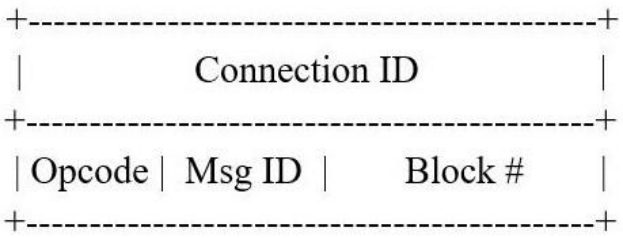

Figura 1 connection-header UDPROS

Connection ID: è un valore su 32-bit che viene determinato durante la fase di negoziazione ed è utilizzato per stabilire a quale connessione è destinato il datagram, questo parametro consente al singolo socket di gestire più connessioni UDPROS alla volta.
Opcode: UDPROS supporta diversi tipi di datagram, che è possibile identificare grazie all'utilizzo di questo campo compost da 8-bit, il quale può assumere i seguenti valori:

- DATA0 (0) – viene inviato nel primo datagram contenente il messaggio ROS
- DATAN (1) – viene utilizzato in tutti i datagram successivi che fanno parte dello stesso messaggio.
- PING (2) – utilizzato con funzione di *"keep-alive*", viene inviato periodicamente per notificare all'altro *peer* che la comunicazione è ancora attiva.
- ERR (3) – viene utilizzato per notificare che la connessione è stata chiusa inaspettatamente.

Message ID – valore su 8-bit che viene incrementato ad ogni nuovo messaggio ed è utilizzato per stabilire se uno o più datagram sono stati scartati o sono andati perduti durante il tragitto in rete.

Block # - questo valore su 16-bit è solitamente zero, a meno che l'Opcode sia uguale a DATA0, in tal caso questo campo contiene il numero di pacchetti necessari per completare il messaggio ROS.

Un messaggio ROS viene perciò inviato tramite uno o più datagram UDPROS, il primo datagram contiene il numero totale di pacchetti in cui il messaggio verrà diviso, è ognuno dei successivi conterrà un header con un numero di sequenza calcolato rispetto al totale indicato nel primo pacchetto, seguito dal messaggio parziale.
La massima dimensione del singolo pacchetto viene stabilita durante la fase di negoziazione e verrà applicata a tutti i datagram ad eccezione dell'ultimo, che ovviamente conterrà meno dati.

2.2.3 Transport Hints

Normalmente il tipo di trasporto utilizzato di default per le comunicazioni tra i singoli nodi è TCPROS, tuttavia è possibile condizionare questa scelta, specificando durante la fase di iscrizione ad un determinato topic, all'interno del metodo subscribe, i cosiddetti *"transport hints"*, nel seguente modo:

```
ros::Subscriber sub = nh.subscribe("topic_name", 1, callback,
                                   ros::TransportHints().unreliable());
```

I transport hints consentono al subscriber di specificare le proprie preferenze durante la fase di negoziazione della connessione, sia per la scelta del protocollo, sia per quella dei parametri ad esso associati; sarà in ogni caso il publisher a decidere quali utilizzare, sulla base dei tipi che è in grado di supportare. I vari parametri non sono per forza di tipo esclusivo, infatti, può anche essere specificato un ordine preferenziale, ad esempio utilizzando un'espressione di questo tipo:

```
ros::TransportHints().unreliable().reliable().maxDatagramSize(1000).tcpNoDelay();
```

Seguendo l'ordine con cui sono inseriti i vari metodi: "unreliable" specifica che si preferisce utilizzare il protocollo UDP, ma nel caso in cui esso non fosse supportato, la connessione non deve essere rifiutata, ma si accetta anche l'utilizzo di TCP ("reliable"). Dopodiché, per il caso UDP viene specificata la massima dimensione per il singolo datagram, mentre per TCP viene specificato l'utilizzo della modalità "no delay". Queste specifiche verranno poi tradotte nei vari messaggi scambiati con il server XMLRPC del publisher, durante la fase di negoziazione. Attualmente non è possibile specificare i *transport-hints* lato publisher, ma si sta valutando se introdurre questa possibilità in futuro.

2.3 La serializzazione dei messaggi ROS e i digest MD5

I messaggi all'interno di ROS sono serializzati secondo una rappresentazione molto compatta, che più o meno corrisponde alla serializzazione di una struttura del linguaggio C secondo un formato *little-endian*.
Tale rappresentazione, comporta il fatto che due nodi per comunicare tra loro, devono essere in grado di stabilire con esattezza se il formato dei messaggi che andranno ad utilizzare coincide o meno. Per effettuare questa verifica, non basta basarsi sul nome del messaggio (*message-type*), ma è necessario avere un qualche controllo di versione, che garantisca a fronte dell'utilizzo dello stesso tipo di messaggio, che quest'ultimo non abbia subito modifiche, diventando incompatibile oltre che con i tipi diversi dal suo, anche con le altre versioni del suo stesso tipo.
Nel sistema ROS questa verifica sulla versione viene fatta svolgendo un calcolo sul testo del messaggio, nello specifico viene utilizzato un particolare tipo di digest MD5.
In generale, le librerie client non consentono al programmatore di effettuare direttamente il calcolo di questo digest, esso è infatti calcolato automaticamente durante la fase di build (compilazione e linking) del package in cui il messaggio è definito, dopodiché il risultato ottenuto viene direttamente inserito all'interno della classe associata al messaggio, definita nel codice sorgente, anch'esso generato in modo automatico durante il build.
Questa classe sarà poi accessibile nel codice dei vari nodi che ne effettueranno l'inclusione, a questo punto si potranno creare delle nuove istanze e utilizzarne i metodi; tra questi vi è anche quello consente di ottenere il digest calcolato in precedenza, utile per effettuare la verifica della versione.
Nello specifico, il digest MD5 viene calcolato a partire dal "testo del messaggio", ovvero la struttura definita nel file con estensione: ".msg", da quest'ultima vengono poi rimossi gli eventuali commenti, spazi bianchi, ecc..; vengono inoltre riordinate le costanti, anteponendole a tutti gli altri campi.
Poiché i messaggi all'interno di ROS possono essere costituiti da strutture annidate, ovvero ciascuno di essi può essere definito a partire da altri messaggi ROS utilizzati come campi interni, è necessario che la presenza di un'eventuale modifica in uno di questi, venga propagata fino al messaggio corrente, per questo motivo il digest di quest'ultimo viene concatenato con quelli di tutti i sotto-messaggi di cui esso è composto e così via.
In questo modo, anche il minimo cambiamento all'interno di un messaggio ROS, si ripercuoterà in tutti gli altri messaggi che ne fanno utilizzo all'interno della loro struttura, indicando ai vari publisher e subscriber che se i digest non corrispondono, da una parte o dall'altra è stata apportata qualche modifica e perciò, anche se il tipo di messaggio è lo stesso, le versioni utilizzate sono differenti, ciò implica che la comunicazione non può avvenire.

2.4 Elementi principali di un sistema basato su ROS

Di seguito viene fatta un'introduzione agli elementi principali utilizzati da un qualunque sistema basato su ROS, verranno inoltre presentati alcuni strumenti indispensabili che ne consentono l'utilizzo.
L'avvio di tutti questi elementi, indispensabili affinché il sistema possa funzionare, viene riassunto nell'utilizzo di un unico comando chiamato: *"roscore"*, che sarà il primo a dover essere eseguito**.** È necessario che questi processi siano già in esecuzione affinché gli altri nodi del sistema siano in grado di comunicare tra loro, nello specifico il comando roscore permette l'avvio dei processi: Master, Parameter-Server e rosout.

2.4.1 roscore

Master
Il suo scopo è permettere agli altri nodi del sistema di trovarsi a vicenda realizzando poi tra loro la comunicazione effettiva. Come già detto, l'architettura ROS realizza un sistema basato su Publish-Subscribe, ciò prevede che le parti coinvolte nella comunicazione non sappiano a priori a chi dovranno inviare i dati o da chi dovranno riceverli. Nella pratica, ciò che avviene è che ogni nodo del sistema nel momento in cui viene istanziato, necessita di conoscere un unico indirizzo oltre a quello della propria macchina, ovvero quello del Master.
Nel momento in cui un nodo intende pubblicare dei dati sottoforma di messaggi su un dato topic, l'unica operazione che dovrà svolgere sarà comunicarlo al master, che tra le informazioni principali avrà bisogno di conoscere: l'indirizzo e la porta del nodo che effettua la richiesta, il nome del topic e il tipo di messaggio; successivamente, quando un secondo nodo sarà interessato a ricevere i dati pubblicati su quel topic, anche in questo caso, l'unica informazione che dovrà conoscere sarà l'indirizzo e la porta del Master.

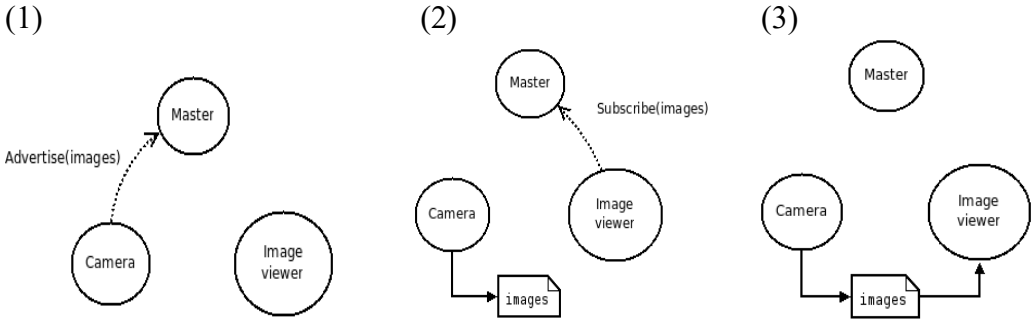

Figura 2 funzionamento master ROS

A questo punto per il Master risulterà abbastanza semplice dato un topic, stabilire se per esso esiste almeno un publisher e un subscriber. Nel caso in cui questa condizione risulti essere verificata, si notifica a ciascun subscriber l'esistenza dei vari publisher in

base al topic di interesse dichiarato in precedenza. A questo punto per ogni coppia di nodi inizierà una fase di negoziazione (avviata dal subscriber), in seguito alla quale sarà possibile instaurare una nuova connessione peer-to-peer.
Il numero di connessioni totali per un dato topic coincide con il numero di publisher moltiplicato per il numero dei subscriber associati ad esso. Qualora il numero di publisher per un dato topic dovesse variare, il master si occuperà di informare i subscriber interessati dell'esistenza di nuove fonti.
Il funzionamento del Master, così come la fase di negoziazione tra i vari nodi, si basa sul protocollo XMLRPC, che a sua volta sfrutta HTTP. Quest'ultimo viene utilizzato principalmente per il suo formato leggero e leggibile più o meno da chiunque, per le connessioni state-less e infine per il fatto che è supportato all'interno della maggior parte dei linguaggi di programmazione.
Nella pratica le interazioni con il Master sono possibili grazie ha due elementi: le "API di registrazione", che consentono di dichiarare: i publisher, i subscriber e i servizi, e il ROS_MASTER_URI memorizzato in una variabile di ambiente, quest'ultima coincide con l'indirizzo e la porta di ascolto del server XMLRPC su cui il Master è in esecuzione.

Parameter-server
Il *parameter-server* è il secondo elemento, avviato contestualmente con l'esecuzione di roscore. Più che un nodo vero e proprio, esso rappresenta un dizionario condiviso, accessibile tramite le API di rete che a loro volta utilizzano il protocollo XMLRPC. Nelle versioni attuali di ROS, il parameter-server viene eseguito all'interno del master. I nodi utilizzano questo server per salvare e all'occorrenza recuperare, determinati parametri a runtime. Sebbene non sia stato progettato per alte performance, risulta essere la soluzione migliore per ricavare statistiche sui nodi e per recuperare dati non binari come i parametri di configurazione.
Il parameter-server deve essere visibile a livello globale, cosicché i vari strumenti di analisi possano facilmente verificare la configurazione e lo stato attuale del sistema, avendo la possibilità di modificarli se necessario. Esiste un comando specifico utilizzabile da terminale chiamato: *rosparam*, esso consente di visualizzare la lista dei parametri disponibili dichiarati dai vari nodi, di ottenere il valore specifico di ciascuno di essi e all'occorrenza di reimpostarlo.

rosout
È l'ultimo elemento avviato durante l'esecuzione del comando roscore, rappresenta un nodo il cui scopo è tener traccia di tutto ciò che viene loggato dagli altri nodi del sistema, attraverso i metodi di *logging* specifici definiti per ROS. Le varie librerie infatti, mettono a disposizione del programmatore delle funzioni per eseguire il *logging*, queste possono essere utilizzate all'interno del codice di un nodo per tener traccia della sua attività.
Nella pratica la chiamata a queste funzioni coincide con la pubblicazione del messaggio di log su un topic che funge da collettore, chiamato: "/rosout" e a cui l'omonimo nodo effettua l'iscrizione come subscriber. Nel momento in cui un qualunque nodo

chiama una funzione di logging, questa automaticamente pubblicherà il messaggio su "/rosout" e a questo punto, l'insieme dei messaggi provenienti dall'intero sistema viene memorizzato in un file di log e contemporaneamente ciascuno di essi viene ripubblicato su secondo topic chiamato: "rosout_agg", che conterrà l'aggregato di tutti i messaggi provenienti dai vari nodi. L'obbiettivo è quello di tener traccia di ciò che avviene all'interno dell'intero sistema, senza la necessità di dover interpellare ogni singolo nodo, ma effettuando un'unica iscrizione a: "rosout", "rosout_agg" o consultando direttamente il file di log.

2.4.2 Comandi utili

roslaunch
È uno strumento che permette di avviare contemporaneamente, utilizzando un solo comando, l'esecuzione di più nodi, sia localmente che da remoto (in questo caso mediante un collegamento SSH), offre la possibilità di effettuare la configurazione dei parametri nel parameter-server e può inoltre includere delle opzioni per riavviare automaticamente dei nodi terminati in seguito a un crash.
Per realizzare quanto descritto, roslaunch sfrutta dei file di configurazione in formato XML, riconoscibili per l'estensione ".launch"; grazie a questi ultimi è possibile di specificare, oltre ai vari parametri, quali nodi avviare, in quale ordine farlo e su quali macchine.
Il comando roslaunch viene eseguito da linea di comando, seguito dal nome del package contenente il file di configurazione, a sua volta seguito dal nome del file in questione e dagli eventuali argomenti utilizzati come parametri. Questo metodo costituisce un'alternativa per l'esecuzione dei nodi rispetto al comando rosrun, con in più una serie di vantaggi come l'avvio di più nodi contemporaneamente, con le possibilità sopra descritte. L'utilizzo di roslaunch comporta inoltre l'esecuzione automatica del comando: roscore, qualora quest'ultimo non fosse già in esecuzione.

rostopic
È un utile strumento da linea di comando che consente sia di ottenere le informazioni riguardanti i vari topic, sia di interagire con essi. Attraverso le varie opzioni che possono seguire questo comando è possibile ad esempio visualizzare la lista di tutti i topic presenti nel sistema ed effettuare delle ricerche applicando dei filtri.
Per il singolo topic è possibile scoprire: il tipo di messaggio per cui è stato definito e chi sono i vari publisher/subscriber grazie all'opzione "info", si possono pubblicare nuovi messaggi su un topic con l'opzione "publish" e visualizzare in tempo reale (iscrivendosi al topic) il loro contenuto utilizzando: "echo". Oltre a queste, che rappresentano le opzioni principali, è possibile misurare anche la frequenza con la quale vengono pubblicati i messaggi, stimare quanta banda mediamente occupano e così via.

rosmsg e rossrv
sono strumenti simili a rostopic, consentono di ottenere informazioni relative a un determinato tipo di messaggio o all'organizzazione di un servizio. Tra le informazioni disponibili, è possibile ottenere: la struttura dettagliata con i vari campi che compongono il messaggio, il package a cui esso appartiene e il digest md5 calcolato a partire dal message-type.

rosservice
Anch'esso è strumento da linea di comando e viene utilizzato per interagire con i ROS-Service. Analogamente a rostopic, consente di ottenere la lista dei servizi disponibili, effettuare delle ricerche specificando dei filtri, ottenere informazioni dettagliate su uno specifico servizio (tipo di servizio, nodo che lo offre, uri, ecc..) e infine, utilizzando l'opzione: "call", seguita dalla richiesta nel formato previsto, è possibile utilizzare il servizio stesso.

rosbag
comprende un insieme di strumenti per la registrazione dei messaggi, che hanno come scopo principale quello di poter essere utilizzati per l'analisi di ciò che avviene all'interno del sistema, avendo la possibilità di riprodurre in un secondo momento una determinata situazione, attraverso la ritrasmissione degli stessi messaggi registrati in precedenza, esattamente con le stesse tempistiche; si evita inoltre la de-serializzazione e ri-serializzazione dei messaggi, garantendo alte performance.
I dati con cui interagisce il comando rosbag vengo salvati e letti utilizzando dei file chiamati: bagfile, caratterizzati dall'estensione ".bag". Questi ultimi ricoprono un ruolo importante all'interno di ROS e la maggior parte delle librerie client offrono delle funzioni per: salvare, leggere ed analizzare questi file.

2.5 Rqt_graph, software Gazebo e simulazione 3D

rqt_graph è uno strumento che consente di visualizzare sotto forma di grafo orientato la struttura del sistema ROS che si sta utilizzando. I vari nodi sono collegati ai topic mediante delle frecce, queste ultime a seconda che siano entranti o uscenti da un blocco rappresentante il topic, indicano se il nodo a cui si riferiscono è un publisher o un subscriber; è possibile arricchire la visualizzazione con informazioni statistiche ad esempio la frequenza media di pubblicazione e altri dati sul flusso dei messaggi.
Il software Gazebo è un simulatore grafico orientato alla robotica, è stato successivamente introdotto all'interno dell'ambiente ROS con lo sviluppo di package appositi che consentono la visualizzazione e l'utilizzo di modelli robotici all'interno di una simulazione 3D, con lo scopo di farli poi interagire con gli altri nodi presenti all'interno del sistema.

Grazie al simulatore è possibile pilotare un drone all'interno dello spazio virtuale, acquisire immagini con le eventuali fotocamere messe a bardo, verificare l'acquisizione corretta dei dati provenienti dai sensori e così via.

2.6 Organizzazione dell'ambiente di lavoro

Per poter lavorare con ROS è necessario installarlo utilizzando il repository ufficiale, sono disponibili diverse opzioni di installazione a seconda che si voglia utilizzare la versione completa oppure delle varianti più leggere.
A partire dal 2010, anno in cui è nato, si sono susseguite diverse distribuzioni di ROS, quella attuale: "Melodic", è la dodicesima. Tuttavia, per motivi dovuti principalmente: alla stabilità, ai package e ai sistemi operativi supportati, alla documentazione disponibile e alla retrocompatibilità; durante la successiva fase di sviluppo legata a questa attività di tesi, si è scelto di utilizzare la distribuzione: "Kinetic", ufficialmente supportata fino al 2021.

2.6.1 Catkin

Catkin è il sistema di build ufficiale di ROS, necessario per la compilazione e il linking delle varie parti del codice di cui è composto.
Il motivo per cui nell'ambito di ROS si è scelto utilizzare un sistema di build personalizzato è dovuto alla necessità di dover gestire un sistema molto più ampio ed eterogeneo rispetto a quelli tradizionali, dove convivono linguaggi di programmazione e sistemi diversi tra loro, ciascuno con regole di compilazione e dipendenze specifiche, che risulterebbero difficili da gestire utilizzando altri strumenti.
Catkin sfrutta gli script Python e delle macro definite da CMake, software da cui esso deriva, per fornire alcune funzionalità che estendono il normale funzionamento di quest'ultimo. Lo scopo era realizzare uno strumento che fosse più compatto rispetto al suo predecessore: *rosbuild*; sia in termini di struttura che di utilizzo, permettendo una migliore organizzazione dei progetti oltre a un miglior supporto al cross-compiling e alla portabilità del codice.
Come già detto, il funzionamento di Catkin è abbastanza simile a quello di CMake, a quest'ultimo è stato aggiunto il supporto per l'identificazione automatica dei package presenti all'interno dello spazio di lavoro e la compilazione di più progetti con dipendenze reciproche allo stesso tempo.
In generale, un sistema di build è responsabile della generazione dei cosiddetti "target", creati a partire dal codice sorgente. Questi ultimi saranno poi utilizzabili direttamente da un qualsiasi utente finale. I target possono essere: librerie, eseguibili, script, file di header o qualsiasi altra cosa che non sia codice statico. Come già detto, nella terminologia ROS, il codice sorgente viene organizzato all'interno dei "package" e ognuno di essi tipicamente consiste in uno o più target che dovranno essere creati

duranti il processo di build. L'insieme dei package che Catkin considera contemporaneamente durante questo processo sono associati a uno stesso *workspace*.
Catkin permette quindi di generare contemporaneamente più target appartenenti a package diversi, rispettando le dipendenze reciproche. Per riuscire a far questo, come ogni altro sistema di build deve conoscere una serie di informazioni, tra le quali: la posizione degli strumenti di compilazione (ad esempio del compilatore C++), la posizione all'interno del file-system del codice sorgente, quali sono le varie dipendenze e dove sono collocate; inoltre, è necessario specificare dove andranno generati gli eseguibili e dove dovranno essere installati.
Queste informazioni sono tipicamente contenute in un qualche file di configurazione che dovrà essere leggibile dal sistema di build, nell'ambito di CMake e di conseguenza anche di Catkin, queste informazioni vengono specificate all'interno di file chiamati: "CMakeList.txt". Oltre a un file di questo tipo definito a livello di workspace, ogni package ha un proprio file di configurazione con lo stesso nome, collocato all'interno della propria root-directory.

2.6.2 Creazione di un nuovo package

Per creare un nuovo package è possibile utilizzare il comando catkin_create_pkg, seguito dal nome che si vuole assegnare e dalle dipendenze necessarie. Verranno generati automaticamente tutti i file e le cartelle richieste, compreso CMakelist.txt. Quest'ultimo conterrà una configurazione di base, che andrà poi personalizzata in base all'utilizzo.
Oltre a questo file, durante la fase creazione del package viene generato, sempre all'interno della root-directory, un manifest chiamato: "package.xml", dove vengono descritte tutte le proprietà riguardanti il package tra cui: il nome, la versione, l'autore e le dipendenze rispetto agli altri package; se queste dovessero essere incomplete o sbagliate, il programma potrebbe comunque essere compilato sulla propria macchina, a condizione che si abbiano tutti i software necessari anche se non dichiarati, ma in generale non potrà funzionare correttamente altrove, in quanto chiunque dovesse provare ad installare in un secondo momento il package, non avrà modo di sapere di quali altri elementi avrà bisogno per permettergli funzionare.
I vari package, possono contenere il codice per la creazione di nodi, la generazione di messaggi o entrambi; in generale, il codice sorgente relativo al nodo andrà inserito nella cartella src (generata automaticamente), mentre i file relativi ai messaggi e ai servizi (.msg e .srv) andranno inseriti all'interno della cartella msg, che dovrà essere creata manualmente.
Oltre a "msg", è possibile aggiungere manualmente anche altre cartelle tra cui: "launch" (che conterrà i file con l'estensione omonima descritti in precedenza) e "config", che potrà contenere uno o più file in formato ".yaml" per la configurazione dei parametri in ingresso associati ai vari nodi.

2.6.3 Le librerie client

Una libreria client per ROS consiste in un insieme di funzioni che semplificano il lavoro del programmatore e permettono di interfacciarsi con i vari elementi del sistema, rendendoli disponibili attraverso il codice. In generale, attraverso una libreria si ha la possibilità di scrivere i nodi ROS, pubblicare messaggi sui topic o iscriversi ad essi, fornire e chiamare i ROS service, sfruttare il parameter server e molto altro.
Queste librerie possono essere sviluppate in qualsiasi linguaggio di programmazione, anche se attualmente ci si è focalizzati nel fornire un supporto completo a C++ , Python e Lisp.
Sono disponibili, sebbene considerate sperimentali, delle librerie anche in altri linguaggi tra cui Java, quest'ultima ha riscontrato un certo interesse, anche grazie al fatto che offre la possibilità di essere integrata all'interno dei sistemi Android, argomento di cui si discuterà in seguito. In generale, ogni libreria offre delle funzioni che consentono: l'inizializzazione, l'avvio e la verifica di terminazione del nodo. La funzione di inizializzazione (ros::init() in C++ e rospy.init() in Python) consente di definire un nuovo nodo, ne esistono diverse versioni che consento di specificare diversi parametri oltre al nome di quest'ultimo.
Tra i principali, ad esempio, è presente l'attivazione della modalità anonima: si aggiunge automaticamente un numero random al nome del nodo rendendolo unico all'interno del sistema ed evitando collisioni con altri nodi preesistenti. Attraverso un altro parametro si offre la possibilità di ridefinire il gestore per il segnale di sistema: SIGINT. Questo segnale di default provoca la terminazione del processo e perciò ROS utilizza una funzione predefinita che si occupa di gestire l'evento così da avere una chiusura "pulita": eliminando eventuali iscrizioni, pubblicazioni sui topic e collegamenti ai ROS service. Tuttavia, è concesso al programmatore di ridefinire questa funzione, così da poter gestire come meglio crede la terminazione del nodo.
La possibilità di gestire l'avvio di un nodo spesso varia da un linguaggio all'altro, al contrario, quasi tutti definiscono dei metodi per verificare la condizione di terminazione e grazie ad essi si è in grado di regolare il ciclo di vita del nodo. Mentre la logica del generico subscriber tende ad essere sempre più o meno la stessa, ovvero: il nodo, una volta inizializzato, effettua l'iscrizione a un topic passando al metodo "subscribe" una callback, che verrà invocata nel momento in cui arriva un nuovo messaggio, a questo punto, il processo eseguirà la funzione: "spin" entrando in uno stato di attesa indefinito, fino al richiamo della callback.
Una volta terminata l'esecuzione di quest'ultima, se non sono presenti altri messaggi nella coda, riprende la fase di attesa; il ciclo si interrompe con la terminazione del nodo. Il publisher ha invece una logica che può variare a seconda dei casi. Anch'esso può basarsi su un meccanismo asincrono, questo capita ad esempio quando si vogliono raccogliere dei dati da un sensore: nel momento in cui quest'ultimo acquisisce un nuovo dato dovuto alla variazione di una qualche grandezza fisica, viene chiamata una callback e all'interno di quest'ultima si utilizza il metodo "publish" per pubblicare su un topic i nuovi dati. Tuttavia, la logica utilizzata nel publisher può anche essere di

tipo sincrono: si pubblicano nuovi messaggi sul topic con una cadenza fissa, sulla base di un periodo costante.
In quest'ultimo caso è necessario ricorrere a una struttura ciclica, dove le istruzioni vengono eseguite fino alla terminazione del nodo, è in questa fase che diventano utili le funzioni, accennate in precedenza, per poter verificare questa condizione.

Capitolo III

Multimedia in ambito ROS

Le immagini al pari degli altri dati all'interno di un'architettura ROS, vengono gestite all'interno dei singoli nodi in maniera autonoma, esse possono subire una serie di trasformazioni lungo il percorso che le porta dalla sorgente di acquisizione alla loro visualizzazione o memorizzazione. Il passaggio tra i vari moduli avviene anche in questo caso in modo asincrono, attraverso i topic, secondo il paradigma Publish-Subscribe. Si vedrà tuttavia, che a differenza degli altri tipi di dati, le immagini costituiscono un caso particolare. Queste possono infatti essere rappresentate attraverso numerose codifiche e formati, che rendono necessaria l'adozione di meccanismi specifici per consentirne una gestione unificata, flessibile e adeguata.
Parlando di gestione delle immagini, è possibile dividere concettualmente questa operazione in quattro fasi: acquisizione, elaborazione, trasmissione e visualizzazione.
Sebbene ROS non si occupi direttamente di definire un meccanismo standard per l'acquisizione, utilizza vari package che sono in grado di comunicare direttamente con i driver e di regolarne i parametri, al fine di acquisire i dati grezzi, che dovranno poi essere processati in un secondo momento. Le fasi sopra elencate vengono svolte attraverso l'utilizzo di numerosi package che a loro volta vengono raggruppati in stack, tra questi i principali sono: image_pipeline e image_common.

image_pipeline
Questo stack definisce al suo interno i vari package responsabili dell'acquisizione delle immagini dai dispositivi, della fase di elaborazione e infine della loro visualizzazione ad alto livello. Come suggerisce il nome, i moduli sono pensati per essere disposti in cascata e ciascuna fase, benché autonoma, presuppone il corretto completamento delle precedenti. Lo scopo dei vari package presenti in image_pipeline, è quello di gestire il processamento delle immagini grezze acquisite dalle varie fotocamere, attraverso l'interazione diretta con i driver, con l'obbiettivo di trasformarle in dati utilizzabili come input dagli algoritmi definiti nei package di visualizzazione.

image_common
anche image_common come il precedente include alcuni package per gestire i parametri relativi alla configurazione e in particolare alla calibrazione delle fotocamere, come: "camera_calibration_parser" e "camera_info_manager". Tuttavia, il package più importante definito al suo interno è "image_transport", quest'ultimo regola i vari meccanismi utilizzati per la trasmissione delle immagini nell'ambiente ROS, definendo nuove classi publisher e subscriber specializzate e fornisce un supporto flessibile per il trasporto di immagini in formati compressi. Si vedrà in seguito perché il suo utilizzo sia fortemente consigliato ogniqualvolta sia debbano pubblicare e/o ricevere le immagini attraverso i topic.

3.1 Acquisizione delle immagini

Nel momento in cui si decide di aggiungere una nuova sorgente in grado di acquisire immagini, all'interno di un'architettura ROS, è necessario fare in modo che i dati grezzi ottenuti dal driver, che comunica direttamente con il sensore della fotocamera digitale, vengano inseriti all'interno di una struttura il cui formato risulti adatto per l'utilizzo all'interno di ROS. È perciò necessario definire un nuovo tipo di messaggio che tenga conto di quelle che sono le informazioni indispensabili, al fine di poter interpretare correttamente un'immagine. Per questo motivo è stato definito il tipo di messaggio: "sensor_msgs/Image", che presenta il seguente formato:

std_msgs/Header header
uint32 height
uint32 width
string encoding
uint8 is_bigendian
uint32 step
uint8[] data

Come detto nel capitolo precedente, il campo "header" è comune a molti messaggi ROS, al suo interno contiene un sequence-number, un timestamp e un frameID; che potranno essere utilizzati nel modo più opportuno a seconda dei casi.
Per poter essere in grado di interpretare correttamente le informazioni associate a un'immagine è indispensabile innanzitutto conoscere le sue dimensioni, intese come numero righe e colonne della matrice bidimensionale contenente i dati. Questi valori sono memorizzati all'interno dei campi: "width" e "height".
Un altro elemento fondamentale è il campo "encoding". Infatti, al suo interno viene inserito il nome della codifica utilizzata per l'immagine corrente, questo può variare a seconda del contesto e determinerà il valore del campo "step". Quest'ultimo rappresenta la reale dimensione di ciascuna riga della matrice 2D associata all'immagine, il suo valore corrisponde al numero di byte necessari per codificare ogni singola cella (che varieranno appunto in base alla codifica utilizzata), moltiplicato per il numero totale di celle di ogni riga (ovvero il valore del campo width).
"is_bigendian" è semplicemente un flag necessario per stabilire quale convenzione sia stata utilizzata per il byte-ordering. Infine, il campo "data" è un array di byte, che contiene i dati effettivi dell'immagine, questi sono organizzati secondo una matrice di dimensioni: height * step.
A questo punto, nel momento in cui si intende utilizzare un driver che sia in grado di interfacciarsi con l'ambiente ROS, occorre realizzare delle funzioni che siano in grado di riempire i campi del messaggio appena descritto.
Tuttavia, è necessario notare, che nel momento in cui il driver di una fotocamera digitale acquisisce i dati relativi a una nuova cattura, generalmente questi non sono ancora pronti per essere visualizzati. Ciò è dovuto a svariati elementi di distorsione e all'utilizzo di particolari tipi di codifica, vincolati alla struttura dei singoli sensori.

Per tale motivo, le cosiddette “immagini grezze”, ovvero quelle contenute all’interno dei messaggi sensor_msgs/Image, riempiti con i dati provenienti direttamente dai driver, necessitano di una successiva fase di elaborazione.

3.2 Elaborazione delle immagini

All’interno dello stack image_pipeline, sono presenti numerosi package per il processamento delle immagini e ognuno di essi è specializzato in una particolare operazione:

• Calibrazione: le fotocamere devono essere calibrate in modo da correlare le immagini prodotte con il mondo tridimensionale. Il package camera_calibration fornisce gli strumenti per calibrare le camere monoculari e stereoscopiche all’interno di un sistema ROS.

• Elaborazione monoculare: il flusso di immagini grezze può essere incanalato attraverso un nodo image_proc per rimuovere la distorsione introdotta dalla fotocamera. Il nodo esegue anche un’interpolazione del colore per le fotocamere che utilizzano lo schema Bayer.

• Elaborazione stereo: un nodo stereo_proc esegue le stesse operazioni svolte da image_proc, ma per una coppia di camere co-calibrate per la visione stereoscopica. Utilizza l’elaborazione stereoscopica per la produzione di immagini stereo e nuvole di punti.

• Depth processing: depth_image_proc fornisce le componenti per il processamento delle immagini con l’informazione di profondità (prodotte ad esempio da Kinect, fotocamere a tempo di volo, ecc.), ad esempio per la generazione di nuvole di punti.

3.2.1 image_proc

Questo package definisce il codice per un generico nodo image_proc, quest’ultimo si colloca tra il driver della camera e i nodi di visualizzazione. image_proc, si occupa della rettificazione singola dell’immagine e dell’elaborazione del colore, perciò ha il compito di rimuovere la distorsione presente nel flusso di immagini grezze e quando necessario, di convertire i dati di un’immagine che utilizza lo schema Bayer, nella corrispondete forma RGB o YUV, attraverso appositi algoritmi di demosaicizzazione. Inoltre, image_proc quando è utilizzato congiuntamente ad alcuni dei suoi nodelet, è anche in grado di applicare via software la decimazione è il ROI (region of interest) dell’immagine grezza.

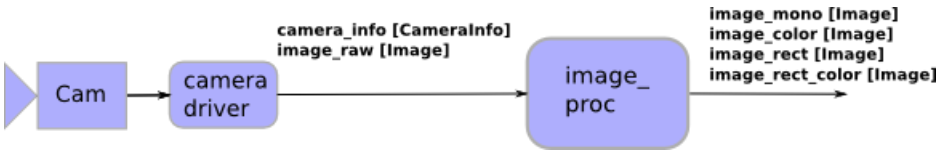

Figura 3 pipeline di acquisizione ed elaborazione [21]

L'intera fase di processamento è gestita on-demand. L'elaborazione del colore così come la rettificazione delle immagini, vengono eseguite solamente se è presente almeno un subscriber per il topic loro associato, mentre durante il periodo in cui non ci sono nodi iscritti quest'ultimo, image_proc elimina a sua volta la propria sottoscrizione ai relativi topic image_raw e camera_info, evitando di processare inutilmente i dati.

Nodelet:
Analogamente al concetto di applet [1], inteso genericamente come un programma che viene eseguito come "ospite" nel contesto di un altro programma detto: "*container*"; all'interno dell'ambiente ROS, i nodelet sono stati introdotti per consentire l'esecuzione del codice relativo a nodi distinti, all'interno di un unico processo.
I nodelet infatti, consentono il caricamento dinamico di più classi all'interno di uno stesso nodo, per ognuna di esse è fornito semplicemente un nuovo namespace. Questo è sufficiente per far sembrare che ciascun nodelet agisca come un nodo separato, nonostante esso appartenga allo stesso processo; questa operazione porta con sé una serie di vantaggi.
Tra questi il principale consiste nella possibilità di evitare la copia fisica dei messaggi, nel momento in cui questi vengano scambiati tra nodi in esecuzione sulla stessa macchina. Infatti, sarà sufficiente il passaggio del puntatore durante la chiamata dei metodi publish e subscribe. Ovviamente questo non sarebbe possibile nel caso classico in cui i nodi sono associati a processi distinti, poiché ciascuno di essi avrebbe un proprio spazio di indirizzamento, che normalmente non è accessibile dall'esterno.
La possibilità di evitare la copia dei messaggi durante lo scambio tra i nodi, consente di aumentare notevolmente il throughput tra essi, ciò risulta particolarmente utile in quei casi in cui gli scambi sono frequenti e le dimensioni del messaggio sono elevate, questo è il caso di image_proc.
Per questo motivo, all'interno del package sono stati definiti alcuni nodelet che si occupano eseguire quelle fasi di debayering e rettificazione introdotte in precedenza e che concettualmente dovrebbero appartenere a nodi distinti, con l'obbiettivo finale di velocizzare l'elaborazione complessiva delle immagini.
Di seguito verranno presentati i vari nodelet definiti per image_proc:

• **image_proc/debayer:** questo nodelet definisce un algoritmo di demosaicizzazione, che ha lo scopo di ricostruire la rappresentazione a colori di un'immagine, ad esempio nel formato RGB, partendo dai dati grezzi ottenuti dal sensore di una fotocamera

[1] Applet – Wikipedia: https://it.wikipedia.org/wiki/Applet

digitale che utilizza un CFA (*color filter array)* con schema Bayer. I dati vengono poi inseriti all'interno di messaggi ROS e pubblicati nelle due versioni: monocromatica e a colori.

• **image_proc/rectify:** accetta in input un flusso di immagini non rettificate insieme ai relativi parametri di regolazione e genera in uscita il corrispondente flusso di immagini rettificate.

• **image_proc/crop_decimate:** questo nodelet ha il compito di applicare l'algoritmo di decimazione (software binning) e il ROI alle immagini in input. Inoltre, consente di ri-mappare i namespace: camera e camera_out nei namespace di input/output desiderati.

• **image_proc/resize:** consente di ridimensionare le immagini in input e di pubblicarle nel nuovo formato e/o modificare i parametri relativi alla dimensione associati alla fotocamera.

Figura 4 immagine prima e dopo rettificazione e debayering [21]

Durante la normale esecuzione di image_proc vengono caricati di default: un nodelet image_proc/debayer e due image_proc/rectification, ma è possibile variare il comportamento del nodo, decidendo di volta in volta i singoli nodelet da caricare.

image_proc.launch
Utilizzando questo file ".launch" è possibile avviare i nodelet di debayering e di rettificazione per una specifica camera, all'interno di un gestore di nodelet fornito dall'utente, ciò equivale all'esecuzione mediante rosrun di un normale nodo image_proc.

stereo_proc
Il package stereo_proc ha le stesse caratteristiche già citate per image_proc, con la differenza che invece di trattare immagini singole, lavora su una coppia di immagini stereoscopiche.

3.2.2 cv_bridge

cv_bridge insieme a image_geometry e opencv_tests appartiene allo stack: vision_opencv. Lo scopo di questi package è permettere la conversione reciproca tra i messaggi ROS associati alle immagini e le corrispondenti matrici di pixel utilizzate all'interno della libreria OpenCV.

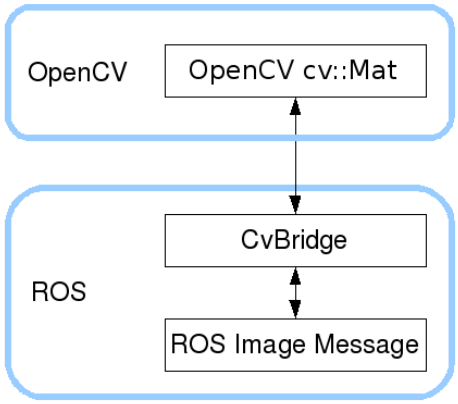

Figura 5 schema concettuale cv_bridge [22]

Come già visto, ROS fornisce gli strumenti utili per il processamento delle immagini direttamente nel formato: sensor_msgs/Image. Tuttavia, molti utenti hanno espresso la necessità di poter svolgere un'elaborazione avanzata tramite l'utilizzo di OpenCV. Per questo motivo è nato il package "cv_bridge", che definisce una libreria (CvBridge) in grado di fornire un'interfaccia tra i messaggi ROS e le classi definite all'interno di OpenCV.
Grazie ad essa, i messaggi di tipo *sensor_msgs/Image* posso essere convertiti nel formato OpenCV::Mat e viceversa questi ultimi possono essere riconvertiti a loro volta in messaggi ROS ed essere ripubblicati.
CvBridge definisce il tipo CvImage, al cui interno è presente un buffer, che conterrà i singoli byte dell'immagine trasportata. Inoltre, sono presenti degli appositi campi che consentono di memorizzare la codifica e l'header del messaggio ROS; questi saranno utili durante la fase di riconversione. CvImage infatti, contiene esattamente le stesse informazioni presenti all'interno del messaggio sensor_msgs/Image, in questo modo è sempre possibile passare da una rappresentazione all'altra senza che ci siano perdite.
Durante la conversione di un messaggio ROS in un CvImage, CvBridge distingue due diversi casi, ovvero: quando i dati possono essere utilizzati in loco e quando invece è necessario fare una copia fisica degli stessi.
In alcuni casi, è possibile condividere in modo sicuro i dati contenuti nel messaggio senza che sia necessario copiarli. In ogni caso l'input utilizzato per la conversione è il puntatore al messaggio, oltre al tipo di codifica, che potrà essere specificato come argomento opzionale; è da notare che quest'ultimo si riferisce all'oggetto CvImage di

destinazione. A questo punto il passaggio dalla rappresentazione ROS a quella OpenCV può avvenire attraverso due funzioni differenti.
La funzione "toCvCopy" effettua una copia dei dati associati all'immagine contenuta nel messaggio ROS, allocando un nuovo buffer che risulta accessibile attraverso un puntatore "cv_ptr", appartenete al nuovo oggetto CvImage. Inoltre, se necessario, si esegue la conversione tra il formato specificato nel campo "encoding" del messaggio e quello di destinazione relativo al dominio OpenCV.
La funzione "toCvShare" invece, restituisce un oggetto cv::Mat che punta direttamente alla zona di memoria associata al messaggio ROS, evitando la copia fisica. Questo ovviamente accade, se le codifiche sorgente e destinazione coincidono, in caso contrario, sarà necessario allocare un nuovo buffer dove verrà eseguita la conversione, perciò il comportamento sarà identico a quello del metodo toCvCopy. Durante il periodo in cui si utilizza l'oggetto CvImage ottenuto mediante il passaggio del puntatore, il corrispondente messaggio ROS non sarà accessibile e i dati relativi in memoria non potranno essere liberati. Occorre inoltre specificare che non è permesso modificare un oggetto CvImage ottenuto attraverso toCvShare, in quanto quest'ultimo, a seconda dei casi, potrebbe condividere i dati con il messaggio ROS, che a loro volta potrebbero essere condivisi con altre callback.
Nel caso in cui non venga specificata la codifica per l'immagine OpenCV di destinazione verrà sempre utilizzata quella indicata nel campo "encoding" del messaggio. In questo caso la funzione toCvShare garantisce di non effettuare mai la copia fisica dei dati.
Per le codifiche più comuni, in modo facoltativo, CvBridge è in grado di eseguire automaticamente le conversioni di profondità di colore o pixel, se necessario.
In generale, mono8 e bgr8 sono le due codifiche più utilizzate dalla maggior parte delle funzioni della libreria OpenCV. CvBridge è anche in grado di riconoscere le codifiche con schema Bayer caratterizzate dal tipo 8UC1 (8-bit unsigned, one channel), ma non effettuerà la conversione da e verso questo formato in quanto, nel tipico sistema ROS, questa ultima dovrebbe essere già stata fatta a monte dal nodo image_proc e ci si aspetta che le immagini arrivino dopo aver superato la fase di debayering.
Per riconvertire un'immagine OpenCV in un messaggio ROS, è necessario utilizzare la funzione "toImageMsg", ovviamente se l'immagine OpenCV non è stata ottenuta a partire da un messaggio ROS, è necessario inserire manualmente i valori associati ai campi "header" ed "encoding".

3.3 Trasmissione delle immagini

3.3.1 image_transport

Nella parte precedente sono stati introdotti i principali nodi utilizzati per l'elaborazione delle immagini. Questa parte verrà dedicata ai principali meccanismi utilizzati per la loro trasmissione.

A differenza degli altri nodi publisher e subscriber, associati a dati e formati specifici, quelli relativi alle immagini necessitano di un approccio differente per quanto riguarda i meccanismi di trasmissione.
Quando si lavora con le immagini, spesso si tende a voler specializzare le strategie di trasporto, ad esempio con l'utilizzo di algoritmi di compressione o l'impiego dei vari codec video. Un'immagine infatti, può essere rappresentata attraverso diversi formati e codifiche, è possibile comprimerla o includerla all'interno di uno streaming video basato sulla codifica differenziale. Ad ognuna di queste possibilità, spesso è associata tutta una serie di parametri specifici, che necessitano di essere regolati per ottenere un funzionamento accettabile. Inoltre, i messaggi utilizzati per le immagini, rispetto alla stragrande maggioranza degli altri tipi di messaggio, hanno una dimensione di gran lunga superiore, ciò rende necessari alcuni accorgimenti e ottimizzazioni, che hanno a che fare con la qualità del servizio e con la banda occupata durante la trasmissione, al fine di garantire un corretto funzionamento delle applicazioni.
L'ampia scelta di formati e codifiche porta con sé tutta una serie di complicazioni, in quanto l'introduzione di un nuovo tipo di trasporto, che si propone di essere largamente utilizzato, comporterebbe la necessità di dover ridefinire tutti i nodi preesistenti al fine di supportarlo. Questa soluzione è apparsa fin da subito poco flessibile.
Per far fronte al problema, ovvero evitare di non dover ridefinire ogni volta classi specializzate per ogni singolo formato e al contempo offrire la possibilità di apportare delle ottimizzazioni nella trasmissione dei flussi di immagini, è stato introdotto un package chiamato: "image_transport".
L'obbiettivo principale di image_transport vuole essere l'eliminazione della correlazione tra utilizzo del dato, ovvero l'immagine, e la sua trasmissione. Si parte dall'assunto che la maggior parte dei nodi, siano essi produttori o consumatori di immagini (come ad esempio quelli definiti nella parte precedente), effettuano le loro operazioni sulle immagini in quanto tali, quindi senza compressione, e nel fare ciò utilizzano dei semplici messaggi: sensor_msgs/Image.
Ovviamente, sebbene questo formato risulti adatto in fase di elaborazione, non è altrettanto efficiente durante la trasmissione, sia essa una copia fisica sulla stessa macchina o un inoltro attraverso la rete. A questo punto la soluzione è stata quella di introdurre un nuovo livello, ovvero quello rappresentato da image_transport. Quest'ultimo si occuperà di volta in volta delle conversioni dei messaggi da e verso il tipo sensor_msgs/Image, fornendo un supporto trasparente e permettendo il trasporto con bassa banda e a bassa latenza, grazie all'utilizzo dei formati compressi. In questo modo i vari nodi non devono più occuparsi di quale formato debbano avere le immagini per poter essere elaborate e/o trasmesse; mentre chi definisce il nuovo tipo di trasporto dovrà semplicemente creare una singola coppia di classi publisher e subscriber che dovranno solamente effettuare una conversione dal proprio formato verso il tipo sensor_msgs/Image e viceversa, il tutto in un'ottica di maggiore flessibilità.
Alla luce di quanto detto, appare chiaro il motivo per cui il package image_transport dovrebbe sempre essere utilizzato per pubblicare e ricevere le immagini all'interno di un'architettura ROS. A questo punto resta da chiarire come sia possibile aggiungere un nuovo tipo di trasporto. Per fare ciò, si ricorre all'utilizzo di particolari package che

hanno la funzione di plugin. Infatti, image_transport di base fornisce solamente il tipo di trasporto cosiddetto: "raw", ovvero non compresso, con l'intento di non imporre dipendenze esterne che non sono strettamente necessarie per la visualizzazione delle immagini, nella sua versione base infatti esso non applica nessuna conversione al formato delle immagini, ma semplicemente le pubblica cosi come sono.
Nella pratica, image_transport definisce i punti di aggancio, necessari per consentire il passaggio dell'immagine sensor_msgs/Image dal nodo in cui viene utilizzata alla specifica classe di un determinato plugin package, questa si occuperà della conversione tra i formati e dopodiché passerà il risultato a una callback a sua volta ricevuta come parametro. Sarà all'interno di quest'ultima che ci si occuperà delle fasi successive, ovvero la pubblicazione nel caso del publisher e l'elaborazione nel caso del subscriber. Perciò di base image_transport non definisce nessun formato aggiuntivo, gli eventuali altri tipi di trasporto saranno disponibili solamente se verranno inclusi arbitrariamente nel proprio sistema. Su Ubuntu, la distribuzione base di ROS include di default uno stack di plug-in per image_transport chiamato: "image_transport_plugins", di cui si parlerà in seguito, all'interno di quest'ultimo sono definiti i tipi di trasporto "compressed" e "theora" .

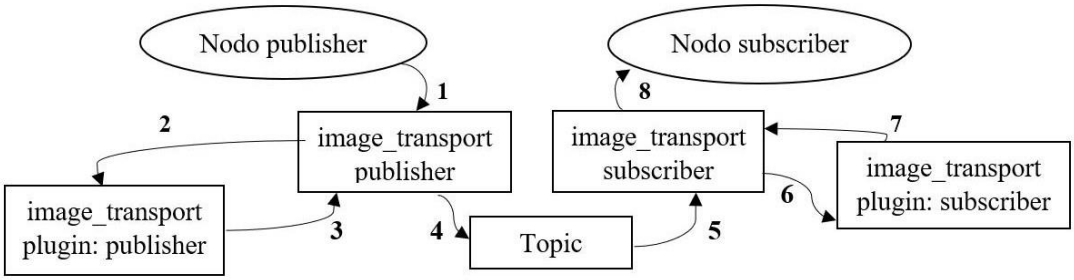

Figura 6 schema di funzionamento image_transport

image_transport: publisher

Sebbene i publisher image_transport vengano utilizzati come gli altri publisher ROS, essi possono offrire una certa varietà di opzioni specializzate per il trasporto multimediale (compressione JPEG, streaming video, ecc.). Detto ciò, appare chiaro come image_transport offra intrinsecamente un altro vantaggio: i vari subscriber possono richiedere allo stesso nodo di utilizzare tipi di trasporto differenti a seconda dell'occorrenza e in maniera dinamica.
Infatti, i publisher definiti con image_transport, rispetto al comportamento classico degli altri nodi, annunciano tanti topic quanti sono i tipi di trasporto al momento disponibili nel sistema. Il nome di questi topic segue una convenzione standard che verrà descritta in seguito. Durante l'utilizzo si può notare come le varie le interfacce che interagiscono con questi topic, non si riferiscano mai al nome completo, associato a un dato tipo di trasporto, ma utilizzino sempre il nome di base del topic, ovvero quello specificato nel nodo di elaborazione.
Riassumendo, i messaggi di tipo sensor_msgs/Image, vengono pubblicati sul topic base, utilizzando un semplice publisher che non fa nulla di particolare. Nel momento

in cui all'interno del sistema sono presenti dei plugin per image_transport, questi annunceranno a loro volta dei topic derivati da quello base, aggiungendo convenzionalmente al nome di quest'ultimo, il nome del tipo di trasporto per cui sono stati definiti. Ad esempio, nel caso dei plugin "compressed" e "theora", se si considera un topic che trasporta immagini non compresse come: "/usb_cam/image_raw", i due topic da esso derivati si chiameranno rispettivamente: "/usb_cam/image_raw/compressed" e "/usb_cam/image_raw/theora".
I publisher image_transport non hanno parametri propri, ma i vari plugin sono fatti in modo da permettere l'utilizzo di impostazioni specifiche attraverso il parameter server. Questo consente di interagire con le opzioni di configurazione, come: il bit-rate, il livello di compressione, ecc. Questi plugin offrono la possibilità di configurare lato publisher il tipo di codifica utilizzata, adattandola alle diverse necessità che si possono avere lato client. Occorre perciò focalizzarsi sul namespace pubblico, definito dal topic di base, piuttosto che sul namespace privato relativo al publisher che sta pubblicando. È importante precisare che questi parametri rappresentano delle risorse condivise e in quanto tali influiscono sul comportamento osservato da tutti i subscriber che sono iscritti al topic. I vari parametri dovrebbero sempre essere resi disponibili attraverso il formato definito nel messaggio ROS: "dynamic_reconfigure", per garantire una migliore visibilità e un più facile utilizzo, ad esempio nel seguente modo:

$ rosparam /camera/image/compressed/jpeg_quality (int, default: 80)

Questo comando consente di modificare la qualità delle immagini in uscita ed è associato l'algoritmo JPEG. "*jpeg_quality"* è un parametro specifico del tipo di trasporto: "compressed" e in questo caso si riferisce al topic: "camera/image".

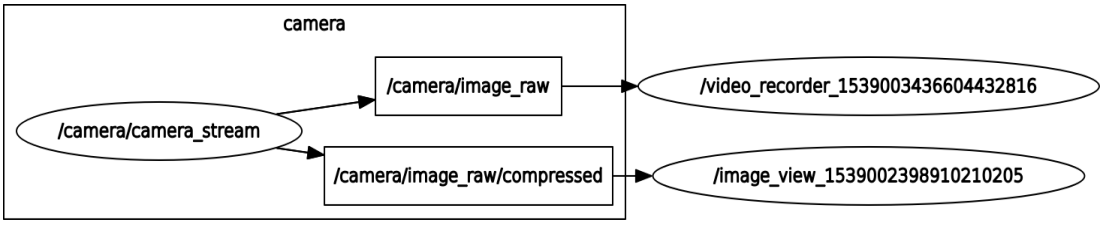

Figura 7 esempio topic image_transport

image_transport: subscriber

Per quanto riguarda i subscriber associati alla ricezione delle immagini, anche questi possono essere utilizzati nel modo classico, ma spesso come accade per i publisher, vengono impiegati congiuntamente a plugin specializzati per un dato tipo di trasporto. L'istanza di un subscriber viene creata utilizzando il nome del topic di base e specificando il tipo di trasporto che si intende utilizzare. All'interno delle librerie ROS, per specificare il tipo di trasporto si ricorre ai cosiddetti: "image_transport::TransportHints", è importante specificare che questi ultimi non coincidono con quelli illustrati

nel capitolo precedente a proposito dei protocolli di trasporto, infatti, sebbene il loro scopo è analogo ovvero indicare una preferenza, questi si riferiscono a parametri differenti e vengano utilizzati in un contesto diverso.
I TransportHints relativi alle immagini possono essere utilizzati per specificare un namespace differente per i parametri. Ciò è utile per ri-mappare image_transport verso un altro namespace e consentire tipi di trasporto diversi per le diverse richieste di iscrizione verso i topic forniti da un unico publisher. Il nodo subscriber può inoltre specificare il nome di un parametro diverso dal tipo di trasporto, sebbene questo venga scoraggiato in favore di una maggiore coerenza. I nodi che effettuano l'iscrizione ai topic associati alle immagini dovrebbero riportare quali sono i parametri per il controllo del trasporto, specialmente se diversi da image_transport.
Le classi subscriber definite nei plugin sono autorizzate a utilizzare il parameter-server per le opzioni di configurazione, ad esempio per impostare il livello di post-elaborazione video. Questi parametri influiscono sul comportamento di un singolo subscriber e quindi sul modo in cui i dati ricevuti vengono interpretati (decodificati). Questo differisce dai parametri utilizzati dai publisher, che come detto in precedenza, rappresentano una risorsa condivisa e perciò un'eventuale modifica influisce sui dati inviati a tutti coloro che sono iscritti al topic. Il namespace utilizzato per identificare i parametri nel parameter-server viene nuovamente specificato tramite gli image_transport:: TransportHints, come scelta di default nel namespace privato del nodo subscriber.
Anche i parametri utilizzati dai subscriber andrebbero esposti attraverso l'utilizzo del message-package: "dynamic_reconfigure", per una migliore visibilità e facilità d'uso. Ad esempio:

$ rosparam theora/post_processing_level (int, default: 0)

All'interno del package image_transport oltre alle classi publisher e subscriber appena descritte e utilizzate dagli altri nodi, sono presenti anche i nodi republish e list_transport:

republish
è nodo che consente di iscriversi a un topic su cui vengono pubblicate delle immagini con uno specifico tipo di trasporto e grazie alle classi publisher e subscriber definite all'interno di image_transport, le ripubblica su un secondo topic con un formato diverso. Normalmente permette di utilizzare tutti i plugin di trasporto disponibili nel sistema; è possibile specificare un determinato tipo di trasporto in uscita inserendolo come parametro nel comando di avvio del nodo (di default viene utilizzato il formato "raw").

list_transports
Consente di ottenere l'elenco di tutti i tipi di trasporto definiti per image_transport, attualmente installati nel sistema. Inoltre, tenta di determinare se questi siano o meno utilizzabili, ad esempio: verifica che tutti i package necessari (specificati tra le varie dipendenze) siano stati compilati correttamente, si accerta che i vari plugin siano in

grado di essere caricati in modo appropriato (provando a istanziarne le classi), ecc.. Infine, insieme alla lista vengono mostrati: il tipo di messaggio da utilizzare per ogni tipo di trasporto e tutta una serie di informazioni utili per il suo impiego.

3.3.2 image_transport_plugins

Come anticipato, image_transport_plugins è insieme di package, raggruppati all'interno di un unico stack, utilizzati per la pubblicazione e l'iscrizione a topic associati a messaggi di tipo sensor_msgs/Image, con lo scopo di fornire delle alternative più efficienti rispetto al classico invio di immagini non compresse. I vantaggi sono molteplici, ad esempio, per visualizzare un flusso di immagini provenienti da un robot, un codec video è in grado di garantire una banda e una latenza notevolmente inferiori. Rispetto al formato raw, per garantire una trasmissione con un buon frame-rate e una definizione più alta, sarebbe preferibile l'utilizzo di formati come JPEG o PNG.
Come anticipato in precedenza, image_transport_plugins definisce due tipi di trasporto aggiuntivi chiamati: "compressed" e "theora".

Il trasporto "compressed", definito all'interno del package:" compressed_image_transport", fornisce dei metodi per la compressione JPEG e PNG, molto utili per quelle sorgenti che devono essere caratterizzate da un frame-rate basso e spesso discontinuo o in quelle situazioni in cui è richiesta una compressione senza perdite (PNG); consente ai vari nodi, utilizzando le classi fornite da image_transport, di pubblicare e iscriversi ai topic che utilizzano dei messaggi di tipo sensor_msgs/CompressedImage, i quali hanno la seguente struttura:

std_msgs/Header header
string format
uint8[] data

Rispetto al messaggio sensor_msgs/Image, quest'ultimo contiene meno campi, infatti durante le fasi di compressione e decompressione dell'immagine, memorizzata all'interno del campo "data", l'unica informazione che è necessario conoscere è il nome dell'algoritmo da utilizzare, quest'ultimo è memorizzato nel campo "format".
Il formato e la qualità delle immagini possono essere cambiati durante l'esecuzione, utilizzando i parametri, accennati in precedenza, presenti nel parameter server.
A partire dalla distribuzione ROS: Diamondback, tutti i parametri lato publisher sono riconfigurabili dinamicamente e in maniera autonoma per ogni topic.
Talvolta, alcune camere, in particolare le webcam, generano le immagini già compresse nel formato JPEG. Durante la scrittura di un driver per questi dispositivi, è possibile utilizzare un approccio più veloce rispetto all'utilizzo di un classico publisher basato su image_transport, copiando direttamente i dati dell'immagine nel campo "data" di un messaggio sensor_msgs/CompressedImage e poi pubblicarlo normalmente su un topic con la forma: /image_raw/compressed.

In questo caso, un qualsiasi nodo ROS che utilizza image_transport sarà in grado di iscriversi ugualmente al topic specificando il tipo di trasporto: "compressed", come se anche il publisher utilizzasse anch'esso image_transport, e tutto funziona correttamente. È ovvio che se si cerca di utilizzare lo stesso topic per gli altri tipi di trasporto (incluso quello raw), questi non saranno disponibili. Questa è inoltre l'unica soluzione utilizzabile nel caso di sistemi Android. Infatti, nella versione di ROS utilizzata nei dispositivi mobile, non esiste il supporto per image_transport, pertanto, nel momento in cui si decide di pubblicare un'immagine, è necessario creare dei publisher distinti, specializzati per ogni tipo di trasporto che si intende fornire. Questi ottengono direttamente i dati nel formato per cui sono specializzati e semplicemente li copiano direttamente all'interno del messaggio ROS loro associato.

Il package theora_image_transport è un plugin che consente di comprimere le immagini utilizzando il codec open-source: Theora, utile per trasmettere sequenze video in diretta, con una banda minima, grazie alle ottimizzazioni tipiche degli standard video. Il plugin è in grado di lavorare solamente con colori a 8-bit o immagini in scala di grigi.
Tra i parametri che è possibile configurare in questo caso, vi è il livello di post-processing, maggiore è quest'ultimo e migliore sarà la qualità delle immagini decodificate, ovviamente al prezzo di un maggiore lavoro per la CPU. I messaggi utilizzati prendono il nome di: theora_image_transport/Packet e hanno la seguente struttura:

std_msgs/Header header
uint8[] data
int32 b_o_s
int32 e_o_s
int64 granulepos
int64 packetno

I campi "b_o_s" e "e_o_s" servono ad indicare se i dati del pacchetto corrente coincidono con l'inizio o la fine di uno stream logico. Il campo "granulepos" indica la posizione del pacchetto rispetto al resto dei dati decodificati, mentre "packetno" è un numero di sequenza che identifica il pacchetto attuale all'interno del bitstream logico a cui appartiene. Le classi TheoraPublisher e TheoraSubscriber, si occupano delle conversioni da/verso questo formato e per fare ciò sfruttano le funzioni messe a disposizione dalla libreria OpenCV, accessibile grazie all'utilizzo del package: cv_bridge, oltre alle quelle della libreria ufficiale: "libtheora".
Nel publisher, si ha la possibilità di utilizzare oggetti cv::Mat, sui quali è possibile applicare le varie funzioni per la conversione dello spazio dei colori e la manipolazione delle singole componenti cromatiche (ad esempio il sotto-campionamento di alcune di esse), infine è possibile generare uno nuovo stream video, grazie alla funzione "cvToTheoraPlane()", che consente la creazione di un buffer nel formato YCbCr, il quale potrà finalmente essere utilizzato dall'encoder per la generazione dei dati, questi verranno poi inseriti all'interno dei messaggi "theora_image_transport/Packet".

In maniera analoga, il subscriber effettua gli stessi passi appena descritti, ma nell'ordine inverso, perciò, il messaggio ROS viene prima decodificato, a questo punto i dati contenuti all'interno del buffer YCbCr riempito dal decoder devono essere inseriti all'interno di una matrice cv::Mat, così da poter effettuare le operazioni di conversione. Infine, grazie alla funzione toImageMsg() , definita nel package cv_bridge, è possibile ottenere nuovamente l'immagine nel formato ROS: sensor_msgs/Image, pronta per essere utilizzata dal nodo, attraverso la solita callback ricevuta come parametro.
Oltre alle classi publisher e subscriber, all'interno del package theora_image_transport è stato definito un nodo chiamato: "ogg_saver", che come suggerisce il nome, consente di salvare il flusso di dati Theora in un file con estensione ".ogv", che potrà poi essere riprodotto in un secondo momento con un qualsiasi player. Dal momento che Theora attualmente non supporta i video a frame-rate variabile, i file salvati avranno sempre indicata la velocità di un frame al secondo.

3.3.3 x264_image_transport

È un package sviluppato da Orchid Project che a sua volta deriva dall'omonimo package realizzato da IntRoLab: un gruppo di ricerca dell'Università di Sherbrooke, Canada. Questo package definisce un plugin per image_transport e ha lo scopo di fornire un meccanismo per pubblicare e ricevere dei frame video codificati utilizzando gli standard MPEG4 e H264, attraverso dei messaggi ROS definiti ad-hoc che prendono il nome di "x264Packet".
Per poter funzionare, il package si appoggia alla suite specializzata per la codifica video: FFMPEG. Quest'ultima comprende al suo interno un insieme di librerie, tra cui AVCodec e AVFormat, che permettono di lavorare con i codec di diversi standard. All'interno del package è presente uno script bash chiamato "build_ffmpeg.sh", quest'ultimo facilita l'installazione della versione 4.0 della suite.
x264_image_transport definisce le classi publisher e subscriber, scritte in C++, responsabili della pubblicazione e ricezione dei messaggi x264Packet contenenti i singoli frame, inoltre le due classi si occupano anche della codifica e decodifica di questi ultimi. È importante notare che nonostante i loro nomi, queste due classi non rappresentano dei nodi all'interno del sistema ROS, esse infatti non effettuano nessuna iscrizione verso il master e non chiamano mai direttamente le funzioni: publish() e subscribe(), come tutti i plugin definiti per image_transport, si occupano solamente della conversione dei messaggi ROS da uno specifico formato a un altro e viceversa.

x264Publisher
Questa classe contiene le varie funzioni per poter effettuare la corretta inizializzazione dell'encoder H264/MPEG4. Come ogni plugin definito per image_transport, il publisher utilizza una callback chiamata "publish", che verrà invocata dal nodo di acquisizione nel momento in cui questo ottiene una nuova immagine. Il metodo publish riceve in ingresso due parametri: il primo è il messaggio ROS di tipo sensor_msgs/Image

contenente l'immagine, mentre il secondo è una funzione, chiamata "publish_fn", che dovrà essere utilizzata per la pubblicazione una volta che il messaggio x264Packet sarà pronto.
Innanzitutto, nel momento in cui viene chiamato il metodo publish, si verifica se l'encoder è già stato inizializzato, in caso contrario prima di procedere con la fase di codifica, è necessario chiamare il metodo initialize_codec, fornendogli come parametri in ingresso: le dimensioni (larghezza e altezza), il frame-rate e la codifica delle immagini che il nuovo encoder dovrà ricevere.
All'interno di initialize_codec si procede con la registrazione e inizializzazione del codec H264, da cui poi si ottiene un oggetto di tipo AVCodecContext chiamato encCdcCtx_. Grazie a quest'ultimo è possibile definire i vari parametri da utilizzare durante la codifica video: oltre alle dimensioni e al frame-rate, si possono configurare i campi per la regolazione del target-bitrate del flusso in uscita, il gop (group of picture, ovvero l'intervallo minimo che può intercorrere tra due frame di tipo I), il numero massimo di frame di tipo B all'interno del gop e infine il formato colore utilizzato per l'output (YUV420P). Oltre a questi è possibile specificare il profilo e il livello da utilizzare, nello specifico la classe utilizza gli attributi: "profile", "tune" e "preset".
Una volta terminata la fase di settaggio dei parametri, si procede con l'apertura dell'encoder utilizzando la funzione avcodec_open2, quest'ultima riceve come parametri: l'oggetto codec e l'oggetto encCdcCtx_, contenente la configurazione appena decisa.
Se la funzione avcodec_open2 restituisce un valore negativo, allora non è stato possibile inizializzare il nuovo encoder, ciò può essere dovuto a una configurazione errata dei parametri, che risultano incompatibili con i requisiti del codec, in questo caso l'inizializzazione non può proseguire e il nodo viene terminato.
Invece, nel caso in cui l'apertura dell'encoder restituisce un valore maggiore o uguale a zero, si può procedere con la fase di allocazione dell'oggetto encFrame_ appartenente alla classe: AVCodecFrame, questo ultimo conterrà di volta in volta i dati in ingresso destinati all'encoder ovvero i singoli frame non compressi da cui verrà generato il flusso video; per effettuare un'allocazione corretta è necessario conoscere: le dimensioni del frame e il tipo di formato dei pixel (spazio del colore) accettato dall'encoder, ovvero quello scelto in precedenza.
In seguito, si procede con l'inizializzazione dell'oggetto sws_ctx_ appartenente alla classe SwsContext (software scale context), questo ultimo ha lo scopo di convertire le immagini in ingresso dalla loro codifica colore (si considerano i casi: BGR8, RGB8, RGB16, YUV422 e BAYER_GRGB16) a quella utilizzata dall'encoder, ovvero YUV420P, per fare ciò, ci si basa sul valore "encoding", che era stato ottenuto come parametro in ingresso, a sua volta ricavato dall'omonimo campo del messaggio sensor_msgs/Image.
L'ultima operazione svolta all'interno della funzione initialize_codec, è l'inizializzazione dell'oggetto encodedPacket_, che conterrà i dati codificati in uscita dall'encoder, la funzione di inizializzazione in questo caso non si occupa di riservare lo spazio necessario per contenere i dati, infatti, la fase di allocazione vera e propria verrà poi svolta nella funzione che effettuerà la codifica, in quanto non è possibile stabilire a priori quale sarà di volta in volta la dimensione dei dati in uscita.

Terminata questa fase, l'encoder H264 è stato inizializzato, quindi l'esecuzione riprende nella funzione publish; a questo punto si può procedere con la parte di codifica vera e propria delle immagini.
Innanzitutto, viene chiamata la funzione sws_scale, che utilizzando sws_ctx_ e encFrame_, opportunamente configurati in precedenza, ha lo scopo di convertire i dati contenuti nel campo "data" del messaggio sensor_msgs/Image, dalla loro codifica pixel originaria al formato YUV420P, ovvero quello accettato dall'encoder. I dati così ottenuti vengono inseriti all'interno dell'oggetto encFrame_.
A questo punto si può procedere con la codifica vera e propria chiamando la funzione: *avcodec_encode_video2*, quest'ultima riceve come parametri di ingresso: encCdcCtx_, encodedPacket_, encFrame_ e una variabile chiamata: got_output, che conterrà il numero effettivo di byte codificati.
Se l'operazione va a buon fine, sia il campo size dell'oggetto encodedPacket_ che la variabile got_output, avranno ragionevolmente un valore maggiore di zero. Se questa condizione è verificata, ciò significa che la codifica è avvenuta senza errori e si può procedere con la pubblicazione dei dati. Perciò, viene creato un nuovo oggetto x264Packet, quest'ultimo ha la seguente struttura:

```
Header header     # Original sensor_msgs/Image header
uint8[] data      # x264 NAL unit
int32 img_width   # Image width
int32 img_height  # Image height
uint8 codec       # Specifies the exact codec used by the encoder
uint8 CODEC_H264 = 0
uint8 CODEC_MPEG4 = 1
```

Il campo header è quello solito definito per i messaggi ROS, in questo caso viene direttamente copiato quello già presente nel messaggio sensor_msgs/Image ricevuto in ingresso. I campi img_width e img_height conterranno rispettivamente la larghezza e l'altezza del frame, mentre il campo "codec" , che può assumere valore zero o uno, specifica quale encoder effettivamente è stato utilizzato tra H264 e MPEG4; questo campo sarà utile lato subscriber, quando si dovrà decidere quale specifico decoder utilizzare. Infine, il campo "data" rappresenta il buffer destinato ai dati in uscita dall'encoder, ovvero il frame codificato, questi sono contenuti all'interno dell'oggetto: encodedPacket_, perciò viene effettuata una copia del buffer di quest'ultimo e la si assegna al campo in questione. A questo punto è possibile pubblicare il messaggio chiamando la callback: publish_fn, ricevuta in ingresso e passandogli come parametro il messaggio x264Packet corrente.

x264Subscriber

Analogamente a quanto accade per il publisher, anche nel subscriber prima di procedere con la decodifica, è necessario inizializzare il codec, in questo caso vengono creati sia il decoder H264 che quello MPEG4; l'utilizzo di uno o dell'altro verrà stabilito sulla base del campo "codec" all'interno del messaggio x264Packet in ingresso.

Prima di fare ciò, è sempre necessario chiamare le funzioni di registrazione dei codec, identiche a quelle già descritte per il publisher, dopodiché si procede con la settaggio dei campi del messaggio sensor_msgs/Image che conterrà i dati decodificati, operazione possibile perché rispetto al caso duale del publisher, dove non si sapeva a priori che dimensione avrebbero avuto i singoli frame in uscita, in questo caso, una volta note: la larghezza, l'altezza e la codifica utilizzata per il colore (e quindi il numero di byte per pixel) del frame, è già possibile allocare una struttura dati di dimensione opportuna.
Finita questa fase, si entra all'interno di un ciclo for che esegue un'iterazione per ciascuno dei codec (2 iterazioni), all'interno del ciclo gli oggetti associati ai due codec H264 e MPEG4 sono organizzati all'interno di un array chiamato "codecs", ogni elemento di quest'ultimo rappresenta una struct di tipo codec_meta (definita nel file header della classe x264Subscriber), al cui interno sono presenti i vari campi necessari per l'utilizzo del codec in questione e che pertanto devono essere inizializzati. Durante il ciclo, per ogni iterazione viene innanzitutto inizializzato il codec, in seguito grazie ad esso è possibile ottenere l'oggetto AVCodecContext che analogamente a ciò accede nel publisher consente di specificare i parametri da utilizzare per la configurazione del decoder.
Dopodiché viene chiamata la funzione avcodec_open2 e nel caso in cui questa vada a buon fine si procede con l'allocazione dell'oggetto m_pFrame, che dovrà contenere i dati una volta decodificati. A questo punto, una volta terminata l'inizializzazione, nel momento in cui arriva un nuovo messaggio x264Packet viene chiamato il metodo x264Subscriber::internalCallback(), quest'ultimo, oltre al messaggio, riceve come parametro una funzione callback, che dovrà essere chiamata una volta eseguita la decodifica del frame, questa funzione appartiene alla classe del nodo di elaborazione o visualizzazione, che grazie ad essa potrà in seguito lavorare sull'immagine decodificata.
Nel momento in cui viene ricevuto un nuovo messaggio, il campo codec presente in quest'ultimo viene utilizzato come indice per accedere alla struttura: "codecs", descritta sopra. In seguito, viene inizializzato un oggetto AVPacket e al suo interno verranno copiati i byte del campo data del messaggio, dopodiché, in modo esattamente speculare a quanto accadeva del publisher, viene chiamata la funzione: av_decode_video2 che si occupa della decodifica. Questa funzione inserisce i dati prodotti, all'interno dell'oggetto "codecs[i].m_pFrame" di tipo AVFrame. A questo punto, l'ultima operazione da compiere è la riconversione dal formato: YUV420P, utilizzato per i frame x264, in un formato accettato dal nodo di visualizzazione, in questo caso: RGB24. Infine, una volta effettuata la conversione, i dati vengono copiati all'interno del messaggio sensor_msgs/Image allocato in precedenza, dopodiché quest'ultimo viene passato come parametro alla funzione "callback()".

Il comportamento descritto sia per il publisher che per il subscriber, risulta in linea con quanto già accadeva nel caso degli altri image_transport_plugins, il loro scopo consiste nel gestire la conversione da un formato all'altro e viceversa, senza che ciò influisca sugli utilizzatori finali, ovvero i nodi di acquisizione e visualizzazione. x264_image_transport definisce altre due classi, chiamate x264Publisher_test e

x264Subscriber_test, utili per verificare il funzionamento del package. Sono inoltre presenti dei file "launch" ovvero: "extract.launch" e "playback.launch", questi consentono di avviare un nodo republish passandogli come parametro il nome del topic x264 a cui iscriversi. Nel primo caso le immagini verranno pubblicate su un nuovo topic in formato raw, mentre nel secondo in formato compressed (nel frattempo viene avviata anche la registrazione utilizzando un nodo rosbag).

3.3.4 x264_video_transport

Questo package è stato creato nell'ambito di questa attività di tesi e deriva direttamente dal precedente, con lo scopo di aggiornarlo, migliorarlo e soprattutto adattarlo all'utilizzo congiunto con l'applicazione Android basata su ROS, che verrà introdotta nel prossimo capitolo.
Per quanto riguarda gli aggiornamenti, sono state sostituite molte delle funzioni delle librerie ffmpeg utilizzate per: codifica, decodifica, allocazione e de-allocazione dei frame, ormai deprecate, con le equivalenti versioni più aggiornate.
All'interno della classe x264Subscriber è stata modificata l'espressione del blocco condizionale, dove veniva effettuata la verifica sullo stato di inizializzazione del codec. Nella nuova versione, la funzione initialize_codec, non viene chiamata solamente quando il codec non è ancora inizializzato, ma ogniqualvolta viene rilevata una modifica nella dimensione (larghezza e altezza) riportata nel messaggio x264Packet. Ciò è molto utile, ad esempio, nell'uso congiunto con l'applicazione Android, dove la risoluzione delle immagini può essere modificata dinamicamente dall'utente; nel caso del package x264_image_transport, un cambiamento improvviso della risoluzione, portava a una decodifica errata, in quanto i parametri del codec venivano impostati una sola volta durante l'avvio. Per risolvere la situazione era necessario terminare il nodo e quindi riavviarlo, in questo caso invece, il subscriber sarà in grado di adattarsi dinamicamente agli eventuali cambiamenti senza interrompere la visualizzazione.
È stato in seguito modificato il messaggio x264Packet aggiungendo un nuovo campo di tipo uint32 chiamato "angle", che rappresenta l'angolo rispetto al quale dovrà essere ruotata l'immagine una volta decodificata. Questo campo è stato introdotto con lo scopo di evitare l'esecuzione della rotazione delle immagini direttamente sul dispositivo Android, poiché questa operazione nel caso di immagini di grandi dimensioni risulta essere particolarmente onerosa in termini di tempo, ciò che si è notato, è stato un drastico abbassamento nel frame-rate, dovuto alle tempistiche dell'algoritmo di rotazione applicato su ogni singola immagine.
Svolgendo questa operazione sul PC, vengono notevolmente ridotte queste tempistiche, avendo a disposizione una maggiore potenza di calcolo. La rotazione viene effettuata dopo la fase di decodifica, l'immagine viene copiata all'interno di un messaggio sensor_msgs/Image dopodiché, utilizzando il package cv_bridge, presentato nei paragrafi precedenti, l'immagine viene copiata all'interno di un oggetto CvImage grazie alla funzione: cv_bridge:: toCvCopy(), che riceve come parametri l'immagine ROS e la codifica pixel utilizzata per essa. L'immagine OpenCV così ottenuta è accessibile

mediante un puntatore chiamato cv_ptr, attraverso quest'ultimo si accede campo "image" che verrà poi assegnato a una matrice cv::Mat, successivamente passata alla funzione rotate().
All'interno di quest'ultima viene chiamata la funzione cv::getRotationMatrix2D, responsabile della rotazione della matrice. Una volta svolta questa operazione, la nuova matrice viene assegnata a cv_ptr e l'immagine OpenCV viene nuovamente trasformata in un messaggio sensor_msgs/Image utilizzando la funzione cv_ptr-> toImageMsg().

3.4 Visualizzazione e salvataggio delle immagini

Per la visualizzazione delle immagini pubblicate attraverso i topic, il package maggiormente utilizzato è image_view, esso fornisce un'alternativa più leggera rispetto al package concorrente: rviz, specializzato per la visualizzazione di sequenze 3D. Il package e collocato all'interno dello stack: "image_pipeline" e definisce il comportamento dei nodi: image_view, stereo_view, disparity_view, image_saver, extract_images e video_recorder.

3.4.1 image_view

È un nodo per la visualizzazione delle immagini, contenute all'interno di appositi messaggi ROS, una volta che queste sono state pubblicate sui relativi topic. Se si vuole visualizzare un flusso di immagini compresse (preferibile ad esempio se si utilizza una connessione wireless), sfruttando le funzionalità del package image_transport, è possibile specificare il tipo di trasporto desiderato aggiungendolo come argomento da linea di comando. Per esempio, se nel sistema risultano installate le modalità di trasporto: theora_image_transport e x264_video_transport è possibile richiedere l'utilizzo di una o dell'altra, avviando il nodo image_view e specificando il trasporto desiderato:

$ rosrun image_view image_view image:=/camera/image theora
$ rosrun image_view image_view image:=/camera/image x264

stereo_view
All'interno del package: image_view, oltre al nodo omonimo è presente anche "stereo_view", quest'ultimo è un nodo specializzato nella visualizzazione di immagini stereoscopiche, per fare ciò è necessario iscriversi contemporaneamente a due topic distinti, su ognuno di questi vengono pubblicate rispettivamente il flusso di immagini di sinistra e di destra ottenute con due inquadrature differenti, una volta acquisite e sincronizzate, è possibile visualizzare l'immagine stereoscopica risultante. Ad esempio, per visualizzare una coppia di immagini stereoscopiche, pubblicate rispettivamente sui topic "/camera/left/image_rect" e "camera/right/image_rect", è possibile utilizzare:

$ rosrun image_view stereo_view stereo:=/camera image:=image_rect

Il nodo stereo_view è in grado di generare l'immagine di disparità, calcolata a partire dalla coppia stereoscopica, utilizzando la mappatura del colore per maggiore chiarezza. Come per image_view, è possibile specificare il tipo di image_transport da utilizzare per la coppia di immagini acquisite, indicandolo come argomento da linea di comando.

image_saver
È un nodo che consente di salvare il flusso di immagini pubblicate su un topic, come singoli file in formato jpg o png. Da linea di comando, è possibile utilizzare:

$ rosrun image_view image_saver image:=[nome del topic]

video_recorder
Questo strumento consente di registrare un video, a partire dal flusso di immagini pubblicate su un generico topic ROS e infine di salvarlo su un file locale. Il nodo sfrutta la classe, appartenente alla libreria OpenCV, chiamata: "VideoWriter". È possibile specificare: il nome del file, il frame-rate, il codec e il formato colore; con le opzioni predefinite, si ha un frame-rate di 15fps, per la codifica si utilizza MJPG e il formato è bgr8. Il video viene poi salvato all'interno di un file AVI, collocato nella cartella in cui è stato eseguito il comando.

3.4.2 Web server multimediale

Il package web_video_server, consente la visualizzazione delle immagini, normalmente pubblicate utilizzando i topic ROS, attraverso un qualsiasi browser web.
Il nodo web_video_server, una volta avviato, effettua una ricerca di tutti i topic associati al tipo di messaggio: sensor_msgs/Image, che verranno poi inseriti all'interno di una lista.
Dopodiché, il server resterà in ascolto sulla porta 8080 e sarà raggiungibile utilizzando l'indirizzo IP dell'host su cui è in esecuzione. A questo punto, accedendo dal browser verrà mostrata una pagina html con la lista di tutti i topic ROS a cui il server è in grado di collegarsi. È possibile avviare la visualizzazione continua delle immagini pubblicate oppure ottenere delle istantanee, selezionando uno dei topic presenti sulla lista.
Le varie richieste per il server vengono effettuate utilizzando il metodo GET del protocollo http, gli eventuali parametri di configurazione vengono poi specificati all'interno del URL. Tra le varie opzioni è possibile specificare (oltre al nome del topic): la larghezza e l'altezza delle immagini e il tipo di trasporto da utilizzare lato web: MJPEG (default), vp8, vp9, ros_compressed e H264. Per ogni tipo di trasporto è possibile specificare a sua volta, una serie di parametri (ad esempio, la qualità della compressione dell'algoritmo JPEG per il tipo ros_compressed). Per quanto riguarda le codifiche video è possibile specificare il livello di quantizzazione massima e minima, la qualità dello stream (default: real-time), il gop e il bitrate.

La soluzione web_video_server, è particolarmente utile nei casi in cui è necessario visualizzare le immagini pubblicate sui topic, utilizzando sistemi in cui ROS non è installato, infatti è necessario avere solamente un browser con accesso alla rete su cui è in ascolto server, per potersi connettere ad esso ed utilizzarne i servizi. Tuttavia, questa soluzione presenta anche alcune limitazioni.
Innanzitutto, per quanto riguarda lo streaming web, utilizzando le codifiche video (vp8, vp9 e H264), non è possibile gestire con HTML5 lato browser, il buffer di play-out associato al tag:<*video*> . Per questo motivo, nonostante siano state apportate tutte le ottimizzazioni necessarie lato server ad esempio: evitando i frame di tipo B e riducendo al minimo le dimensioni dei buffer di rete, la latenza finale risulta essere di parecchi secondi (dai 10 ai 15), in quanto spetta al browser, in base alle condizioni della rete, decidere se e di quanto posticipare la riproduzione del video.
Un altro svantaggio è rappresentato dal fatto che la parte ROS del server web, ovvero il nodo che comunica con il resto dell'infrastruttura, può effettuare la subscribe solamente a quei topic associati ai messaggi: "sensor_msgs/Image" , relativi al trasporto "raw"; eventuali altri topic che utilizzano tipologie di trasporto più efficienti come i vari plugin per image_transport (compressed, theora e x264), vengono completamente ignorati ed è necessario ricorrere ad altri espedienti (ad esempio: l'esecuzione di un nodo republish()) se si desidera utilizzarli.

3.4.3 Server RTSP

Questa soluzione sfrutta le funzioni offerte dalla libreria GStreamer per creare un server RTSP in grado di iscriversi a un topic sensor_msgs/Image e di inoltrare le immagini in rete sotto forma di stream video H264, utilizzando il protocollo RTP.
Per la visualizzazione e necessario ricorrere a un player video in grado di utilizzare i protocolli di rete: RTP e RTSP. Anche in questo caso sono state osservate le stesse problematiche della soluzione web-server, infatti non è possibile intervenire sui parametri di riproduzione lato client e quindi migliorare la latenza, inoltre non è possibile utilizzare direttamente i plugin image_transport lato ROS, ciò porta alla necessità di dover ricorrere a dei nodi intermediari.

Capitolo IV

Sviluppo di un'applicazione Android basata su ROS

Sebbene attualmente lo sviluppo dei package ROS sia prevalentemente orientato all'utilizzo delle librerie client per i linguaggi: C++, Python e Lisp; si sta estendendo l'impiego, anche se in via sperimentale, ad altri linguaggi come Java. Quest'ultimo tra i suoi vantaggi ha quello di estendere la panoramica delle piattaforme supportate, favorendo l'integrazione di dispositivi di largo utilizzo come gli smartphone e i tablet basati su Android.

4.1 Applicazione Android: PIC4SeR Monitoring

Il progetto si basa sui package ufficiali: "rosjava" e "android_core_components", che mettono a disposizione tutte le classi necessarie per la gestione degli elementi caratteristici dell'ambiente ROS: nodi, topic, servizi, messaggi e protocolli di comunicazione.
Lo scopo dell'applicazione consiste nel voler sfruttare i dati forniti dai sensori presenti sugli smartphone Android rendendo disponibili le informazioni da essi acquisite, utili per dare un supporto aggiuntivo al monitoraggio remoto di un generico rover/drone non particolarmente equipaggiato.
Grazie all'integrazione con l'architettura ROS, le singole risorse di cui dispone il dispositivo possono essere viste come *topic* a cui ci si può iscrivere e *service* a cui si possono inoltrare delle richieste, messi a disposizione del resto del sistema.
Allo stato attuale, l'applicazione consente di acquisire le informazioni riguardanti: la localizzazione del dispositivo, l'accelerazione lineare lungo i tre assi cartesiani, la velocità angolare, il vettore di rotazione, la temperatura ambientale, la pressione, il campo magnetico, l'illuminamento, un insieme di informazioni sulla rete cellulare (come il rapporto segnale/rumore e la potenza del segnale relativo agli standard 3G/LTE), informazioni sullo stato del Wi-Fi, le immagini acquisite in tempo reale utilizzando le fotocamere a disposizione, informazioni sullo stato della batteria del dispositivo. È inoltre presente una parte di acquisizione dei dati forniti dal sistema GNSS, che negli smartphone più recenti è in grado di fornire un insieme di informazioni dettagliate provenienti dai satelliti, note con il nome di *"raw measurements"*.
L'applicazione ha una struttura simile ad altre che fanno utilizzo del package ROS: "android_core_components", pertanto, al suo interno sono presenti, oltre all'attività principale rappresentata dalla classe *Monitoring*, l'attività: "*MasterChooser*", che si occupa di acquisire lo URI corrispondente al master ROS e di stabilire una connessione con esso, e un terzo processo (Android-S*ervice*), sempre attivo in background, chiamato *"NodeMainExecutorService"*, che si occupa di eseguire le operazioni associate al nodo ROS definito nell'applicazione.

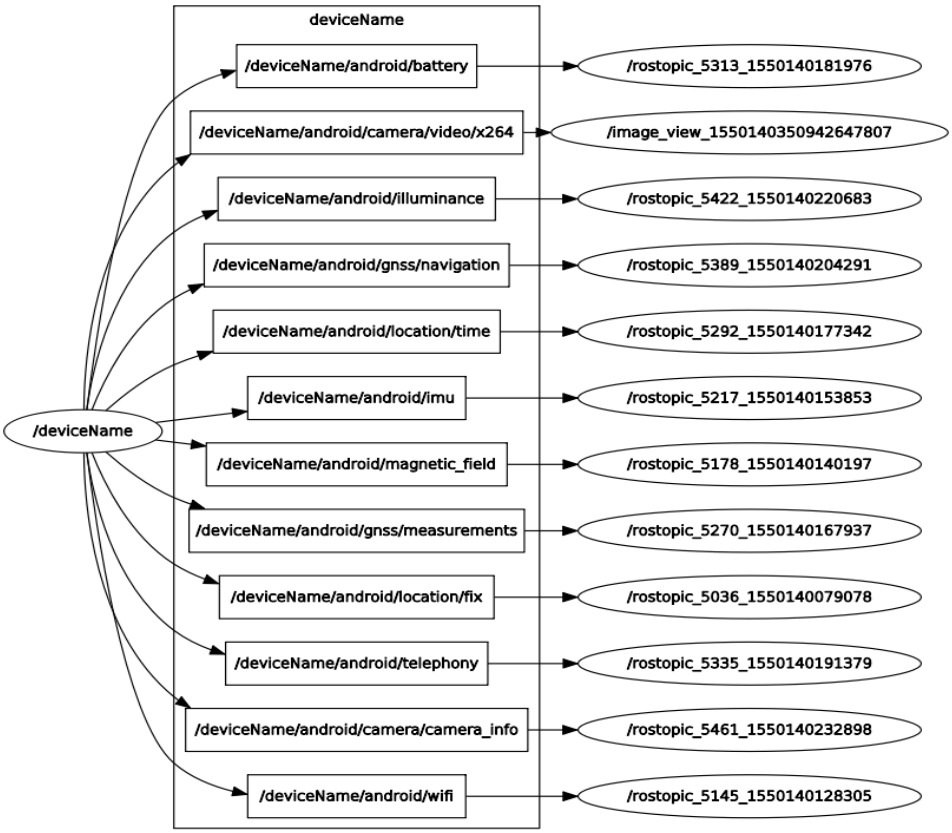

Figura 8 topic utilizzati dall'applicazione Android

4.2 MasterChooser

La classe MasterChooser, relativa all'omonima "activity" Android, è definita all'interno del modulo: *"android_core_components"* e a sua volta estende la classe *AppCompatActivity*. La classe è associata alla risorsa *master_chooser.xml* e definisce la prima interfaccia presentata all'utente nel momento in cui viene avviata l'applicazione. Questa classe ha il compito di gestire l'acquisizione dello URI contenente l'indirizzo IP e la porta su cui è in ascolto il server XMLRPC del master ROS. Un'alternativa all'inserimento dello URI è l'accoppiamento mediante l'utilizzo di un QRCode, selezionabile grazie all'apposito bottone. Se non si desidera collegare il dispositivo a un master remoto è possibile, selezionando il bottone: *"Show advanced options"*, decidere se istanziare un nuovo master direttamente sul dispositivo, questo potrà essere pubblico o privato a seconda che si scelga o meno di voler associare altri nodi da remoto. L'interfaccia originale della classe MasterChooser definita nella risorsa layout: *master_choser.xml*, è stata in seguito modificata con l'obbiettivo di adattarla allo stile utilizzato nelle altre parti dell'applicazione.

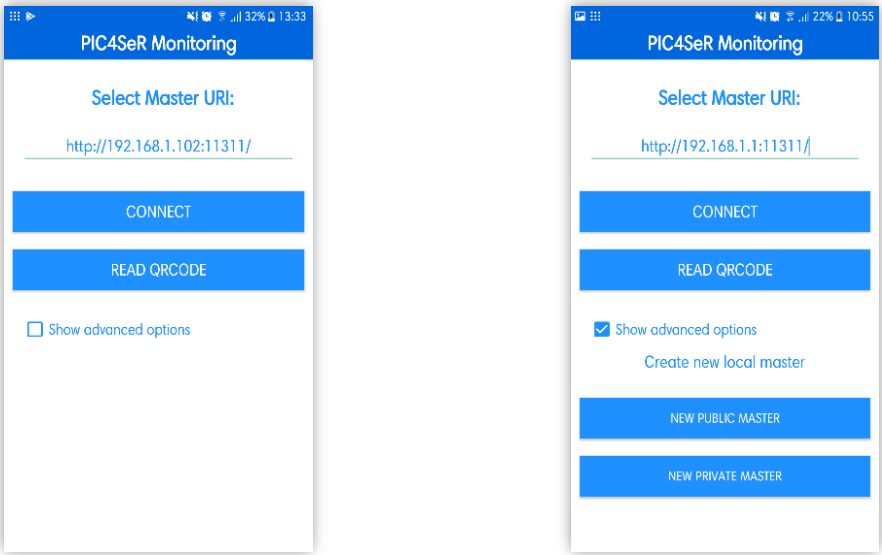

Figura 9 schermata MasterChooser

4.3 MainActivity: Monitoring

Monitoring è la classe principale dell'applicazione e a sua volta estende la classe *RosActivity* implementando il metodo astratto: *init*. Al suo interno *Monitoring* contiene, oltre ai metodi necessari a creare il nodo ROS che rappresenta il dispositivo, anche la parte di codice dove si definiscono i vari topic e service.
Il primo metodo della classe Monitoring ad essere chiamato durante la fase di avvio dell'applicazione è : *onCreate*. Al suo interno, oltre ad associare la risorsa principale: *"main.xml"* (dove viene definito il layout dell'interfaccia utente) alla classe corrente, si istanzia anche un oggetto di tipo: *Logger,* che ha il compito di registrare tutte le operazioni svolte all'interno del programma e di tener traccia dei valori provenienti dai vari sensori. Una volta stabilita la connessione con il master, il servizio *NodeMainExecutorService* chiama il metodo: *init*, della classe principale passandogli come parametro un'implementazione dell'interfaccia: *NodeMainExecutor*, che insieme alle impostazioni di configurazione del nodo, contenute nell'oggetto: *NodeConfiguration*, vengono assegnati a degli attributi della classe Monitoring. Inoltre, sempre all'interno dello stesso metodo viene definita una nuova finestra di dialogo, eseguita in seguito nel thread responsabile dell'interfaccia grafica, al suo interno si dovrà inserire il nome da assegnare al nodo ROS che rappresenterà il dispositivo.
Una volta terminata questa fase, viene chiamato il metodo: *runRosNode*, della classe principale. All'interno di questo metodo viene inizializzato l'attributo chiamato: *rosnode*, come un nuovo oggetto a cui viene assegnato il nome scelto dall'utente, parallelamente viene definito il metodo *onStart* che assegna in modo sicuro, all'interno di un blocco syncronized, il proprio parametro in ingresso: *connectedNode*, all'omonimo attributo della classe *Monitoring*. Il metodo *onStart* verrà poi chiamato in un secondo momento dal servizio *NodeMainExecutor* (attivo in background). Dopodiché, sempre

nel metodo *runRosNode* viene avviato il nodo ROS appena definito utilizzando il metodo: *"nodeMainExecutor.execute()"*, che riceve come parametri: il nodo e la configurazione di rete necessaria a ROS per stabilire la comunicazione. L'ultima operazione svolta nel metodo *runRosNode* consiste in una chiamata a un altro metodo della classe Monitoring chiamato: *"SensorsConfiguration"*. Quest'ultimo ha il compito di richiedere i permessi all'utente, nel caso di risorse che necessitano di un'autorizzazione esplicita (camera e servizio di localizzazione), qualora non fossero stati ancora stati concessi, dopodiché, si occupa di inizializzare correttamente tutte le classi necessarie per l'acquisizione e la pubblicazione dei dati provenienti dai vari sensori.

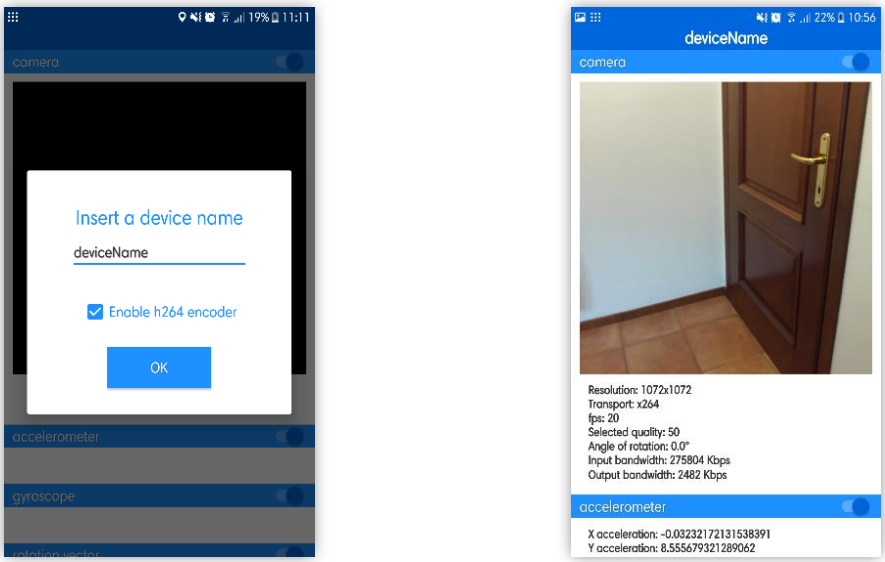

Figura 10 a sinistra finestra di dialogo per la scelta del nome, a destra schermata MainActivity

4.4 Localizzazione

La localizzazione cosiddetta "*fine*" (non approssimata) consiste nell'informazione ricavata principalmente dal sistema GPS, che fornisce, oltre alle coordinate del dispositivo, una serie di informazioni aggiuntive utili per determinarne la posizione. Per poter utilizzare il servizio di localizzazione è necessario dichiararne l'utilizzo all'interno del file: *"AndroidManifest.xml"*, dopodiché, si verifica se l'utente ha già concesso i permessi per l'accesso al servizio, in caso contrario deve essere creata una finestra di dialogo per richiederli.
La configurazione del servizio di localizzazione inizia all'interno del metodo *SensorsConfiguration* della classe principale, la prima operazione consiste nell'istanziare un oggetto della classe "LocationManager" che fornisce una serie di metodi, fondamentali per la gestione e l'acquisizione dei dati.
Il servizio di localizzazione oltre al GPS, può sfruttare altri *provider* quando quest'ultimo non è disponibile, ad esempio a causa della mancata copertura satellitare. L'applicazione può infatti ricevere informazioni, circa la posizione, da altre sorgenti come

la rete mobile cellulare o le informazioni di posizionamento fornite dal Wi-Fi, che tuttavia hanno un'accuratezza decisamente inferiore, per questo motivo chi riceve i dati pubblicati, deve ottenere anche il nome del *provider* da cui provengono, così da poter valutare in maniera adeguata la qualità delle informazioni.
Per determinare se il servizio di localizzazione è disponibile su un dato dispositivo viene innanzitutto effettuata una verifica sul numero di provider a disposizione e qualora questo coincida con zero il servizio viene considerato non disponibile. Nel caso in cui è presente almeno un provider, la configurazione può proseguire, verificando i permessi e richiedendoli all'utente se necessario.
Una volta ottenuti questi ultimi, viene chiamato il metodo *configureLocation,* che si occupa di creare una nuova istanza della classe *LocationProvider*, vengono inoltre creati due nuovi publisher associati ai topic *"deviceName/android/location/fix"* e *"deviceName/android/location/time"*.
La classe "LocationProvider", che si occupa di acquisire i dati relativi al GPS e di pubblicarli sul topic ROS specifico, a sua volta sfrutta le classi del *package*: "android.location" (parte dell'API ufficiale) che consente l'acquisizione dei dati in modo asincrono secondo un meccanismo comune basato su *callback*.
LocationProvider include tra i propri attributi un'istanza della classe *LocationManager*, che si occupa di richiedere gli aggiornamenti sulla posizione, specificando le sorgenti a cui è interessato e fornendo un oggetto di tipo *LocationListener,* che deve essere utilizzato nel momento in cui si ottengono nuovi dati.
Un *listener* definisce al suo interno i metodi da chiamare nel caso in cui si verificano le seguenti situazioni: il *provider* è stato abilitato, il *provider* è stato disabilitato, lo stato della risorsa è cambiato (es: varia il numero di satelliti) e infine quando la posizione cambiata. In quest'ultimo caso, il metodo in questione, chiamato: *"onLocationChanged"*, riceve come parametro un oggetto di tipo *Location* contenente al suo interno le informazioni sulla posizione del dispositivo, tra queste: latitudine, longitudine, altitudine, covarianza e istante temporale UTC fornito dal satellite.
I valori così ottenuti devono essere mappati all'interno di messaggi ROS, in questo caso vengono utilizzati due tipi di messaggio differenti: *"NavSatFix"* e *"TimeReference"*; entrambi appartenenti al package *sensor_msgs*.
I messaggi del primo tipo vengono utilizzati per le informazioni riguardanti le coordinate, mentre quelli del secondo tipo contengono l'informazione riguardante l'istante di acquisizione, i dati vengono quindi inviati a coppie, correlati tra loro grazie al fatto che entrambi hanno lo stesso *header* che comprende un identificativo univoco e lo stesso timestamp.
Una volta generati i messaggi, all'interno del metodo *useNewLocation()*, questi vengono pubblicati sui *topic* loro associati, con l'utilizzo di due oggetti *publisher*, anch'essi presenti tra gli attributi della classe *LocationProvider*. Allo stesso tempo questi dati vengono anche scritti nel file di log e utilizzati per aggiornare l'interfaccia dell'applicazione.

4.5 IMU

Un'unità di misura inerziale (inertial measurement unit) è un sistema elettronico basato su sensori inerziali; nel caso dei dispositivi Android, si considerano l'accelerometro e il giroscopio. Le informazioni pubblicate utilizzano il formato definito dal messaggio ROS: "sensor_msgs/Imu" che prevede i seguenti campi:

- std_msgs/Header header
- geometry_msgs/Quaternion orientation
- float64[9] orientation_covariance
- geometry_msgs/Vector3 angular_velocity
- float64[9] angular_velocity_covariance
- geometry_msgs/Vector3 linear_acceleration
- float64[9] linear_acceleration_covariance

Il campo orientation rappresenta il vettore di rotazione calcolato in un sistema ortonormale, dove la direzione dell'asse y coincide con quella del nord magnetico e l'asse z è sempre perpendicolare alla superficie terrestre.
Orientation si divide in quattro componenti:

- componente lungo l'asse x ($x * \sin(\theta/2)$)
- componente lungo l'asse y ($y * \sin(\theta/2)$)
- componente lungo l'asse z ($z * \sin(\theta/2)$)
- componente scalare ($\cos(\theta/2)$)

Il vettore di rotazione viene normalmente ricavato grazie ai valori ottenuti dal giroscopio, tuttavia qualora quest'ultimo non fosse presente nel dispositivo, il valore viene ricavato a partire dai dati forniti dal magnetometro, i quali risultano essere meno precisi; in assenza di entrambi i sensori questo campo non viene utilizzato.
Il secondo componente del messaggio IMU è la velocità angolare lungo i tre assi cartesiani, espressa in radianti al secondo. Tale campo viene utilizzato per i valori forniti dal giroscopio presente sul dispositivo.
Infine, il terzo componente è l'accelerazione lineare lungo i tre assi, espressa in metri al secondo quadrato e i cui valori coincidono con quelli ottenuti dall'accelerometro.
Ad ognuno dei tre campi è inoltre associata una matrice delle covarianze, rappresentata su un'unica dimensione da un vettore con nove elementi, ognuno dei quali esprime il valore della covarianza lungo un asse rispetto agli altri due; la covarianza è ricavata come il quadrato della deviazione standard, che coincide con il campo *accuracy*, della classe *SensorEvent*. Qualora uno o più sensori non fossero presenti nel dispositivo, il primo campo del relativo vettore delle covarianze viene impostato al valore: -1 (come da convenzione), ciò è utile per indicare agli eventuali lettori del messaggio che tale campo non è presente. Inoltre, l'assenza del sensore viene segnalata all'utente mediante un apposito messaggio sull'interfaccia grafica dell'applicazione.

4.6 Package: android.sensor

L'API Android oltre all'accelerometro e al giroscopio, consente l'accesso anche ad altri sensori che possono essere utili per il monitoraggio, questi sono riuniti all'interno del package *android.sensor* e comprendono:

- magnetometro
- termometro ambientale
- barometro
- sensore di luminosità

Per ognuno di questi sensori viene definita una specifica classe che si occupa di creare gli oggetti necessari per: *logging*, acquisizione e pubblicazione.
All'interno del metodo *SensorsConfiguration*, nel momento in cui ciascuna di queste classi viene creata, le viene passato un oggetto di tipo *TextView*, che consente di aggiornare il relativo campo testuale nell'interfaccia grafica dell'applicazione, inoltre vengono inizializzati il *SensorManager* e il *ROS-publisher* relativo.
Ciascuna classe associata a un sensore implementa l'interfaccia *"SensorEventListener"* che prevede la definizione dei metodi: *"onSensorChanged"* e *"onAccuracyChanged"*. Il primo metodo viene invocato nel momento in cui il sensore acquisisce un nuovo dato e riceve come parametro un'oggetto *SensorEvent*, che in maniera analoga ai casi dell'accelerometro e del giroscopio, contiene due campi: il vettore *values* (con i dati che il sensore ha appena acquisito) e un campo *accuracy* che esprime l'accuratezza della misurazione.
Nel momento in cui il metodo *onSensorChanged* viene invocato, si verifica che l'oggetto passato come parametro corrisponda a quello atteso e in caso di esito positivo si procede con la fase di *logging*. Questa sarà seguita dalla pubblicazione sul relativo topic ROS e infine si procederà con l'aggiornamento dell'interfaccia dell'applicazione.
Per poter essere pubblicati all'interno del sistema ROS, i dati provenienti dai vari sensori devono essere associati ad uno specifico messaggio, per fare ciò sono stati utilizzati i formati definiti all'interno del package ROS: *sensor_msgs*.

Sensore	Messaggio ROS
Magnetometro	*sensor_msgs/MagneticField*
Termometro (ambientale)	*sensor_msgs/Temperature*
Barometro	*sensor_msgs/FluidPressure*
Sensore di luminosità	*sensor_msgs/Illuminance*

Figura 11 Tabella corrispondenze Sensore-Messaggio

4.7 Stato della batteria

La classe *BatteryStateProvider*, si occupa di acquisire e pubblicare le informazioni sullo stato della batteria, per fare ciò sfrutta il formato definito all'interno del messaggio ROS: *sensor_msgs/BatteryState*.
BatteryStateProvider estende a sua volta la classe: *BroadcastReceiver*, della quale implementa il metodo astratto: *onReceive*, il quale viene invocato ogniqualvolta lo stato della batteria subisce una variazione.
Poiché questi cambiamenti tendono ad essere poco frequenti, si è deciso di utilizzare un thread secondario che ogni dieci secondi richiede nuovi dati sulla batteria anche nel caso in cui questi non siano variati, questo allo scopo di fornire un aggiornamento costante a coloro che sono in ascolto sul topic e che magari non hanno ancora ricevuto alcun dato.
I valori provenienti dalla batteria sono accessibili mediante un oggetto di tipo *Intent*, i cui valori devono essere mappati all'interno del messaggio *sensor_msgs/BatteryState*.
Si possono quindi identificare alcuni campi comuni ai due formati:

- *status:* definisce lo stato della batteria e che può essere assumete i valori: *unknown*, *charging*, *discharging*, *not-charging* e *full*.
- *health:* stabilisce lo stato di salute della batteria e può essere: *unknown, good, over-heat, over-voltage, dead, cold, unspecified-failure*.
- *technology:* identifica la tecnologia a cui la batteria appartiene: NIMH, LI-ON, LIPO, LIFE, NICD, LIMN.

Oltre a questi campi sono presenti la percentuale di carica, il voltaggio, la corrente, il tipo di ricarica e un booleano che sta ad indicare se la batteria è presente o meno.

4.8 Camera

Tra i dati forniti dall'applicazione, una componente fondamentale è rappresentata dalle immagini acquisite dalle fotocamere presenti sul dispositivo. All'interno della classe *CameraActivity* sì definiscono i vari metodi per: la configurazione, l'acquisizione, la pubblicazione e il rilascio delle risorse necessarie per l'utilizzo di una singola fotocamera. La classe CameraActivity sfrutta le funzionalità offerte dal package: *android.hardware.camera*.
Nel momento in cui si decide di creare una nuova istanza della classe (all'interno dell'attività: Monitoring) è necessario associarla a una corrispondente risorsa dell'interfaccia grafica, la quale permetterà la visualizzazione su quest'ultima di una *preview* associata alle immagini. Dopodiché, è necessario fornire alla classe le varie istanze, già inizializzate, dei ROS-publisher associati ai tre topic relativi alla camera:

- *Publisher< sensor_msgs.CompressedImage> imagePublisher*
  - */[deviceName]/android/camera/image/compressed*
- *Publisher< x264_video_transport.x264Packet> videoPublisher*
  - */[deviceName]/android/camera/video/x264*
- *Publisher<sensor_msgs.CameraInfo> cameraInfoPublisher*
  - */[deviceName]/android/camera/camera_info*

Allo stesso modo vengono inizializzati i vari ROS-service, per ognuno di essi viene creato un nuovo server che riceve: il nome associato al servizio e un'istanza della classe all'interno della quale è definito il comportamento di quest'ultimo [2].
L'ultima fase dell'inizializzazione prevede l'utilizzo del metodo *setCamera*, chiamato sempre nell'ambito della classe principale, al quale viene passato come parametro un'istanza della classe: *Camera*. Quest'ultima, oltre a rappresentare un riferimento alla componente hardware, fornisce i metodi necessari per configurarne i parametri e per accedere di volta in volta ai dati acquisiti.
Per fare ciò, all'interno di *setCamera* viene chiamato il metodo *setupCameraParameters*. All'interno di quest'ultimo si inizializzano i vari parametri. L'altezza e la larghezza delle immagini vengono selezionate grazie all'utilizzo del metodo: *getOptimalPreviewSize*, che date le dimensioni associate alla preview e l'elenco delle risoluzioni supportate dalla camera attualmente utilizzata, è in grado di selezionare la risoluzione ottimale. Ciò viene fatto sulla base dell'*aspect-ratio:* applicando un confronto tra il formato desiderato e quello supportato usando come criterio una soglia massima di tolleranza.
Oltre ai vari parametri associati alla camera, durante questa fase vengono anche impostati i valori iniziali degli attributi: *quality* ed *angle*; indispensabili per il funzionamento di alcuni dei ROS service presentati in seguito.
Terminata la fase di configurazione, si può procedere con l'acquisizione delle immagini, questa segue principalmente due vie, la prima porta i dati acquisiti ad essere visualizzati all'interno dell'apposita *preview* destinata all'interfaccia dell'applicazione, la seconda consente la pubblicazione dei dati all'interno del contesto ROS.
In quest'ultimo caso, si ricorre a una sottoclasse, definita all'interno di *CameraActivity*, chiamata: *compressedImagePublishing*, presente come attributo nella classe madre. L'acquisizione dei dati dalla camera avviene utilizzando un meccanismo asincrono. Ogniqualvolta viene ricevuta una nuova immagine, a questa segue una chiamata al metodo: *onPreviewFrame* , appartenente a un'altra sottoclasse di *CameraActivity,* chiamata: *bufferingPreviewCallback* [3]*,* la quale a sua volta chiama: *onNewRawImage*, della classe *CompressedImagePublishing*, passandogli un array di byte contenente i dati dell'immagine appena acquisita. Quest'ultima sarà infine pubblicata attraverso appositi messaggi ROS sui relativi *topic*. Questa operazione può avvenite secondo due

[2] Ulteriori dettagli sono presenti nella parte successiva dedicata ai ROS-Service associati alla fotocamera.
[3] Un'implementazione dell'interfaccia: Camera.PreviewCallback

modalità: invio di immagini compresse in formato JPEG (tipo: compressed) e invio di una sequenza di frame video codificati utilizzando lo standard H.264 (tipo: x264).
Come anticipato nel precedente capitolo, attualmente la versione ROS presente su Android, non supporta l'utilizzo di *image_transport*, pertanto le immagini vengono convertite utilizzando delle librerie opportune, dopodiché queste vengono inserite direttamente all'interno dei messaggi ROS, utilizzando dei publisher distinti, ciascuno specializzato per un determinato formato. In ogni caso, i topic associati ad essi saranno comunque accessibili dal PC, utilizzando dei subscriber che sfruttano *image_transport*.
Per limitare l'impatto sulle performance del dispositivo, si è deciso di gestire l'iscrizione ai topic associati ai due tipi di trasporto, secondo una logica di mutua esclusione[4].
Al momento dell'avvio, l'applicazione utilizza di default il tipo: x264, lasciando disattivata la parte di compressione e pubblicazione relativa alla modalità: *compressed* .
Dopodiché, nel momento in cui un nuovo nodo effettua la subscribe su un *topic* associato a una dei due tipi di trasporto, le fasi di conversione e pubblicazione dell'altro verranno automaticamente sospese.

4.8.1 Pubblicazione immagini in formato JPEG

Una volta chiamato il metodo onNewRawImage, al suo interno è necessario verificare, attraverso un blocco condizionale, qual è il tipo di trasporto attualmente utilizzato. Nel caso in cui esso coincida con il tipo: *compressed*, partendo dalle singole immagini contenute all'interno dell'*array* di byte ricevuto come parametro in ingresso, queste vengono convertite dal formato NV21 al formato: YUV. Dopodiché, queste vengono compresse nel formato JPEG è inserite direttamente all'interno del campo *data* del messaggio: *sensor_msgs/CompressedImage*. A questo punto, viene effettuata una verifica sul valore dell'angolo di rotazione (rappresentato dall'attributo: *angle*, modificabile tramite ROS-*service*). Se quest'ultimo è uguale a zero, le immagini non necessitano di essere ruotate, pertanto è possibile passare direttamente alla fase di pubblicazione. Invece, nel caso in cui l'angolo sia diverso da zero, le singole immagini compresse vengono messe all'interno di un oggetto di tipo *Bitmap*. Una volta creato quest'ultimo, ad esso viene applicata una rotazione di un angolo pari al valore dell'attributo *angle*. A questo punto l'immagine ottenuta viene nuovamente compressa.
In entrambe le situazioni in cui viene utilizzata, la funzione di compressione, oltre a un oggetto di tipo *stream* (in cui verranno inseriti i nuovi dati), accetta un parametro chiamato *quality*, introdotto in precedenza. Quest'ultimo consente di regolare il "livello di compressione" utilizzato dall'algoritmo JPEG; anche questo parametro è modificabile dall'utente, grazie all'utilizzo del ROS *Service*: *"set_quality"*. Una volta ottenuto lo stream contenente l'immagine compressa, questo viene nuovamente assegnato al campo: *data*, del messaggio: *sensor_msgs/CompressedImage*, che verrà infine

[4] a meno che la codifica H.264 non sia stata disabilitata nella finestra di dialogo iniziale. In questo caso, la modalità compressed sarà l'unica disponibile.

pubblicato sul topic: *deviceName/android/camera/image/compressed*, utilizzando il publisher ad esso associato.

4.8.2 Streaming video H.264 con FFMPEG

Sebbene l'invio di immagini in formato JPEG risulti particolarmente vantaggioso, in termini di banda occupata, rispetto all'utilizzo di immagini non compresse, questo presenta una serie di limitazioni nel momento in cui la banda a disposizione diminuisce ulteriormente o quando aumenta la risoluzione delle immagini in ingresso.
Il primo caso si verifica nel momento in cui la distanza del dispositivo rispetto all'access-point aumenta, questo determina una diminuzione della banda che a sua volta porta a una diminuzione del frame-rate e ad un aumento della latenza, fenomeni che oltre un certo limite rendono inutilizzabile l'applicazione per un controllo real-time. Ciò è dovuto anche al fatto che con l'aumento della distanza aumenta anche il numero di pacchetti persi o corrotti durante la trasmissione. Ne consegue che a questo punto entrano in funzione i meccanismi di ritrasmissione caratteristici del protocollo TCP [5], ciò comporta l'attesa dei vari time-out nonché dei successivi segnali di *acknoledgement*.
L'aumento della risoluzione delle immagini provoca anch'esso una diminuzione del frame-rate. In questo caso, essa è dovuta alla maggiore quantità di tempo richiesta per effettuare la compressione, che si traduce nella pubblicazione di un minor numero di messaggi: *sensor_msgs/CompressedImage*, nell'unità di tempo.
A causa di queste limitazioni, oltre all'invio di immagini compresse in formato JPEG, si è deciso di aggiungere una classe che fosse in grado di effettuare la conversione delle immagini acquisite con la fotocamera in uno stream video compresso, utilizzando un codec H.264. Inizialmente, si è deciso di ricorre alle funzioni offerte dalla suite di librerie FFMPEG, con un utilizzo analogo a quello già visto per il *publisher* definito nel package: *x264_ image_transport*.
Per poter utilizzare FFMPEG su Android, è stato necessario includere all'interno del progetto, le varie librerie compilate in codice nativo. Una volta terminata la configurazione e verificato il funzionamento, si è riscontrato che questa soluzione, sebbene migliori le performance in termini di banda occupata rispetto al tipo *compressed*; non risolve i problemi legati all'abbassamento del frame-rate e all'aumento della latenza (due dei motivi principali per qui era stata introdotta). Ciò è dovuto al fatto che l'encoder H.264, definito all'interno di libavcodec [6], viene attualmente realizzato puramente in software (funzioni C definite ad alto livello che non fanno uso di particolari acceleratori hardware) [7]. Perciò i tempi necessari per la compressione aumentano con la risoluzione delle immagini in ingresso.

[5] occorre precisare che la versione di ROS attualmente disponibile su Android non supporta UDPROS
[6] Libreria compresa nel progetto FFMPEG, contenente l'implementazione dei vari codec
[7] Attualmente infatti, le librerie FFMPEG sfruttano l'API di basso livello: *MediaCodec*, solo per l'implementazione dei *decoder*. Sito di riferimento: [11]

Infine, l'utilizzo di librerie native è risultato essere abbastanza problematico, in quanto rende l'applicazione meno flessibile rispetto alla varietà di piattaforme su cui può essere installata. Questo è dovuto alla necessità di dover compilare più versioni, ciascuna specifica per una determinata architettura. Inoltre, come effetto secondario, ciò determina un aumento considerevole nelle dimensioni finali del file apk. Per tali motivi si è deciso di archiviare questa soluzione.

4.8.3 Streaming video H.264 con MediaCodec

Come si è visto, la soluzione basata su FFMPEG comporta alcune problematiche legate all'utilizzo delle librerie native e a un'implementazione poco performante dell'encoder. Ciò la rende inadatta, in termini di tempistiche di compressione, per la trasmissione in tempo reale di un flusso video. La classe *H264Encoder* risolve queste problematiche, sfruttando le funzionalità offerte dalla Platform API ufficiale definita per i sistemi Android: *MediaCodec*.

MediaCodec
Questa classe, appartenente al package ufficiale: *android.media.MediaCodec*, viene utilizzata per accedere, attraverso delle funzioni che lavorano a basso livello, alle componenti hardware dedicate, presenti oggi in quasi tutti i dispositivi considerati compatibili con Android. In questo modo è possibile aumentare le prestazioni rispetto alle versioni equivalenti realizzate puramente in software, allo stesso tempo ciò consente di gravare meno sul lavoro complessivo svolto dalla CPU.
Un codec ottenuto attraverso questa classe costituisce un processo autonomo, le cui funzionalità risultano accessibili attraverso l'utilizzo di una serie di buffer. Questi ultimi sono a loro volta organizzati in code di input e di output. Una classe che necessita di utilizzare un determinato codec, dopo averlo inizializzato e configurato, può sottomettere dei nuovi dati a quest'ultimo estraendo un buffer dalla coda di input e copiando al suo interno i dati da elaborare. Dopodiché il codec, una volta terminata la fase di elaborazione, inserisce i dati ottenuti all'interno della coda di output, dalla quale il processo interessato potrà poi estrarli.

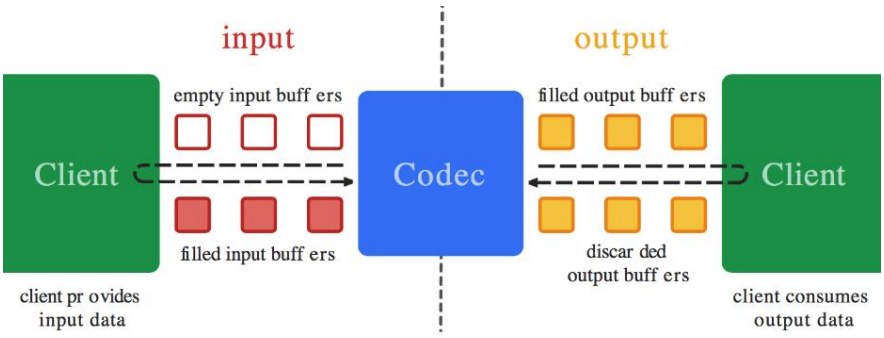

Figura 12 schema di funzionamento di un codec definito con MediaCodec [10]

Il ciclo di vita del codec si svolge passando attraverso una serie di fasi rappresentabili mediante l'introduzione del seguente diagramma di stato:

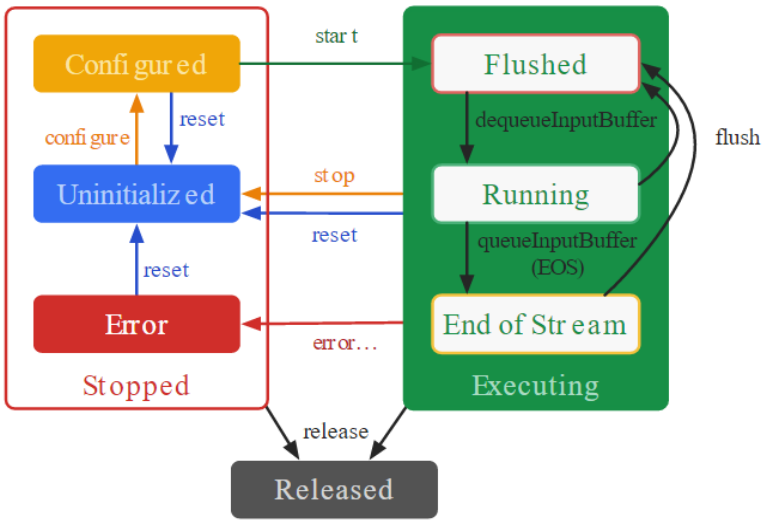

Figura 13 diagramma di stato associato a un codec ottenuto utilizzando la classe MediaCodec [10]

È perciò necessario che una qualsiasi classe che intenda definire un codec di questo tipo tenga ben presente l'insieme degli stati e delle transizioni possibili, definite all'interno diagramma, al fine di poter: configurare, utilizzare e infine terminare il codec nel modo corretto, strutturando il codice di conseguenza. Una classificazione possibile sulla base dell'utilizzo consiste nella differenziazione tra modalità sincrona e asincrona, nel primo caso tendenzialmente il processo client esegue un ciclo dove periodicamente si verifica la disponibilità di buffer vuoti nella coda di input e parallelamente la disponibilità di buffer pieni nella coda di output.
Per quanto riguarda la modalità asincrona, questa prevede un passaggio diretto dallo stato *configured* allo stato *running*, in questo caso il codec interagisce con il client attraverso l'utilizzo di funzioni *callback*.

H264Encoder

La classe *H264Encoder* sfrutta le funzionalità offerte da *MediaCodec*. In questo caso l'*encoder* ottenuto viene utilizzato in modalità asincrona. La prima operazione che è necessario svolgere, prima di poter passare alla fase di utilizzo, è la configurazione dell'*encoder*. Per fare ciò viene utilizzata un'istanza della classe *MediaFormat*. All'interno di questo oggetto vengono inserite le informazioni necessarie per il corretto funzionamento del codec, ovvero: larghezza e altezza dell'immagine, frame-rate, target bitrate, GOP e formato dei pixel. Successivamente è necessario inizializzare un'istanza di *NV21Convertor [18]*, derivata dall'omonima classe della libreria *libstreaming [18]*. Il suo scopo è quello di convertire il formato dei pixel, utilizzato per le immagini in ingresso, da NV21 (che è lo standard utilizzato all'interno di Android) in uno dei

formati YUV420, supportato dal dispositivo e utilizzato dall'*encoder*. Questo formato varia da un costruttore all'altro e spesso anche tra modelli diversi dello stesso produttore. Pertanto, è necessario effettuare una serie di tentativi per identificare quello corretto, poiché non è possibile stabilire a priori quale formato si debba utilizzare.
A questo punto, dal momento che si è deciso di utilizzare la modalità asincrona, è necessario assegnare all'*encoder* un oggetto che appartenga alla classe *MediaCodec.Callback*. Quest'ultima consente di definire i metodi che dovranno essere chiamati nel momento in cui sarà disponibile un nuovo buffer di input per la scrittura o di output per la lettura. Una volta terminata con successo la fase di configurazione è possibile avviare l'*encoder* chiamando il metodo: *start*, associato a quest'ultimo.
La classe *H264Encoder* viene utilizzata come attributo all'interno di un'istanza di *CameraActivity*, che come descritto in precedenza, lavora anch'essa in modalità asincrona, ovvero viene chiamata una *callback* ogniqualvolta sono disponibili dei nuovi dati acquisiti dalla fotocamera. È perciò necessario gestire le chiamate asincrone provenienti da entrambe le classi, in quanto, al fine di garantire il corretto funzionamento complessivo, sì richiederebbe una loro esecuzione in sequenza. Tuttavia, le due chiamate risultano tra loro indipendenti e pertanto non è possibile sincronizzarle.
Per risolvere il problema, tra gli attributi della classe *H264Encoder*, è stato aggiunto un oggetto di tipo *Map* chiamato: *availableInputBuffers*. Questo ha lo scopo di garantire il corretto funzionamento, nonostante le varie chiamate relative alle *callback* delle due classi siano tra loro disaccoppiate.
Dal momento che il numero di buffer di input che un codec può gestire è limitato (massimo 4), la mappa in questione viene gestita come un buffer circolare che utilizza un metodo di transito di tipo FIFO.

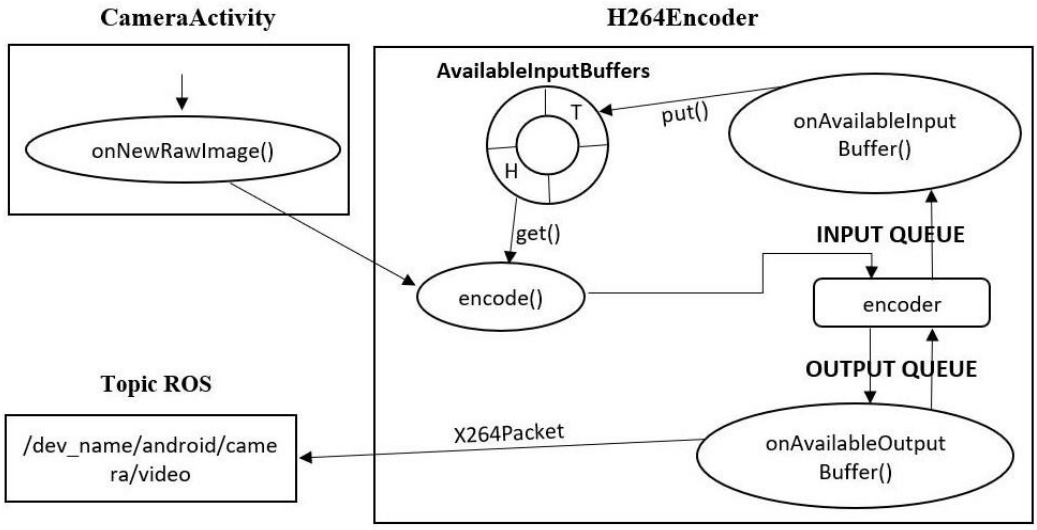

Figura 14 schema concettuale che descrive il funzionamento asincrono delle classi CameraActivity e H264Encoder

Ogniqualvolta l'*encoder* rende disponibile un nuovo buffer in scrittura, il metodo *onAvailableInputBuffer* inserisce quest'ultimo all'interno della mappa insieme a un indice,

il quale si riferisce alla posizione del buffer corrente all'interno della coda di input. Parallelamente, nel momento in cui vengono acquisiti dei nuovi dati dalla camera e viene chiamata la *callback*: *onNewRawImage*, all'interno di *CameraActivity*; questa a sua volta invocherà un metodo chiamato: *encode*, della classe *H264Encoder*. All'interno di quest'ultimo si verifica se nella mappa *availableInputBuffers* è presente almeno una *entry*, in tal caso viene selezionata la meno recente tra quelle presenti. Dopodiché, i dati relativi all'immagine (ricevuti come parametro dal metodo: *encode*), una volta convertiti grazie all'istanza della classe *NV21Convertor*, introdotta in precedenza, vengono inseriti all'interno del buffer prelevato dalla mappa. Dopodiché, quest'ultimo verrà nuovamente inserito all'interno della coda di input dell'*encoder* per poter essere elaborato.
Una volta terminata la fase di elaborazione, i nuovi dati codificati vengono inseriti in un buffer di uscita, rendendoli disponibili nella coda di output. A questo punto viene chiamato il metodo *onOutputBufferAvailable* definito all'interno dell'oggetto *MediaCodec.Callback*. In questo metodo viene eseguita la pubblicazione sul *topic* ROS: */deviceName/android/camera/video/x264,* attraverso un *publisher* specializzato per l'utilizzo dei messaggi: *x264_video_transport/X264Packet*.
L'utilizzo di questa soluzione ha consentito di migliorare notevolmente la qualità dello stream video, portando a risultati superiori sia rispetto al caso *compressed* che alla precedente versione realizzata con FFMPEG.

Uso congiunto con il package x264_video_transport

Come anticipato nel capitolo precedente, x264_video_transport, oltre ad introdurre alcuni aggiornamenti riguardanti le funzioni FFMPEG utilizzate nel package: x264_image_transport; è stato pensato per garantire una maggiore compatibilità e un supporto aggiuntivo rispetto all'esigenze dell'applicazione Android. Come si è visto, attraverso i *ROS-service*, l'utente può modificare a *run-time* una serie di parametri legati alla camera, uno di questi è la risoluzione. Quest'ultima rappresenta il parametro più critico sia dal punto di vista del publisher che da quello del subscriber; in quanto richiede che tanto l'encoder quanto il decoder, nel momento in cui essa cambia, debbano essere prima bloccati e riconfigurati e solo a questo punto potranno essere riavviati. Un altro ROS-service che richiede il supporto del package x264_video_transport per il suo funzionamento è quello relativo alla rotazione delle immagini. Infatti, mentre nel caso *compressed*, questa operazione viene fatta direttamente sul dispositivo (con tutti gli svantaggi che comporta), nel caso "x264" è stato possibile spostare questa fase all'interno del codice relativo al *subscriber*.

4.8.4 ROS-Service associati alla camera

Oltre ai vari topic attraverso i quali l'applicazione è in grado di pubblicare le informazioni relative ai sensori del dispositivo su cui è in esecuzione, quest'ultima fornisce anche dei ROS-Service che consentono di modificare alcuni parametri relativi al comportamento della fotocamera.

L'utilizzo di questi servizi può essere utile in quelle situazioni in cui l'utente vuole intervenire sul comportamento dell'applicazione. In questo caso, attraverso i ROS-service è possibile intervenire sulla qualità delle immagini trasmesse o modificare la loro risoluzione; questo può essere utile ad esempio in quelle situazioni in cui la banda a disposizione è bassa, e per favorire una trasmissione più fluida o con minor latenza, può essere necessario ridurre entrambi questi parametri. Inoltre, è possibile: ruotare le immagini, passare dalla camera posteriore a quella anteriore e attivare il flash.
Per fare ciò, tra gli attributi della classe *CameraActivity*, è necessario definire delle implementazioni specializzate dell'interfaccia: *ServiceResponseBuilder*. Ognuna di queste include il metodo: "build", il quale riceve due parametri secondo la classica coppia: *Request-Response*. Infatti, il primo parametro è associato alla richiesta del *client*, in base alla quale si potrà variare lo stato dell'applicazione. A sua volta, l'esito di questa operazione determinerà il valore della risposta, che verrà inserita all'interno del secondo parametro. Quest'ultimo sarà inoltrato nuovamente verso il *client*.
Una volta definita l'implementazione di uno specifico *ServiceResponseBuilder*, quest'ultimo dovrà essere assegnato a un server, affinché il ROS-Service ad esso associato diventi operativo. Questa operazione viene svolta all'interno della classe *Monitoring* (durante la configurazione dell'istanza *CameraActivity*) utilizzando il metodo *newServiceServer* (appartenente alla classe *ConnectedNode*). Anche in questo caso, il metodo accetta due parametri: il primo è una stringa contenente il nome del ROS-*Service* che si sta creando, il secondo è l'oggetto *ServiceResponseBuilder* associato ad esso. Di seguito verranno presentati i vari servizi definiti all'interno dell'applicazione:

rotateImage

questo servizio consente di modificare l'angolo di rotazione dell'immagine visualizzata attraverso il *topic*, rispetto all'orientamento originale della camera. Il servizio modifica il valore dell'attributo: *angle*, definito all'interno della classe *CameraActivity*. Anche in questo caso gli effetti del servizio variano a seconda della modalità di trasporto utilizzata per la trasmissione delle immagini.
Nel caso del tipo *compressed*, come si è già visto, l'attributo angle viene utilizzato come parametro per la funzione responsabile della rotazione dell'oggetto di tipo Bitmap associato all'immagine. Nel caso x264 il valore dell'attributo angle viene inserito all'interno di un omonimo campo all'interno del messaggio *x264Packet*, questo campo verrà poi utilizzato dal subscriber definito nel package *x264_video_transport* per eseguire la rotazione.
L'ultima soluzione porta notevoli vantaggi in termini di frame-rate e latenza rispetto a quella utilizzata nel caso *compressed* [8], in quanto essa demanda l'esecuzione della rotazione al *subscriber*. Quest'ultimo disponendo normalmente di una maggiore potenza

[8] al quale non è stato possibile applicare la stessa soluzione in quanto questa richiederebbe la modifica del messaggio standard: sensor_msgs/CompressedImage, oltreché del subscriber ad esso associato, definito nel package: image_transport_plugins, rendendoli incompatibili con le versioni preesistenti. A questo punto chiunque volesse utilizzare l'applicazione dovrebbe anche scaricare la versione modificata del plugin.

di calcolo oltreché di librerie più adatte, sarà in grado di eseguire l'operazione più velocemente, evitando che questa possa avere un impatto sulle performance.

setQuality

Questo servizio consente di modificare, in fase di esecuzione, la qualità delle immagini trasmesse, attraverso l'utilizzo dell'attributo: *quality*, appartenente alla classe *CameraActivity*. I valori che questo campo può assumere sono compresi tra 1 e 99 e suo significato varia a seconda del tipo di trasporto utilizzato in un dato momento.
Se si considera il tipo *compressed*, l'attributo *quality* consente di modificare il parametro di quantizzazione utilizzato durante la compressione nel formato JPEG. Mentre, se si sta utilizzando la modalità di trasporto x264, il parametro *quality* rappresenta un coefficiente, compreso zero e uno, utilizzato per regolare il valore del parametro: *MediaFormat.KEY_BIT_RATE*, specificato durante la configurazione del codec H.264. Quest'ultimo parametro consente di impostare il cosiddetto: *"target bitrate"*, ovvero un valore di soglia che i dati in uscita dall'*encoder* non possono superare, perciò sulla base di quest'ultimo i singoli frame risulteranno più, o meno compressi. L'attributo *quality* consente di variare il valore di questo parametro da un minimo uguale a zero[9] a un massimo di 7 Mbps.

setResolution

Attraverso questo servizio è possibile modificare la risoluzione delle immagini trasmesse. Per fare ciò è necessario rilasciare la fotocamera ed eventualmente bloccare l'esecuzione dell'encoder H.264. Successivamente utilizzando il metodo *getSupportedPreviewSizes*, è possibile ottenere la lista delle risoluzioni supportate dalla camera attualmente selezionata. Poiché l'ordine degli elementi nella lista non è noto a priori, è necessario stabilire un criterio di ordinamento. Perciò si è deciso di ordinare le varie risoluzioni in modo decrescente.
A questo punto il client può decidere se aumentare o diminuire la risoluzione, inviando rispettivamente i valori 1 o -1. Seguirà la risposta del server. Questa sarà costituita da una stringa che conterrà il nuovo valore utilizzato, seguito dall'elenco di tutte le risoluzioni supportate, ordinate secondo il criterio descritto sopra.

switchCamera

Consente di modificare la camera utilizzata selezionandone un'altra tra quelle disponibili, per fare ciò è necessario rilasciare tutte le risorse acquisite relative alla camera, in quanto generalmente non saranno più valide e il loro possesso in determinati casi impedisce l'acquisizione da parte della nuova camera selezionata. Se l'operazione va a buon fine, il servizio restituisce una stringa che può variare in base al numero di fotocamere disponibili sul dispositivo, nel caso in cui fosse presente solo una sola

[9] Il valore effettivo del bitrate medio in uscita, ovviamente non potrà mai essere zero; in questo caso viene applicata la massima compressione possibile nei limiti consentiti dall'immagine.

camera, viene comunicato all'utente che non è possibile selezionarne un'altra, nel caso più comune in cui son presenti due fotocamere: il metodo restituisce *"front"* o *"back*" a seconda di quale viene selezionata. Infine, nel caso in cui le fotocamere siano più di due, viene restituito il valore dell'attributo: camera_id, ovvero un intero che partendo da zero rappresenta il numero associato alla camera selezionata.

4.9 Comparto telefonico (GSM, CDMA, EVDO, LTE)

All'interno della classe *PhoneStateProvider* ci si occupa di acquisire e pubblicare i dati provenienti dal comparto telefonico. Nello specifico, la classe fornisce informazioni sull'intensità del segnale per quanto riguarda: GSM, CDMA, EVDO e LTE. Per accedere al servizio di telefonia è necessario istanziare un *TelephonyManager,* inoltre, è necessario che *PhoneStateProvider* a sua volta estenda la classe *PhoneStateListener* al fine di poter ricevere le varie chiamate asincrone a seguito di variazioni del segnale. Anche in questo caso, analogamente a quanto fatto con la classe *BatteryStateProvider*, viene creato un task secondario per garantire l'aggiornamento continuo dei messaggi sul *topic*, per questo motivo acquisizione e pubblicazione dei dati avvengono seguendo due percorsi differenti.
Il primo caso considerato è quello dell'acquisizione asincrona: la classe *PhoneStateProvider* estendendo a sua volta la classe *PhoneStateListener*, può ridefinire il metodo: *onSignalStrengthsChanged* che viene invocato ogniqualvolta viene rilevata una variazione nell'intensità del segnale.
Il metodo *onSignalStrengthsChanged* riceve come parametro un oggetto di tipo *SignalStrength*, che al suo interno contiene i nuovi valori. Analizzando il contenuto di quest'oggetto (con l'utilizzo dei metodi: *toString* e *split*), si ottengono i seguenti dati:

[1] mGsmSignalStrength	*[2] mGsmBitErrorRate*	*[3] mCdmaDbm*
[4] mCdmaEcio	*[5] mEvdoDbm*	*[6] mEvdoEcio*
[7] mEvdoSnr	*[8] mLteSignalStrength*	*[9] mLteRsrp*
[10] mLteRsrq	*[11] mLteRssnr*	*[12] mLteCqi*
[13] mTdScdmaRscp		

Nel secondo caso invece, un task secondario esegue ogni cinque secondi il metodo: *run*, dell'oggetto *tel_runnable*. All'interno di quest'ultimo viene chiamato il metodo *getTelephoneMsg* attraverso il quale è possibile acquisire esplicitamente nuovi dati dal comparto telefonico e organizzarli all'interno del messaggio ROS. Per richiedere le informazioni sullo stato corrente del segnale si ricorre al metodo *getAllCellInfo* della classe *TelephonyManager*. Infine, una volta terminata l'acquisizione, il messaggio contenente i nuovi dati viene utilizzato come valore di ritorno per la funzione chiamante (*run*).
Nella fase finale, in entrambi i casi descritti (sincrono e asincrono), oltre a effettuare il *logging*, vengono chiamati i metodi: *updateInterface* e *pubPhoneValues*; il primo

consente di aggiornare l'interfaccia utente con in nuovi valori, utilizzando l'oggetto textView associato alla classe *PhoneStateProvider*, il secondo si occupa di pubblicare gli stessi dati sul topic associato ad essi:

- *Publisher<senspub_msgs.telephone> tel_publisher*
  - *deviceName/android/telephony*

Occorre notare che poiché non esiste un messaggio standard definito per questi dati, è stato necessario creare un nuovo tipo, definito ad-hoc durante lo sviluppo dell'applicazione e chiamato: *senspub_msgs/telephone*. La struttura di quest'ultimo è in grado di contenere tutte le informazioni messe a disposizione dall'API (ovvero quelle mostrate a nell'elenco della pagina precedente). Inoltre, è stato definito un nuovo oggetto appartenente alla libreria ROS, di tipo: *PublisherListener*. Questo ultimo consente di definire un comportamento specifico in relazione a un determinato evento riguardante il publisher a cui è associato. Nell'ambito di questa classe, tra i metodi definiti per questo tipo di oggetto si sfrutta: *onNewSuscriber*, per fare in modo che ogniqualvolta un nuovo nodo si iscrive al *topic* associato, venga subito effettuata l'acquisizione di nuovi dati, evitando di dover attendere chiamate asincrone o task secondari, per poter effettuare la pubblicazione successiva.

4.10 Wi-Fi

Durante il monitoraggio remoto può essere importante conoscere lo stato del segnale Wi-Fi con il quale si sta lavorando, allo scopo di poter adattare determinati parametri, in base alle esigenze che possono presentarsi durante la fase di utilizzo.
Innanzitutto, occorre istanziare un oggetto di tipo *WifiManager*, necessario per garantire l'accesso alle informazioni provenienti dall'antenna Wi-Fi oltre a tutte le strutture dati, metodi e costanti, necessarie per il loro utilizzo. Dopodiché, come per la classe precedente, sono state definite due modalità di acquisizione: sincrona e asincrona.
Nel primo caso viene utilizzato un oggetto di tipo *Runnable* per consentire l'esecuzione di un task secondario che periodicamente acquisisce nuovi dati. Nel secondo caso, il metodo: *onReceive*, viene chiamato ogniqualvolta si verifica una variazione nel valore RSSI (received signal strength indicator) relativo al Wi-Fi.
Entrambe le modalità convergono nell'utilizzo del metodo: *Wifi*. All'interno di quest'ultimo viene aggiornata l'interfaccia dell'applicazione, successivamente si genera un nuovo messaggio ROS di tipo *"senspub_msgs/wifi"* e si copiano i dati appena acquisiti all'interno quest'ultimo. Dopodiché viene effettuato il salvataggio degli stessi sul file di log e infine si ha la pubblicazione del messaggio appena creato sul relativo topic:

- *Publisher<senspub_msgs.wifi> wifi_publisher*
  - */deviceName/android/wifi*

Anche in questo caso, in assenza di un messaggio standard in grado di contenere i nuovi dati, è stato necessario definire il tipo: *wifi*, la cui struttura è stata realizzata sulla base delle informazioni rese disponibili attraverso le istanze degli oggetti forniti dall'API Android. All'interno della classe *WifiProvider* si definisce anche un oggetto di tipo: *ServiceResponseBuilder* che, come si è già visto nel caso della fotocamera, consente di definire il comportamento di un ROS-service. In questo caso, il servizio in questione consente a un generico client di richiedere al server il valore corrente relativo al RSSI.

4.11 Logging locale

Durante l'utilizzo dell'applicazione è fondamentale poter tener traccia di ciò che viene acquisito dal dispositivo con l'intento di poter svolgere un'analisi più dettagliata a posteriori e di verificare lo stato dei sensori e dell'applicazione stessa nei casi in cui venisse a mancare la connessione.
La classe Logger attraverso il metodo statico: *getLogger*; consente di configurare un nuovo oggetto di tipo *Log4J*, definito in una libreria esterna, all'interno del metodo *configureLog4J*. In quest'ultimo metodo, si definisce il percorso di salvataggio all'interno del file-system. Per fare ciò, si verifica in primo luogo se è possibile utilizzare la memoria esterna (scheda micro-SD), in caso contrario si utilizza la memoria interna del dispositivo. Successivamente viene assegnato un nome al file di log secondo il seguente formato: *logfile_{current_timestamp}.log*, viene inoltre definito il formato da utilizzare per la data e ora stampate per ciascuna riga del file.
Infine, si definiscono la dimensione massima del backup e del file di log, quest'ultima espressa in byte. Ognuna delle classi presenti nell'applicazione definisce il proprio oggetto di tipo *Logger*, che ha il compito di tener traccia di tutte le operazioni svolte dalla classe e di memorizzare tutte le informazioni da essa acquisite.

4.12 GNSS Raw Measurements

Le prestazioni tipiche dei dispositivi mobili disponibili oggi sul mercato sono in grado di fornire una stima della posizione a cui viene associato un valore di accuratezza di alcuni metri, ma che può aumentare (fino a un ordine di grandezza) in quelle condizioni in cui il segnale o la copertura satellitare risultano essere particolarmente deboli. Tali limitazioni per il momento hanno consentito di distinguere nettamente i dispositivi di tipo professionale da quelli di uso comune. Tuttavia, negli ultimi anni, con l'introduzione di nuovi chipset multi-costellazione e a doppia frequenza sempre più performanti all'interno degli smartphone, è diventata sempre più concreta la possibilità di aumentare l'accuratezza di questi ultimi fino di arrivare, in un prossimo futuro, a livelli di precisione inferiori al metro.

Gli smartphone e i dispositivi indossabili potrebbero essere impiegati per usi semi-professionali, favorendo la nascita di una nuova gamma di applicazioni fino ad oggi ritenute irrealizzabili in questo settore. Allo stesso tempo ne gioverebbero anche le applicazioni, come quella presentata in questo capitolo, che hanno come obbiettivo quello di permettere il monitoraggio di dispositivi quali i rover. Questi ultimi spesso devono potersi muovere con precisione in ambienti eterogenei, che a seconda dei casi possono essere molto ampi o parecchio ristretti e perciò si ha la necessità di avere un servizio il più possibile affidabile e accurato. A questo punto, l'unica limitazione rimasta rispetto a tali miglioramenti era costituita dall'assenza di un'interfaccia software adeguata all'interno dei sistemi operativi *mobile* e dalla conseguente mancanza di applicazioni in grado di sfruttare queste opportunità.
Nel maggio 2016, Google ha annunciato la disponibilità di una nuova API per la localizzazione introdotta a partire da Android 7 (Nougat), caratterizzata dalla possibilità di poter accedere ai dati grezzi utilizzati dal chipset GNSS a livello di sistema operativo: i *raw-measurement*. In questo modo per la prima volta, gli sviluppatori hanno avuto accesso alle misurazioni acquisite direttamente dal chipset e ai messaggi di navigazione all'interno di dispositivi destinati al mercato di massa (infatti, questi dati erano già accessibili da diverso tempo su apparecchiature professionali di fascia alta, ma mai sugli smartphone).
Per poter utilizzare i raw-measurement, è necessario prima assicurarsi di avere uno smartphone in grado di catturare tali dati[10]. A seconda del dispositivo, questi possono includere tutte o solo alcune delle informazioni che normalmente costituiscono i raw-measurements.

4.12.1 Vantaggi introdotti

Ci sono diversi vantaggi nell'utilizzo dei raw-measurement sugli smartphone. Il loro impiego può portare a un aumento delle prestazioni, in quanto apre le porte a tecniche di elaborazione GNSS avanzate, che fino ad ora erano limitate ai soli ricevitori GNSS di tipo professionale. Sebbene in condizioni normali la posizione calcolata con i raw-measurements potrebbe non essere ottimale se confrontata con i valori PVT (Position, Velocity, Time) calcolati automaticamente dal chipset, in alcuni casi, quando si applicano opportune correzioni esterne, l'utilizzo di dati "grezzi" del GNSS può portare a una soluzione molto più accurata. Diverse aree di applicazione si distinguono per trarre profitto da questa maggiore precisione; tra queste ad esempio: la pubblicità basata sulla posizione, la realtà aumentata o tecniche di *cooperative-positioning*. I raw measurement consentono anche di ottimizzare le soluzioni Multi-GNSS basate sull'utilizzo di diverse frequenze e sull'uso simultaneo di satelliti appartenenti a costellazioni differenti. Inoltre, è possibile selezionare i satelliti (o meglio i dati da essi forniti) in base alle loro prestazioni e caratteristiche specifiche.

[10] la maggior parte degli smartphone fabbricati a partire dal 2016 e venduti già con la versione Android 7 (o superiore) dovrebbero essere già in grado di acquisirli

La disponibilità dei raw-measurements riscuote un certo interesse anche dal punto di vista dell'innovazione tecnologica. Potendo effettuare test più dettagliati svolti con algoritmi che lavorano in post-processing. Con Android 7 gli utenti possono utilizzare l'API: *android.location*, per accedere ai dati non elaborati necessari per calcolare gli pseudorange e decodificare i messaggi di navigazione. Questi dati possono essere sfruttati per esplorare nuovi algoritmi e applicazioni.

4.12.2 Le limitazioni hardware

Come si è visto, i raw-measurement consentono di utilizzare tecniche di posizionamento avanzate che permettono di ridurre gli errori e migliorare la precisione del posizionamento. Tuttavia, gli attuali modelli di smartphone, oltre alla necessità di una maggiore precisione del servizio di localizzazione, devono anche tener conto di altri fattori, tra questi ad esempio: l'ottimizzazione del consumo della batteria, l'esperienza utente e il design. Tutti elementi che influiscono sia sulla selezione delle componenti hardware che sulla scelta degli algoritmi (software). Inoltre, si consideri che normalmente i telefoni sebbene più performanti rispetto al passato, utilizzano comunque componenti a basso costo e tra questi, i più critici per la navigazione sono l'antenna e gli oscillatori utilizzati per la misurazione degli intervalli temporali. Queste scelte hanno un certo impatto sul funzionamento sistema GNSS e questo a sua volta si riflette sulla qualità e l'affidabilità dei raw-measurement generati.
Un chipset GNSS in continuo funzionamento tende ad esaurire velocemente la batteria. È possibile implementare diverse tecniche per mantenere bassi i consumi, ad esempio cambiando gli oscillatori utilizzati durante la fase di acquisizione e quella di inattività del GNSS (*Duty-Cycle*), tuttavia queste possono influire sulle prestazioni.
Per quanto riguarda l'antenna, quella tipicamente utilizzata nei telefoni cellulari è abbastanza economica, ma è comunque adatta a qualsiasi costellazione che utilizzi la banda di frequenza L1. Tuttavia, fattori legati alla qualità, come la polarizzazione lineare (invece che circolare) e la direzionalità del modello di radiazione, possono portare a una forte attenuazione del segnale captato. Oltre a queste criticità, occorre notare che spesso la posizione dell'antenna all'interno del dispositivo è dettata quasi unicamente da logiche design. Ciò può condurre alla scelta di posizioni non ottimali in cui l'interazione con la mano dell'utente indebolisce ulteriormente la ricezione dei segnali GNSS. Questi sono solo alcuni degli esempi che dimostrano come le scelte di progettazione hardware influiscano sulla qualità dei raw-measurement generati, influenzando di conseguenza l'accuratezza della posizione ricavata a partire da essi.

4.12.3 Accedere ai raw-measurements utilizzando l'API di Android

Prima del rilascio di Android 7, gli sviluppatori potevano accedere alle informazioni relative al posizionamento attraverso le classi e i metodi forniti dall'interfaccia: *android.gsm.location*. Questa API permetteva di ottenere informazioni posizionali

semplificate, utilizzando un provider di posizione unificata, ovvero calcolata utilizzando i dati provenienti da diverse sorgenti oltre al chipset GNSS come: Wi-Fi, bluetooth, rete mobile, accelerometro, giroscopio, magnetometro ecc.. cercando di fornire, allo stesso tempo, un servizio ottimizzato per il risparmio della batteria. Ciò tuttavia limitava parecchio la libertà degli sviluppatori, che si ritrovavano a dover utilizzare dati di cui non si conosce esattamente l'origine o gli algoritmi utilizzati per calcolarli. Fino alla versione Android API 23 (inclusa) l'interfaccia: *android.gsm.location*, si limitava a fornire i seguenti dati:

- informazioni satellitari GPS
  - C/No
  - angolo azimutale
  - elevazione
  - se un particolare satellite è stato utilizzato o meno nel PVT
- indicazioni NMEA
- valori PVT (latitudine, longitudine, altitudine, velocita, timestamp UTC)

Gli utenti potevano inviare comandi di configurazione di base al chipset, incluso il riavvio/avvio di quest'ultimo o la pulizia dei dati A-GNSS (Assisted-GNSS [11]). Tuttavia, tutte le impostazioni di configurazione che includono ad esempio: le priorità nella scelta della costellazione GNSS e i diversi algoritmi utilizzati per il calcolo del PVT sono controllate dal chipset stesso. Quest'ultimo può essere visto come una black-box dove: acquisizione e tracciamento dei blocchi, decodifica del messaggio di navigazione, misurazioni di fase e generazione degli pseudorange, vengono fatti all'interno. L'utente può accedere solamente all'informazione elaborata, senza conoscerne i dettagli.
A partire da Android 7 (Nougat), sebbene sia stata mantenuta la retrocompatibilità con le versioni precedenti conservando tutte le funzionalità preesistenti, è stata introdotta una nuova API di localizzazione: *android.location*, che apporta delle innovazioni significative al servizio di posizionamento. Attraverso quest'ultima, gli sviluppatori hanno ottenuto l'accesso a nuove informazioni, utilizzabili attraverso delle nuove classi Android: *GnssNavigationMessage*, *GnssClock*, *GnssMeasurementEvent* e *GnssMeasurement*. Occorre specificare che gli pseudorange, necessari per il calcolo della posizione del ricevitore, non vengono forniti direttamente dalla nuova API, ma si rendono disponibili i parametri necessari per calcolarli. Come già visto nel paragrafo relativo alla localizzazione, le applicazioni accedono al servizio di posizionamento tramite la classe: *android.location.LocationManager*, specificando il provider (rete, Wi-Fi o GPS/GNSS) o utilizzando quello unificato (classico), a seconda dei criteri di accuratezza e potenza per cui esse sono pensate.
Nel momento in cui si utilizzano i raw-measurement, questa struttura può essere estesa definendo più provider di localizzazione personalizzati (basati ad esempio, su

[11] Utilizzano informazioni esterne allo scopo di ridurre il TTFF (Time To First Fix), consentendo di ottenere una stima della posizione pressoché immediata, anche in condizioni di scarsità del segnale (ad esempio all'interno di un edificio)

"pseudorange-smoothing" o sui dati GNSS differenziali). È importante specificare che in ogni caso, sebbene questi dati provengano direttamente dal chipset GNSS, non è ancora possibile l'accesso diretto al chipset stesso, perciò determinati parametri come la frequenza di pubblicazione dei dati attraverso le callback definite nell'API e i meccanismi di acquisizione non possono essere modificati dallo sviluppatore.
L'applicazione corrente, attraverso la classe *GNSSRawMeasurements*, è in grado di acquisire questi dati utilizzando le classi definite all'interno dell'interfaccia *android.location* e di pubblicarli attraverso i topic ROS. Per fare ciò, è stato necessario definire dei nuovi tipi di messaggi, adatti a trasportare le informazioni contenute all'interno dei raw-measurement forniti da Android:

- *senspub_msgs/GnssNavigation*
- *senspub_msgs/GnssClock*
- *senspub_msgs/GnssMeasurement*
- *senspub_msgs/GnssMeasurements.*

La nuova API di localizzazione consente l'acquisizione dei dati esclusivamente con l'utilizzo di un meccanismo asincrono. All'interno del costruttore della classe *GNSSRawMeasurements*, utilizzando i metodi di registrazione disponibili attraverso l'istanza della classe *LocationManager*, oltre a definire il comportamento associato alle variazioni di stato del GNSS, è possibile registrare le callback associate: all'acquisizione di un nuovo messaggio di navigazione e alla presenza di nuove misurazioni GNSS. Nel primo caso il metodo *onGnssNavigationMessageReceived* ottiene come parametro un'istanza della classe *GnssNavigationMessage*. A questo punto i dati contenuti all'interno della classe vengono inseriti all'interno del messaggio ROS associato, per poi essere pubblicati sul topic: */deviceName/android/gnss/navigation.*

```
public void onGnssNavigationMessageReceived(GnssNavigationMessage event) {
        …
        GnssNavigation navigation_msg = gnss_navigation_publisher.newMessage();
        …
        navigation_msg.setSvid(event.getSvid());
        navigation_msg.setType(event.getType());
        navigation_msg.setStatus(event.getStatus());
        navigation_msg.setMessageId(event.getMessageId());
        navigation_msg.setSubmessageId(event.getSubmessageId());
        navigation_msg.setData(stream.buffer().copy());
        …
        gnss_navigation_publisher.publish(navigation_msg);
        …
}
```

Figura 15 callback messaggi di navigazione

Per quanto riguarda l'acquisizione delle misurazioni GNSS, in questo caso la callback associata, riceve come parametro un oggetto di tipo: *GnssMeasurementEvent*. A quest'ultimo è associato un oggetto GnssClock e una lista di elementi di tipo: *GnssMeasurement*, ognuno di questi contiene i dati relativi a un singolo raw-measurement.

Anche in questo caso i dati devono essere inseriti all'interno dei relativi messaggi ROS. Il messaggio: *senspub_msgs/GnssMeasurements* possiede una struttura adatta a contenere tutti i dati presenti all'interno dell'istanza *GnssMeasurementEvent*. Innanzitutto, vengono copiati i dati relativi al clock, nel fare ciò è necessario utilizzare i vari metodi per la verifica della presenza di uno specifico campo, ognuno di essi restituisce un valore booleano: vero/falso, a seconda che il dato sia presente o meno nella misurazione. Si ricorda che a seconda del dispositivo sia i valori associati al clock che quelli associati ai raw-measurement possono essere incompleti (in base al tipo di chipset utilizzato o a limitazioni imposte dal costruttore), perciò è sempre necessario verificare la presenza di ogni singolo campo prima di tentare di leggerlo generando un'eccezione.

```
public void onGnssMeasurementsReceived(GnssMeasurementsEvent event) {
        …
        GnssMeasurements gnssMeasurements_msg = gnss_measurements_publisher.newMessage();

        GnssClock clock = event.getClock();
        senspub_msgs.GnssClock clock_msg = gnssMeasurements_msg.getGnssClock();

        //clock
        clock_msg.setHardwareClockDiscontinuityCount(clock.getHardwareClockDiscontinuity-
        Count());
        clock_msg.setTimeNanos(clock.getTimeNanos());

        if(clock.hasBiasNanos()){
            clock_msg.setBiasNanos(clock.getBiasNanos());
        }

        if(clock.hasBiasUncertaintyNanos()){
            clock_msg.setBiasUncertaintyNanos(clock.getBiasUncertaintyNanos());
        }

        if(clock.hasDriftNanosPerSecond()){
            clock_msg.setDriftNanosPerSecond(clock.getDriftNanosPerSecond());
        }
        …
```

Figura 16 callback GNSS measurements: Clock

Successivamente si procede con la copia dei singoli raw-measurement contenuti nella lista citata in precedenza. Per fare ciò, viene utilizzato un ciclo for all'interno del quale ogni elemento della lista viene copiato all'interno di un messaggio ROS: *senspub_msgs/GnssMeasurement*.
Anche in questo caso, per quegli elementi che possono non essere presenti, sono stati definiti degli appositi metodi di verifica che devono essere utilizzati prima di effettuare la lettura. Al termine di ogni ciclo il messaggio ROS contenente i nuovi dati viene inserito all'interno di un array definito come campo all'interno del messaggio: *senspub_msgs/GnssMeasurements*. Una volta aggiunti tutti gli elementi presenti nella lista, quest'ultimo viene pubblicato sul topic: */deviceName/android/gnss/measurements.*

```
…
Collection<GnssMeasurement> measurements = event.getMeasurements();

for(GnssMeasurement m : measurements){

   senspub_msgs.GnssMeasurement measurement_msg = messageFactory
                                        .newFromType(senspub_msgs.GnssMeasurement._TYPE);
    //measurements
    measurement_msg.setElapsedRealtimeNanos(SystemClock.elapsedRealtimeNanos());
    measurement_msg.setState(m.getState());
    measurement_msg.setSvid(m.getSvid());
    measurement_msg.setAccumulatedDeltaRangeState(m.getAccumulatedDeltaRangeState());
    measurement_msg.setConstellationType(m.getConstellationType());
    measurement_msg.setMultipathIndicator(m.getMultipathIndicator());
    measurement_msg.setAccumulatedDeltaRangeMeters(m.getAccumulatedDeltaRangeMeters());
    measurement_msg.setAccDeltaRangeUncertaintyMeters(m.getAccumulatedDeltaRangeUncertain-
tyMeters());
    …
    gnssMeasurements_msg.getGnssMeasurementsArray().add(measurement_msg);
}
gnss_measurements_publisher.publish(gnssMeasurements_msg);
…
```

Figura 17 callback GNSS measurements: Measurements

Come anticipato più volte in precedenza, l'API: *android.location*, sebbene consenta l'accesso ai raw-measurement, non prevede la possibilità di accedere direttamente ai parametri associati al chipset, in questo modo non è possibile conoscere la frequenza con cui vengono generati i raw-measurement, che in generale potrà variare a seconda del dispositivo. Inoltre, non è possibile intervenire attivamente sull'acquisizione, essendo il meccanismo asincrono l'unico disponibile.

4.12.4 Logging locale

Parallelamente alla pubblicazione delle informazioni contenute nei messaggi ottenuti a partire dai raw-measurements sui topic ROS, all'interno delle varie callback descritte in precedenza viene anche fatto il logging locale degli stessi dati.
Per questa operazione si utilizza un file separato rispetto a quello destinato per il log principale dell'applicazione, e i vari metodi, necessari per l'apertura e la scrittura di quest'ultimo, sono gestiti dalla classe *LoggerGnss*, realizzata in modo autonomo (senza ricorrere alla libreria Log4J). Il file generato viene poi inserito all'interno della cartella *logs_gnss*, che viene creata opzionalmente nella memoria esterna o interna (secondo le stesse priorità già discusse per il log principale), seguendo il percorso previsto per l'applicazione all'interno del file-system del dispositivo.

4.12.5 Post elaborazione sul PC

Al fine di poter elaborare i dati associati ai raw-measurement in un secondo momento, è stato realizzato un nodo ROS, scritto in Python, in grado di iscriversi ai topic:

/deviceName/android/gnss/navigation e */deviceName/android/gnss/measurements*. Lo scopo di questo nodo è quello di leggere i dati contenuti all'interno dei messaggi ROS relativi al GNSS pubblicati dagli smartphone e di memorizzarli all'interno di una serie di file distinti, in base al tipo di messaggio: GnssNavigation / GnssMeasurements. Inoltre, per ogni tipo vengono generati due file, nei formati "txt" e "csv". Il nome di questi file è ricavato a partire da: tipo del messaggio, *deviceName* e data/ora di creazione del file.

Il formato adottato nei file con estensione "csv", ne consente l'utilizzo all'interno di software come Matlab, ciò offre la possibilità di effettuare delle analisi sui dati e rende possibile l'utilizzo di questi file all'interno di algoritmi che lavorano in post-processing.

Capitolo V

Considerazioni conclusive

La parte finale di questa attività di tesi si propone di discutere le prove svolte utilizzando l'applicazione Android in un contesto reale, che comprende l'utilizzo congiunto con altri moduli ROS esterni e con l'ausilio di un rover. Per fare ciò è stato necessario integrare quanto fatto su Android con il lavoro svolto in parallelo dal Dott. Federico Barone, che si è occupato dello sviluppo di un'applicazione web basata su javascript in grado di comunicare con l'ambiente ROS.

5.1 Integrazione con applicazione Web

I nomi dei vari topic ROS e i tipi dei messaggi comuni tra le due applicazioni, utilizzati per la comunicazione e presentati nel capitolo IV, sono stati di volta in volta concordati, con l'obbiettivo di permettere all'applicazione web di visualizzare i dati pubblicati da un qualsiasi smartphone su cui è in esecuzione l'applicazione Android, e di rappresentarli utilizzando degli appositi grafici.
Inoltre, al fine di favorire il controllo remoto, sono stati integrati all'interno dell'applicazione web anche i vari ROS-Service associati alla fotocamera descritti nel capitolo precedente, cosicché l'utente attraverso il browser utilizzando semplicemente dei bottoni possa: cambiare la fotocamera, ruotare l'immagine e intervenire, sia sulla qualità che sulla risoluzione delle immagini trasmesse, a seconda delle diverse necessità che si possono presentare durante l'attività di controllo remoto.
L'applicazione web inoltre, supporta entrambe le modalità di trasporto immagini presentate in precedenza. È possibile selezionare l'una o l'altra nella fase iniziale, utilizzando l'apposito bottone presente sull'interfaccia all'interno del browser. Per poter utilizzare la modalità x264 occorre prima aver installato all'interno del sistema su cui è in esecuzione l'applicazione web il plugin-package: *x264_video_transport*.
Per poter visualizzare le immagini lato web, l'applicazione utilizza le funzionalità offerte dal package: *web_video_server*. Quest'ultimo tuttavia consente l'iscrizione solamente ai topic associati alle immagini di tipo *raw*. Perciò, affinché l'applicazione web sia in grado di visualizzare le immagini pubblicate sul topic x264, è necessario eseguire un nodo *"republish"* (visto nel capitolo III) in grado di convertire le immagini e di pubblicarle su un secondo topic nel formato *raw*.
A tale scopo è possibile utilizzare il file: *playback.launch*, presente all'interno del package *x264_video_transport*. Quest'ultimo deve essere eseguito specificando come parametro il nome del dispositivo Android che pubblicherà sul topic x264 seguito da

"/android/camera"[12]. A questo punto verrà lanciato un nuovo nodo: *"republish"*, che effettuerà l'iscrizione al topic: */deviceName/android/camera/video/x264* e ripubblicherà i messaggi su: */republished/deviceName/android/camera/image*. Quest'ultimo risulterà accessibile attraverso il nodo *web_video_server* e di conseguenza anche dall'applicazione web.

5.2 Test con rover Clearpath Jackal

Le prove finali con entrambe le applicazioni sono state effettuate grazie all'ausilio del rover: *"Clearpath Jackal"*, fornito dal Centro Interdipartimentale per la Robotica di Servizio: PIC4SeR, del Politecnico di Torino. Lo scopo di queste prove, oltre alla verifica del normale funzionamento delle applicazioni, ha come obbiettivo quello di osservare le diverse criticità che si possono verificare durante utilizzo in quello che potrebbe essere uno scenario tipico.
Il rover è stato pilotato tramite un browser utilizzando l'applicazione web introdotta nel precedente paragrafo, ciò ha permesso di valutare sia l'entità dei ritardi con cui le variazioni dei sensori vengono mostrate all'interno dei vari grafici, sia di verificare la distanza massima, entro la quale è possibile impartire dei comandi utilizzando una connessione Wi-Fi.

5.2.1 Configurazione

Per il test è stato utilizzato uno smartphone *Samsung Galaxy A5-2016*, fissato in orizzontale sulla parte anteriore del rover con la fotocamera posteriore orientata in avanti. La rete per la comunicazione è stata creata utilizzando un secondo smartphone (MEIZU M5) come hotspot Wi-Fi. Infine, è stato utilizzato un notebook con sistema operativo Ubuntu 16.04 e ROS Kinetic, su cui è stata avviata l'applicazione web.
Contestualmente all'avvio di quest'ultima vengono anche avviati in automatico: il master (associato all'indirizzo IP del laptop) e una serie di nodi ROS necessari per il funzionamento. A questo punto è necessario configurare sia il rover che lo smartphone. Nel primo caso, utilizzando una connessione SSH, si impostano le variabili di ambiente: ROS_MASTER_URI e ROS_HOSTNAME, che consentono al rover di comunicare con il resto del sistema, successivamente è necessario avviare i nodi che ne permettono l'utilizzo. A questo scopo è possibile utilizzare il comando:

$ roslaunch jackal_base base.launch

[12] È necessario aggiungere quest'ultimo come parametro, poiché se lo si inserisce nel file xml come elemento statico, il file non sarebbe più di tipo generico e diventerebbe inutilizzabile per quei nodi che non appartengono all'applicazione Android.

Parallelamente, sullo smartphone è necessario avviare l'applicazione Android e collegare anch'essa al resto del sistema. Per fare ciò, una volta che il dispositivo è connesso alla rete Wi-Fi creata in precedenza, nella schermata iniziale: MasterChooser, occorre inserire lo URI del master. Quest'ultimo sarà composto dall'indirizzo IP del notebook su cui è in esecuzione, seguito dalla *well-known port*:11311, su cui è in ascolto il server XMLRPC associato.

5.2.2 Osservazioni sul funzionamento dell'applicazione Android

Per quanto riguarda l'applicazione Android, è stato possibile verificare la reattività dei sensori presenti sullo smartphone e osservare la loro effettiva utilità in questo campo. Inoltre, è stato importante esaminare l'impatto sulla trasmissione dei dati pubblicati da quest'ultima nel momento in cui aumenta la distanza dall'access-point Wi-Fi o quando il rover (con a bordo lo smartphone) si muove a una certa velocità.
Nello specifico, i casi di maggiore interesse riguardano la trasmissione delle immagini. Ciò che si osserva, è che utilizzando la modalità *compressed* la qualità delle immagini è leggermente migliore rispetto a x264 se il rover si trova nelle vicinanze dell'access-point.
Poiché si utilizza il protocollo TCP, le immagini in questa modalità mantengono la qualità anche all'aumentare della distanza, senza perdite di pacchetti, tuttavia, in questo caso la diminuzione della velocità del link determina una drastica riduzione del frame-rate e un considerevole aumento della latenza. Tali ritardi determinano l'impossibilità di effettuare un controllo real-time del rover, basandosi unicamente sulle immagini mostrate dal browser.
È possibile migliorare la situazione diminuendo la qualità delle immagini (modificando il parametro di quantizzazione della compressione JPEG) o riducendo la risoluzione. Il limite della modalità *compressed* è dovuto in generale al framerate in uscita, che risulta essere troppo basso anche in condizioni ottimali. Ciò è dovuto alle tempistiche di compressione JPEG caratteristiche dell'algoritmo utilizzato, determinando una minore quantità di messaggi inviati nell'unità di tempo. La situazione peggiora ulteriormente nel momento in cui si ruota l'immagine tramite ROS-service, per le motivazioni discusse nel capitolo precedente, passando a valori che oscillano tra 5 e 7 fps.
Escludendo quest'ultimo caso, sebbene i valori indicati possano essere tollerabili nel momento in cui il rover è fermo, diventano inaccettabili quando quest'ultimo si muove con una certa velocità, in quanto non si è proprio in grado di osservare la dinamica della scena.
Per quanto riguarda la modalità x264, anche in questo caso nelle vicinanze dell'access-point non si osservano particolari criticità. Rispetto alla situazione precedente, nel momento in cui il rover si allontana, l'aumento della latenza è minimo e il frame-rate rimane inalterato intorno ai 29-30 fps, ciò è dovuto a una minor quantità di banda mediamente occupata durante lo stream video (fino a un ordine di grandezza in meno rispetto al caso *compressed* a parità di risoluzione).

Tuttavia, in questo caso, nel momento in cui aumenta la distanza dall'access-point Wi-Fi, spesso si osservano degli errori nella visualizzazione, i quali tendono ad essere propagati a causa dell'utilizzo della codifica differenziale. Questo comportamento è dovuto a diversi elementi, tra cui l'incremento nel numero di pacchetti corrotti, riconducibile alle caratteristiche fisiche delle connessioni wireless; ciò determina una maggiore quantità di tempo necessaria per l'invio delle informazioni dovuta alle ritrasmissioni[13] in caso di errori, se questo tempo eccede il valore del timeout utilizzato lato ricevitore nell'attesa dei dati associati al frame da riprodurre, interverranno degli algoritmi di occultamento con lo scopo di sostituire il pacchetto mancante, mentre quello in ritardo verrà considerato perso e successivamente scartato.
Per migliorare questa situazione, si potrebbe aumentare la dimensione del buffer di *playout*, ma in questo caso si ricadrebbe nello stesso problema osservato nella modalità *compressed*, ovvero si osserverebbe un aumento della latenza.
Anche per la modalità x264 è possibile intervenire sui parametri utilizzati dalle classi Android attraverso i ROS-service: diminuendo la risoluzione o la qualità dei frame, ottenendo in questo modo un bit-rate medio molto basso (nettamente inferiore al minimo ottenibile con la compressione JPEG). Inoltre, è possibile intervenire anche sui parametri dell'encoder durante la configurazione di quest'ultimo, ad esempio modificando i valor associati al GOP o il numero di frame di tipo B utilizzati durante la codifica (in quest'ultimo caso è possibile diminuire notevolmente il bitrate medio, a discapito della latenza e di una maggiore quantità di lavoro per la cpu).
Con la modalità x264, il framerate (29-30 fps) risulta essere compatibile con l'utilizzo del rover in movimento, ciò è stato possibile grazie all'utilizzo della Framework API: MediaCodec, la quale, sfruttando le componenti hardware presenti nel dispositivo è in grado di eseguire la codifica ad una velocità analoga a quella con cui le immagini vengono acquisite. Occorre precisare che il framerate non sempre raggiunge i valori indicati sopra, ciò può essere dovuto al fatto che lo stesso chip utilizzato da MediaCodec, in determinati momenti venga utilizzato anche da altre applicazioni, oppure a logiche di risparmio energetico che ne limitano il funzionamento.

5.3 Contributi

Durante questa attività di tesi, è stato possibile approfondire l'utilizzo del framework ROS per svolgere l'attività di monitoraggio. Questa prevede il coinvolgimento di diversi elementi e deve tener conto di numerosi fattori. Sia per quanto riguarda parte incentrata sulla gestione delle immagini che in quella dedicata allo sviluppo dell'applicazione Android è stato necessario affrontare le problematiche legate acquisizione dei dati e alla loro successiva trasmissione, cercando di sfruttare le opportunità offerte da ROS nello svolgere queste operazioni.

[13] Infatti, con l'utilizzo di TCP come protocollo di trasporto affidabile, i pacchetti non possono essere corrotti o persi, poiché in questi casi intervengono dei meccanismi di ritrasmissione. Tuttavia, questi ultimi determino un rallentamento complessivo nell'invio dei dati.

La maggior parte delle applicazioni Android basate su ROS consentono di utilizzare lo smartphone per il controllo a distanza di un dispositivo robotico, attraverso controller virtuali, comandi vocali e *gesture*, o di monitorare quest'ultimo, ad esempio visualizzando sullo schermo dello smartphone le immagini acquisite dalla camera del robot. Tuttavia, poche applicazioni prevedono l'utilizzo opposto, ovvero lo sfruttamento della grande varietà dei sensori presenti negli smartphone odierni per fornire informazioni aggiuntive durante il monitoraggio a distanza.
In tal senso l'applicazione: *PIC4SeR Monitoring*, consente di acquisire svariate informazioni e grazie all'architettura ROS è in grado di renderle disponibili in modo relativamente semplice agli altri nodi del sistema. Attraverso quest'ultima, nel momento in cui lo smartphone si trova a bordo di un altro dispositivo, è possibile conoscere: la sua posizione, il suo orientamento spaziale, se esso è in movimento (accelerato o costante); è possibile ricavare indicazioni sulla qualità del segnale della rete cellulare in una determinata zona sorvolata da un drone oppure ottenere informazioni dettagliate sulla qualità della connessione Wi-Fi.
Infine, attraverso la pubblicazione delle immagini in tempo reale è possibile pilotare il robot da uno degli altri nodi del sistema, attività supportata anche con l'ausilio dei ROS-Service dedicati.
Durante lo sviluppo dell'applicazione sono stati definiti nuovi tipi di messaggi ROS per il trasporto di informazioni associate a diverse classi Android. Tra questi, quelli relativi ai dati Wi-Fi e al comparto telefonico (GSM, CDMA, LTE). Inoltre, con la definizione dei messaggi associati al GNSS, è possibile sfruttare ROS per la condivisione e l'elaborazione dei raw-measurement, che come si è visto, potrebbero costituire una nuova frontiera nello sviluppo di applicazioni di posizionamento ad alta precisione, dalle quali molti sistemi robotici potrebbero trarre grande vantaggio.
L'applicazione fornisce anche un esempio di utilizzo della codifica H.264 per trasmettere le immagini acquisite dalle fotocamere dello smartphone, sottoforma di stream video, attraverso appositi messaggi ROS: x264Packet, migliorando le prestazioni. Questi ultimi vengono utilizzati in modo complementare con le classi del plugin-package: *x264_video_transport*, che come si è visto, nasce in risposta all'esigenza di una maggiore integrazione del subscriber x264 con le funzionalità offerte dall'applicazione Android.

5.4 Possibili sviluppi futuri

Il lavoro svolto può rappresentare il punto di partenza per ulteriori miglioramenti, al fine di sfruttare a pieno le numerose opportunità offerte dai dispositivi odierni. Vi sono diverse idee da cui è possibile partire, è l'obbiettivo di questo paragrafo conclusivo consiste nel discutere alcune di esse.

Utilizzo del microfono presente sullo smartphone

Un buon punto di inizio potrebbe essere l'esplorazione delle possibilità offerte dall'utilizzo del microfono presente nello smartphone. Quest'ultimo infatti, al pari degli altri sensori, permette di raccogliere dei dati che possono essere utilizzati in modo intelligente per aggiungere delle funzionalità. Ad esempio, se non ci si limita alla semplice raccolta di segnali audio, sarebbe possibile utilizzare il microfono dello smartphone per impartire dei comandi vocali al robot. Questi ultimi potrebbero avviare l'esecuzione di un nodo responsabile del movimento che pubblicando dei messaggi di tipo *sensor_msgs/twist* su un classico topic: *cmd_vel*, permetterebbero al robot di spostarsi in una determinata direzione. Oppure sarebbe possibile richiedere verbalmente l'esecuzione di un determinato compito attraverso la traduzione del comando vocale in uno specifico ROS-Service.

In questo caso sarebbe necessario approfondire quelle che sono le tecniche e i messaggi per la trasmissione di dati audio attraverso il sistema ROS, oltre a verificare l'esistenza di nodi in grado di effettuare l'elaborazione di queste informazioni.

Cooperative Positioning

Un'altra opportunità offerta dall'utilizzo degli smartphone Android in questo ambito è costituita dalla possibilità di far cooperare più dispositivi, che si scambiano informazioni tra loro, senza la necessità di trasmettere dati al PC. In tal senso si potrebbero approfondire le tecniche di cooperative-positioning, accennate parlando dei GNSS *raw-measurements*.

Possibilità di sfruttare la rete pubblica per la trasmissione dei dati

Una delle criticità, che è apparsa ancor più evidente durante le sperimentazioni con il rover: *Clearpath Jackal*, è costituita dalle limitazioni fisiche imposte dal Wi-Fi. Infatti, utilizzando quest'ultimo con un singolo access-point, non è possibile coprire grandi distanze (con la tecnologia odierna è possibile arrivare al massimo a un centinaio di metri se ci si trova all'esterno, in condizioni ottimali). Se si ha la necessità di coprire spazi maggiori, occorre posizionare più access-point sovrapposti, lungo tutta l'area interessata.

Un'alternativa a questa soluzione sarebbe la possibilità di sfruttare la connessione ad Internet fornita dalla rete cellulare (LTE) per la trasmissione dei dati. Tuttavia, ciò introduce numerose problematiche. Infatti, come si è già visto nel secondo capitolo, il funzionamento di ROS si basa sull'utilizzo di indirizzi IP fissi e ben noti [14], perciò il problema principale sta nel fatto che tutti i dispositivi mobili passando da una cella all'altra della rete mobile cambiano il proprio indirizzo, quindi sarebbe necessario avere un meccanismo che comunichi al master questa variazione. Se anche non fosse presente questo problema, si deve tener conto del fatto che praticamente tutti gli ISP, attualmente utilizzano soluzioni di indirizzamento basate su uno o più livelli di NAT

[14] I publisher comunicano il proprio indirizzo IP e il topic al master, che successivamente informerà gli eventuali subscriber sulla base di queste informazioni, con il presupposto che non cambino

e ciò rende i dispositivi mobili presenti all'interno della loro rete, irraggiungibili dall'esterno. Quest'informazione risulta essere cruciale, infatti si ricorda che in un sistema ROS è sempre il subscriber ad aprire la connessione TCP con il publisher[15]. In questo caso, anche se lo smartphone fosse in grado di raggiungere il master attraverso l'indirizzo pubblico del computer remoto su cui quest'ultimo è in esecuzione, in applicazioni come quella presa in esame (in cui è sempre il dispositivo Android a pubblicare i dati), sarà comunque il subscriber remoto a dover aprire la connessione, ma come già detto, a causa del NAT, questo normalmente non è possibile.
L'indirizzo variabile rende complicato anche l'utilizzo di un'eventuale soluzione VPN. Infatti, sebbene in questo caso si risolverebbe il problema degli indirizzi interni a quest'ultima, in quanto lo smartphone avrebbe sempre lo stesso indirizzo appartenente nella rete interna (e perciò risulterebbe essere raggiungibile dagli altri nodi), il VPN Gateway dovrebbe comunque di volta in volta cambiare indirizzo esterno e si ripresenterebbe lo stesso problema già visto in precedenza, ovvero dover comunicare dall'altro lato la variazione.
Lo studio di una soluzione a queste problematiche consentirebbe di ampliare notevolmente gli scenari in cui un'applicazione di questo tipo potrebbe essere utilizzata.

Utilizzo attivo dei sensori dello smartphone

Un ultimo scenario di sviluppo che potrebbe essere interessante in questo ambito, sarebbe un approfondimento sulla possibilità di utilizzare i dati forniti dai sensori dello smartphone non solo ai fini del monitoraggio passivo, ma anche per il controllo attivo e automatizzato di un dispositivo robotico. Per fare ciò, sarebbe innanzitutto necessario capire se i sensori comunemente usati (e a basso costo), presenti negli smartphone, possiedono la sensibilità e la reattività necessarie per questo tipo di utilizzo. In tal caso si aprirebbero un gran numero di scenari interessanti. L'accelerometro ad esempio potrebbe essere utilizzato per fare in modo che un rover con a bordo il telefono mantenga una velocità o un'accelerazione costante e allo stesso tempo i dati forniti dal sensore potrebbero essere sfruttati per gestire in automatico la frenata del mezzo entro un determinato numero di metri.
È anche possibile immaginare soluzioni in cui i dati del giroscopio potrebbero essere utilizzati da un nodo per stabilizzare automaticamente un drone. Infine, la maggior parte dei telefoni possiede un sensore di prossimità. Sebbene quest'ultimo attualmente fornisca un valore piuttosto grossolano (oltre ad essere monodirezionale anziché a 360°), il che lo rende inadatto per evitare gli eventuali ostacoli in tempi utili, potrebbe comunque essere utilizzato in determinate situazioni in cui il robot si muove lentamente, ad esempio per fermarlo prima che vada a sbattere contro un muro o uno spigolo.
Ovviamente, tornando alla premessa, prima di poter pensare allo sviluppo di queste funzionalità, occorrerebbe fare uno studio di fattibilità per stabilire se i sensori a disposizione sono adatti per essere utilizzati a questo scopo. Inoltre, queste soluzioni

[15] Si fa riferimento alla fase di negoziazione preliminare tra i nodi, non alla trasmissione

prevedono che gli eventuali nodi per il controllo siano eseguiti direttamente sullo smartphone o, in alternativa, sulla cpu del drone/rover su cui questo e montato. Così da evitare i vari ritardi e problemi già osservati nelle trasmissioni a lunga distanza, che in applicazioni simili risultano essere ancor più critici e determinanti.

Bibliografia

[1] GSA GNSS Raw Measurements Task Force, *Using Gnss Raw Measurements on Android Devices*, 2017

[2] Quigley M. et al., *ROS: an open-source Robot Operating System. In IEEE International Conference on Robotics and Automation(ICRA)*, Workshop on Open Source Software, 2009.

[3] Mayoral Vilches V., *ROS Technical Overview* [Online], 2014. Disponibile all'indirizzo: http://wiki.ros.org/ROS/Technical%20Overview

[4] Martinez Romero A., *ROS Concepts* [Online], 2014. Disponibile all'indirizzo: http://wiki.ros.org/ROS/Concepts

[5] Thomas D., *Wiki: ROS/TCPROS* [Online], 2013. Disponibile all'indirizzo: http://wiki.ros.org/action/recall/ROS/TCPROS?action=recall&rev=12 . (Consultato il 15/12/2018)

[6] Conley K.*, Wiki: ROS/UDPROS* [Online], 2009. Disponibile all'indirizzo: http://wiki.ros.org/action/recall/ROS/UDPROS?action=recall&rev=4 . (Consultato il 15/12/2018)

[7] Redelkiewicz J., Sunkevic M., Crosta P., Navarro- Gallardo M., Bonenberg L., *Opportunities and practical use of Android GNSS Raw Measurements* [Online], 2018. Disponibile all'indirizzo: *https://mycoordinates.org/opportunities-and-practical-use-of-android-gnss-raw-measurements/*

[8] Van Diggelen F., *GNSS 102 Raw Measurements from Phones, 2018*

[9] Google, *Raw GNSS Measurements* [Online], Disponibile all'indirizzo: *https://developer.android.com/guide/topics/sensors/gnss.html*

[10] Google - *MediaCodec,* [Online], Disponibile all'indirizzo: https://developer.android.com/reference/android/media/MediaCodec (Consultato il 20/01/19)

[11] Robitza W., *HWAccelIntro* [Online], 2019. Disponibile all'indirizzo: https://trac.ffmpeg.org/wiki/HWAccelIntro (Consultato il 02/02/19)

[12] Ademovic A.*, An Introduction to Robot Operating System: The Ultimate Robot Application Framework* [Online], 2015. Disponibile all'indirizzo: https://www.toptal.com/robotics/introduction-to-robot-operating-system

[13] Quigley M., Gerkey B., W. D. Smart , *Programming Robots with ROS*, O'Reilly Media, Inc., pp. 193-199, 2015

[14] O'Kane J. M., *A Gentle Introduction to ROS*, Jason Matthew O'Kane, 2018.

[15] Lamoine V., *Wiki: image_transport* [Online], 2017. Disponibile all'indirizzo: http://wiki.ros.org/image_transport, 2017

[16] Ha M. Do, Mouser C. J., Sheng W., *Building a Telepresence Robot Based on an opensource Robot Operating System and Android*, 2012.

[17] Oros N., Krichmar J. L., *Android Based Robotics: Powerful, Flexible and Inexpensive Robots for Hobbyists, Educators, Students and Researchers*, 2015.

[18] fyhertz, *libstreaming NV21Convertor* [Online], 2017. Disponibile all'indirizzo: https://github.com/fyhertz/libstreaming/blob/master/src/net/majorkernelpanic/streaming/hw/NV21Convertor.java . (Consultato il 22/01/19)

[19] Botev A., Teacy L., *x264_image_transport* [Online], 2014. Disponibile all'indirizzo: https://github.com/orchidproject/x264_image_transport . (Consultato il 30/01/19)

[20] Serrano D., *Introduction to ROS – Robot Operating System*, 2015. pp. 2-10

[21] Vincent Rabaud, *image_proc* [Online], 2015. Disponibile all'indirizzo: http://wiki.ros.org/image_proc

[22] Allevato A., *Using CvBridge To Convert Between ROS Images And OpenCV Images*, [Online], 2017. Disponibile all'indirizzo: http://wiki.ros.org/cv_bridge/Tutorials/UsingCvBridgeToConvertBetweenROSImagesAndOpenCVImages (Consultato il 30/01/19)

Ringraziamenti

Giunto alla conclusione di questo percorso di studi, desidero ringraziare tutti coloro che mi sono stati vicini, aiutandomi nel raggiungimento di questo importante traguardo.
Vorrei innanzitutto ringraziare il Prof. Enrico Masala, per essere stato sempre presente, per i suoi consigli e per il prezioso aiuto fornitomi durante l'attività di ricerca e nella stesura di questa tesi.
Grazie anche a tutti i colleghi di università, con cui ho affrontato questi anni di studi, per avermi dato l'opportunità di un confronto, che mi ha consentito nel tempo di migliorare, e per i momenti di difficoltà e gioia che abbiamo condiviso.
Un caloroso ringraziamento ai miei amici, sia a quelli di una vita, che nonostante la distanza mi sono sempre stati vicini, facendomi apprezzare ancor di più i momenti passati insieme, che a quelli nuovi, conosciuti durante questa esperienza a Torino, con i quali ho trascorso delle splendide serate.
Non potrei non ringraziare il mio amico Giovanni, con il quale ho condiviso gioie e ostacoli di questo percorso, per avermi sempre supportato oltreché sopportato, in questi anni passati nella stessa casa e per essere stato un importante punto di riferimento su cui ho potuto sempre contare.
Vorrei ringraziare mio nonno per il suo costante supporto e per non aver mai mancato di farmi sentire la sua vicinanza e il suo affetto, al contempo tramandandomi la sua esperienza e suoi consigli, grazie ai quali non ho mai perso di vista il mio obbiettivo.
Infine, un ringraziamento speciale ai miei genitori e mia sorella per aver sempre avuto fiducia in me, standomi accanto e dandomi il sostegno necessario nei momenti più difficili, assicurandosi che non mi mancasse mai niente.
Senza di loro adesso certamente non sarei qui e per tale motivo è proprio a loro che vorrei dedicare questo lavoro, sperando possa renderli orgogliosi.